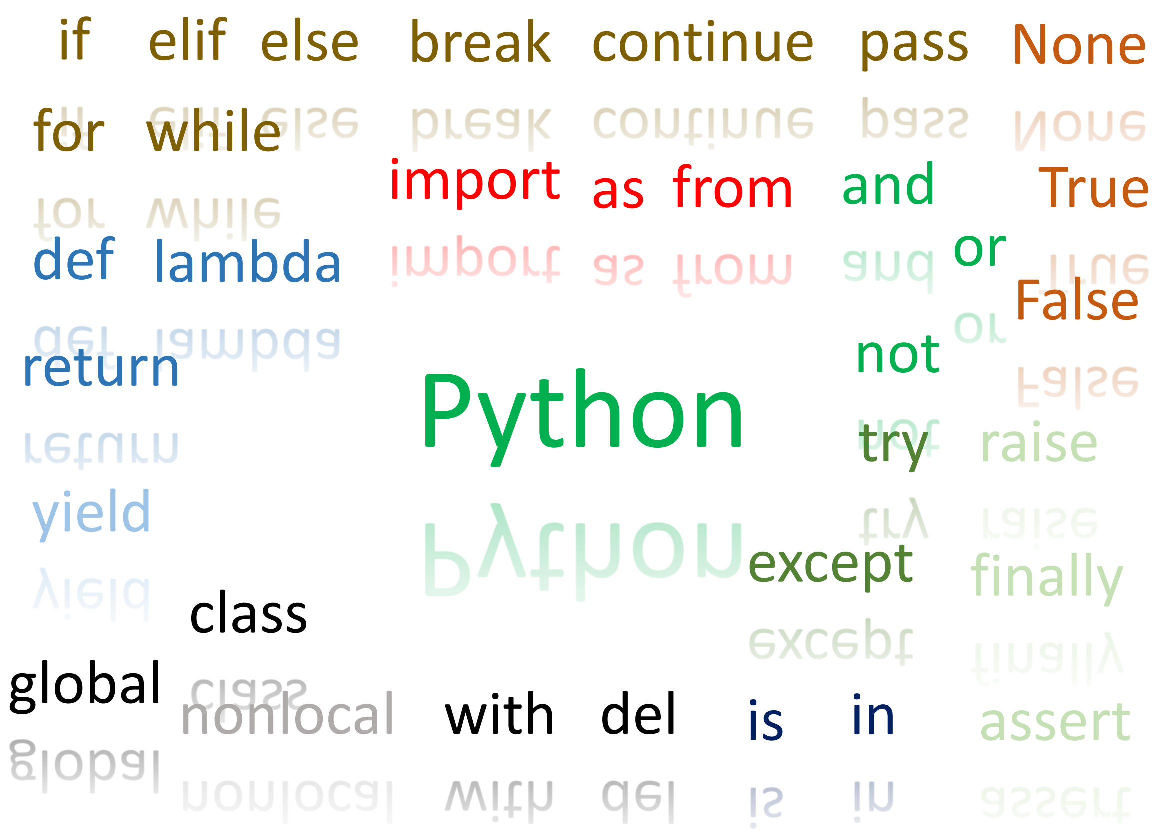
Programming Practice Python 2021



These are reserved Python keywords. Faded words will not be covered in this book.

Hajime Kita, Institute for Liberal Arts and Sciences

Yoshitaka Morimura, Institute for Information Management and Communication

Masako Okamoto, Center for Promotion of Excellence in Higher Education

Kyoto University

Version 2021/10/08

Table of Content

Next Chapter Table of Contents

Table of Figures

Table of Tables

Table of Programs

Table of Exercises

Next Chapter Table of Contents

0. Foreword

This textbook was written for a (Python) programming exercises course as part of Kyoto University's Liberal Arts and Sciences courses.

0.1 Objectives and goals

The objectives and goals of this course are as outlined below.

0.1.1 Objectives

Python is a programming language that has many practical uses and is easy for beginners to learn. In recent years, its use in academic research has steadily increased. In this course aimed at beginners, students will learn to program using Python through various exercises.

0.1.2 Goals

- Students will learn the fundamentals of how to use Python to execute programs.
- After learning about the functions and formats of the fundamental components of a Python program, students will be able to use the examples to put together their own programs.
- Students will be able to design, implement, and test their own simple Python programs.

0.2 Reasons for writing an entirely new textbook

This text was based on the courses from the 2018-2020 academic years. There are already a large number of introductory texts for Python, however, due to the reasons detailed below, it was decided to add yet another text into the mix.

- This course does not aim to introduce the reader to Python, but rather to teach students how to write (to become able to write) their own programs in Python. Many texts have the tendency to merely be an introduction to the language itself.
- For beginners, learning a programming language is simply the overcoming of various mistakes. Even mistakes that seem severe when you are a beginner are often completely forgotten once you have gained some programming experience. Through practical exercises, this course aims to keep the spots that beginners tend to make mistakes in mind, and to tailor the explanations in a manner that helps them overcome these mistakes.
- Related to the above, learning a programming language consists of writing and executing actual programs by yourself. This text includes directions for actual programming exercises.

While this text serves as a guide for beginners that explains the fundamentals of programming using Python, it is not a comprehensive introduction of Python's language specifications. It is recommended for students to prepare other learning materials on Python and take them to classes.

Next Chapter Table of Contents

0.3 Humanities majors: you can do it!

It seems that there are many humanities students that think of programming as something that should be left for the science majors. Programming is indeed a technique to control the complex machines known as computers, and source code can often look like numerical formulas. However, in actuality, programming merely delegates to machines work that humans can perform themselves. It is important to firmly understand the actions performed by humans because in this respect humanities students can gain an edge. Many humanities students have taken this course in the past and received passing grades.

0.4 Regarding the organization of this text

This text was organized based on the courses from the 2018-2020 academic years. As such, it is structured such that you can work through the exercises in order from the front of the book. Topics from the classes that represent a digression from the topic at hand are compiled as independent “Columns.”

0.5 Notation

It is most effective for students to learn using a two-pronged approach. One side of this approach will see students learning by testing simple commands line by line in an interactive environment known as the Python shell. The other side will have students write larger programs (scripts) in an editor and execute them as a batch.

Features that students are to try out in the Python shell are represented in the text using the red K2PFE font seen here:

```
a = 1 + 2
```

The results obtained from executing these commands are shown in blue as seen here:

```
3
```

Please try these out for yourself as you learn.

On the other hand, larger programs will be shown using the table format with three (or two) columns seen below. Input the source code into the editor and execute the program to see the results. In this representation, a character '␣' is placed instead of space ' ' so as to make them visible.

Row	Source code	Explanation
1 2	a␣=␣1␣+␣2 print(a)	'␣' is used instead of space ' ',

Notices in typing actual programs referring the source code lists:

- Spaces in the source code are replaced with '␣' so as to make them visible.
- Ordinally fonts in Windows, a glyph of yen (¥) is used for backslash code (\). K2PFE font uses a glyph of backslash (\) instead.

0.6 Warning regarding copying and pasting

The source code that appears in this text was formatted in Word before being converted to PDF format. In the PDF document, spaces are not saved, and there are instances in which the characters will be automatically changed. As such, please be aware that there are instances in which simply copying and pasting from the PDF will lead to program errors.

0.7 Edits for later versions (2020 and later)

In the 2021 version, some misprints were corrected, some hard-to-read portions were edited, and some additional explanations given during the 2020 course were added. In addition, a 'list' introduced in Chapter 10 of the 2020 version was moved to Chapter 4, and the handling of the 'for' statements that are the subject of this list is improved. Masako Okamoto, who has assisted greatly with the planning of the courses has joined as a coauthor starting with this version.

Further, in 2021 edition, for typesetting of source code, we use K2PFE font newly developed.

Notes for English Version

This textbook is a translation of the Japanese version referred to as “Puroguramingu-Enshuu Python 2021 (プログラミング演習 Python 2021).” The original version cited several Japanese references. We have retained some quotes and references in Japanese.

Acknowledgements

Since the publication of the 2019 version following the 2019 academic year, we have received many helpful opinions as well as identifications of misprints. Our deepest thanks for all of those who shared their opinions with us.

K2PFE font used in this textbook were developed as a collaborative research by Professor Akihisa Tatsumi, Part-time Lecturer Maya Kusunoki of Kyoto City University of Art, Dr. Tamaki Motoki of Kyoto University and Hajime Kita one of the authors of this textbook. It is partly supported by JSPS KAKENHI Grant Number JP 21K028 80.

English translation from Japanese version was supported by Center for Innovative Research and Education in Data Science, Institute for Liberal Arts and Sciences, Kyoto University as one of its activities in development of learning materials and their publication.

1. Computers and Programming

1.1 Objectives for this chapter

- Understand how computers operate in general, and how programs factor in.
- Learn about the role of programming languages in programming.
- Learn about various uses and applications for writing programs.
- Learn about how to study programming.

Exercise 1-1Motives for taking this course

Please answer the following questions.

1. Why did you decide to take this course?
2. Why do you want to learn programming?
3. Why do you want to learn Python?
4. Please indicate whether you have any previous experience with studying programming (Yes or no). If yes, how much?
5. For those with previous experience studying programming, what programming language(s) did you use?

1.2 Computers and programs

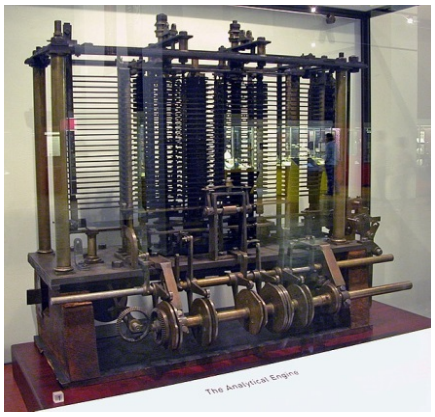
1.2.1 Machines that are run by programs

You have probably heard of the Jacquard machine sometime in a history course. Textiles are woven by passing a weft thread through a warp thread. One can weave a specific design by varying which warps are run on top of the weft and which are run beneath. A Jacquard machine is a machine that can correctly weave the thread provided it is given instructions regarding which way to run the thread in the form of paper with holes punched into it (referred to as punch cards). By binding together many punch cards and feeding them into the machine in order, one can create complex designs. In order to create a different pattern, one simply needs to swap out the punch cards.

One can still find Jacquard machines actively used in the Nishijin district in Kyoto, where weaving is a booming local industry. Figure 1-1

Charles Babbage (1791-1871) was an inventor from Great Britain who set out to develop a mechanized calculator. Starting with a mechanism that automatically generated number tables from sequences of differences, Babbage took inspiration from the Jacquard machine and attempted to create a mechanized calculator (The Analytical Engine) that is run by a program. Unfortunately, he was never able to complete it; however, he is viewed as a pioneer in the field of computer science for his attempts to make a mechanism that performs calculations according to a program.

Jacquard machines and punch cards
Taken at Fukuoka Weaving in Kyoto



Babbage's Analytical Engine

Figure 1-1 Jacquard machine and Analytical Engine

https://commons.wikimedia.org/wiki/File:AnalyticalMachine_Babbage_London.jpg

1.2.2 Computers are composed of "switches" that run by electricity

In Babbage's era, complex mechanical movements could only be attained by using gears or something similar. Later on, it became possible to create mechanisms that use electricity to fit in another electrical switch. One of these was to use an electromagnet to mechanically move (**relay**) an electrical contact. Attempts were actually made to use these electrical relays to make computers. However, while this works from an electrical standpoint, the mechanical movement that accompanies it was far too slow.

After this, computers were developed that utilized **vacuum tubes**. These were tubes in which electrons flowing between an electrode (cathode, anode) are controlled by the voltage applied to a different electrode placed between them. Vacuum tubes had the advantage of fast electronic movement, however, the fact that a filament was needed to add heat in order to make the cathode emit electrons meant they had short lifespans.

Following this, **transistors** were invented. Transistors are devices that enable behavior similar to that of a vacuum tube to occur within the solid matter of a semiconductor. Transistors have a long lifespan, are very small, and do not consume much electricity. At this point, computers that consisted of a large number of discrete transistors wired together were made.

Finally, **integrated circuits** were developed. These circuits are printed as an entire unit and consist of a lot of transistors and wiring on top of a single semiconductor chip. Integrated circuits are much smaller and much more cost efficient when compared to normal electronic circuits. Integrated circuit technology was then used to develop microprocessors, which are essentially the main component of a computer (all contained in a single chip). Microprocessors are the breakthrough that made cheap, small computers that anyone can use, such as PCs and smartphones, into a reality.

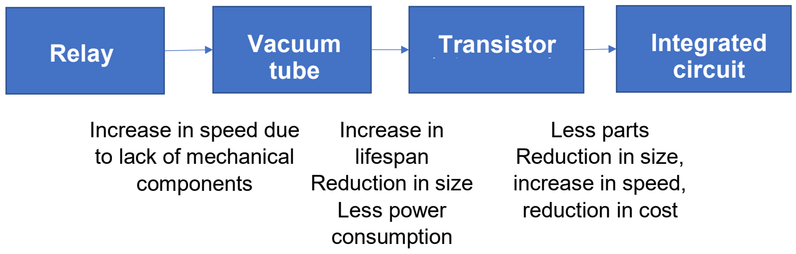


Figure 1-2 Progression of logic gates

Next, we will take a look at the amazingly rapid progression of the number of transistors that could be integrated onto a single semiconductor chip (degree of integration), which has increased by over a million times in 40 years. Technological innovations that improve efficiency by orders of magnitude effected not only integrated circuits, but storage capacity and communication speed as well. It is through technological innovations like this that you are able to use your smartphone to enjoy YouTube videos in the modern era.

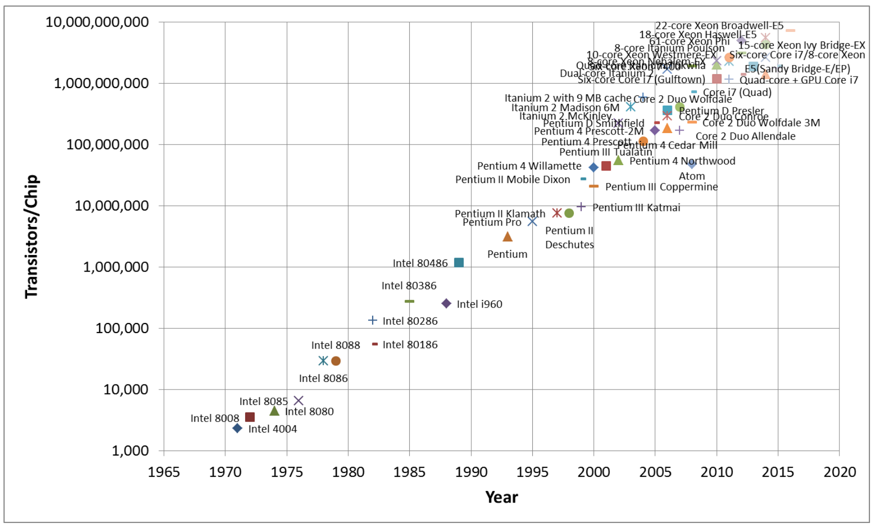


Figure 1-3 Microprocessor transistor counts

https://en.wikipedia.org/wiki/Transistor_count#Microprocessors

Plot of a selection of Intel processors (Accessed Jan. 2, 2017)

1.3 Computer structure

1.3.1 Stored-program computer

So just how do modern computers carry out complex data processing?

The actions a computer (the hardware itself) can perform at one time are very simple, and complex work like data processing is merely a combination of these simple actions. This combination of actions is expressed as a program and put into action.

In modern computers, programs are stored in memory just like the data they handle. They are read at rapid speeds and then executed. Computers that work in this manner are called "stored-program computers." [1] All computers used today, from the small microprocessors used in home appliances to giant supercomputers, are stored-program computers.

By changing programs according to the type of work that needs to be done, the same computer (hardware) can be used to accomplish a wide variety of tasks.

Programming is the act of writing out, in the form of a program, the data processing steps that you wish to perform.

1.3.2 The Components of Computers and Their Functions

The main components of a computer's hardware are the CPU and the memory.

The CPU contains the following major elements of a computer.

- Mechanism(s) to take in and parse commands from the memory
- Counters that point to the memory addresses of the program(s) that are currently being run
- Device(s) to preserve data (registers)
- An arithmetic-logic unit (ALU) that functions to perform arithmetic and logical operations on variables

The fundamental operations that a computer can perform are the following simple tasks.

- Configure a program and data used in the program in the memory (by some method).
- Provide the CPU with an execution start location forthea program
- The CPU repeats the following:

1. Reads one step of the program from the memory and performs calculations or move data according to the instructions.
   - ✧ The result of the calculation can also be stored in the memory.
   - ✧ Input/output can also be performed.
2. The CPU advances the instruction being executed to the next location.
   - ✧ In some cases, the executed location can change based on the program.

[1] In an early computer called ENIAC, calculation settings were not done in stored-program fashion; rather, they were set by altering the wiring of the cables. However, upon developing its successor, EDVAC, the stored-program method was proposed. As the report proposing it was submitted by von Neumann, it is sometimes referred to as the von Neumann architecture.

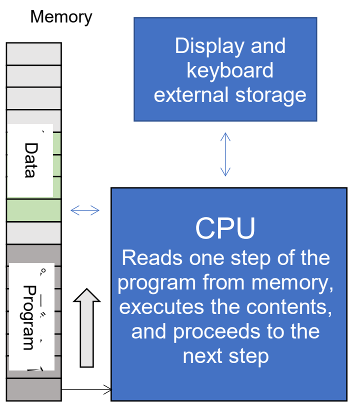

Memory
Data
Program
Display and keyboard external storage
CPU
Reads one step of the program from memory, executes the contents, and proceeds to the next step

Figure 1-4 Computer (hardware) organization

In order for the hardware (CPU + memory) to be able to easily process commands at high speeds, the commands that can be executed in actual computers are limited to extremely simple ones. These commands are referred to as machine code.

1.4 Programming languages

Programming complex actions in machine code is profoundly difficult. Programming languages were the solution devised in order to solve this problem, and they are created in the following manner.

- Determine rules that enable you to write programs in a manner that are more easy-to-understand for people (**decide upon programming language specifications**). Generally something that looks less like machine code and closer to numerical formulas
- Create a program that can execute any program (source code) written in accordance with the rules of the language (**build a language processor**)

Essentially, you write a program in the programming language, and the written program is executed using the language processor. In simpler terms, an language processor + a computer effectively yields a virtual computer that can execute any program written in a programming language.

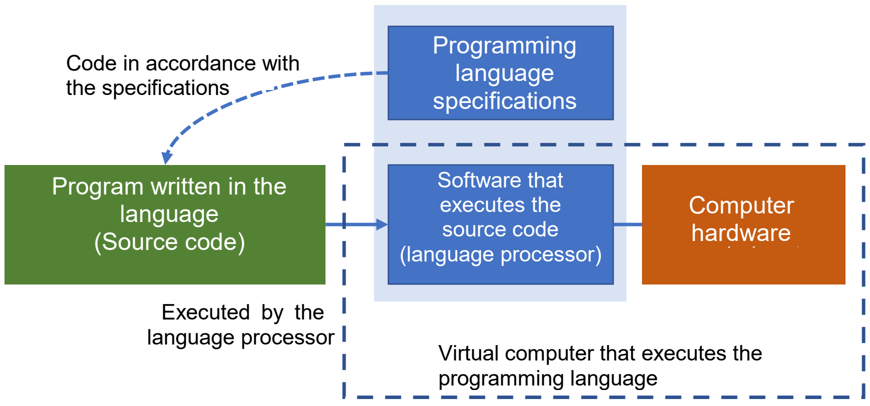

Code in accordance with the specifications
Programming language specifications
Program written in the language (Source code)
Software that executes the source code (language processor)
Computer hardware
Executed by the language processor
Virtual computer that executes the programming language

Figure 1-5 Programming languages and language processors

1.4.1 Various programming languages

As shown in the table below, many programming languages have been created and are in common use.

FORTRAN	COBOL	ALGOL	Pascal	PL/I
BASIC	C, C++, C#	Java	Go	Swift
Perl	Ruby	Python	JavaScript	LISP
Haskel	R	Matlab	ProLog	Scratch

In addition, there are many things that strongly resemble programming languages that are often used together with them. For example, HTML which codes for web pages, CSS which codes for the page's style, XML and JSON which code for data, and SQL which codes for database inquiries.

Why are there so many programming languages being used? Why are they not just all combined into one?

The advancement of computing requires the overall improvement of the programs that are developed. Consequently, the ways of thinking required to effectively code as well the programming languages based in these ways of thinking developed over time. The desire to write programs more easily, more quickly, and more securely is ever-present. There is a demand for programming languages that are specially tailored to specific uses. New programming languages have been developed and the specifications and language processors for specific programming languages have been altered with these facts in mind.

On the other hand, the use of software developed in a certain programming language as well as the programmers who wish to code in that language create demand for its continued usage. Discontinuing old languages can be difficult. FORTRAN, a programming language that is used in scientific computing, is the oldest programming language, yet still sees use following various modifications.

Some programming languages have been developed by software companies, while others have been created by individuals or as a community effort. Certain programming languages are developed by companies strictly for-profit and require that you buy their language processors. Additionally, there are cases in which languages are developed due to the needs of a company and the language processors are distributed freely with their source code made public.

1.4.2 Composition of a language processor

There are a few different ways to compose programs (language processors) that can process and execute programs (source code) written in a programming language. The three variations are as follows:

1) Compiler

Executes the source code and then translates (compiles) it into machine code. The resulting machine code is then executed. Compiling takes time, however the compiled executable program (machine code program) can be executed very quickly.

2) Interpreter

Interprets the source code line by line, simulating the operations. Execution speeds can be slow as interpreting the source code takes time; however, interpreters are so flexible that they can be used for various purposes such as interactive usage.

3) Intermediate representation (IR)

Lying somewhere in between compilers and interpreters, IR translates the source code not into the CPU’s machine code, but rather into machine code (intermediate representation) for a hypothetical virtual computer used for that language. This IR is then executed using an interpreter (virtual machine). Java and Python use this type of translator.

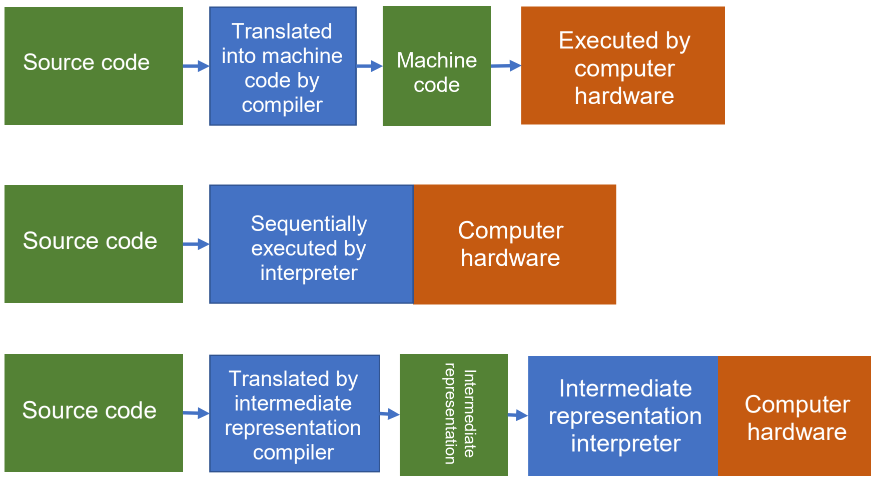

Source code
Translated into machine code by compiler
Machine code
Executed by computer hardware
Source code
Sequentially executed by interpreter
Computer hardware
Source code
Translated by intermediate representation compiler
Intermediate representation
Intermediate representation interpreter
Computer hardware

Figure 1-6 Types of language processors

1.5 Python

1.5.1 History of Python

In 1989, Guido van Rossum began working on Python. Version 2.0 was released in 2000, and version 3.0 was released in 2008.

Note: Python version 3 is not backward compatible with version 2 (version 3 does not contain the specifications of version 2). As such, both versions tend to be run concurrently in order to ensure that programs written with version 2 can run properly.

Note: Python generally comes pre-installed on Mac and Redhat Linux, but in some cases version 2 is the one that is installed. In order to use version 3, you may need to install the version 3 language processor separately and ensure the correct version is being used at all times.

1.5.2 Characteristics of Python

- Easy for beginners to learn yet capable of advanced programming
- Can be used for a wide variety of applications
- Libraries for scientific computing (NumPy, scipy, matplotlib, pandas, etc.) are being developed by many people.
- In recent years, interest in data science and artificial intelligence (machine learning) techniques has increased dramatically. Python has been growing in popularity due to its abundant libraries for these applications.

1.5.3 Python distribution packages

There are several Python language processors in development, and as a result there are many packages that contain different combinations of things such as development environments, and libraries. For this class, we will use the following two packages:

- Python: A distributable from the designers of Python; uses CPython, which is coded in the C language, as its language processor.
- Anaconda: Contains modules for scientific computing wrapped into CPython as a single package. We assume that you will be using this package in this class.

1.6 Various applications

There are likely some of you who have specific applications in mind as reasons for why you want to learn Python. Let’s look at some of its various potential usages.

1.6.1 Applications for personal computers (PCs)

Programs that run on PCs can largely be divided into two groups based on their operating environments.

- CUI programs. These programs operate via the Windows command prompt or something similar, take text input from the keyboard, and output text onto the screen. These programs are relatively easy to learn due to the simplicity of input and output, but they also tend to not be very user-friendly.
- GUI programs. These programs operate within a window and are controlled via buttons or similar objects within the window. GUI programs enable the user to operate the program in a manner they are likely already accustomed to, and images can also be included. However, the number of things that you have to program is usually much higher.
  Reading and writing files, network management, and many other actions require similar programming in both CUI and GUI programs.

A few examples of using potential applications are given below.

- Scientific computing and numerical simulations
- Data processing and analysis of numbers, strings of text, and images
- Video games and graphical works
- Automatic data collection from websites (called web scraping)

1.6.2 Other applications

- Smartphone applications
- Programs that operate on web servers or other servers on networks
- Programs that operate in coordination with electronic circuits. Raspberry Pi is a small computer that runs on Linux and Python that was developed for this purpose.

1.7 How to learn programming

1.7.1 Reasons why programming is difficult

Computer programming can be difficult for a variety of reasons. Understanding how and why it is difficult will likely help you in your journey to learn programming.

1) The concepts that compose programming languages are hard to understand

Even the natural languages that you use every day require complex grammar and diction in order to express complex ideas. Similarly, programming languages require you to employ various concepts and contrivances in order to skillfully express complex programs.

There is no need for you to try to understand all of these concepts at once. You can **gradually work your way up from the simplest ones**.

2) Unable to deal with errors

In programming, various errors (called bugs to liken them to metaphorical bugs eating away at the program) arise from mistakes and typos. It is important to go into programming with the understanding that you will frequently encounter bugs, however:

- You should understand that accurate typing in accordance with the syntax of the source code is necessary.
- Syntax errors aside, there will be times when misconceptions about the program lead to it operating in a manner that defies expectations.
- The vast majority of bugs are due to human error. It is important for you to gradually build up experience recognizing and dealing with various errors.
- Dealing with errors involves backtracking from the result of "an error has occurred" to the root problem that is causing it. You will need to come up with various hypotheses about the cause of the error that correspond to the type of error you have encountered and then investigate to determine whether or not it is actually the cause.
- There are times when the computer's response (when the computer has been rendered unable to process) does not match up with the actual mistakes in the program.

3) Unable to add the feature that you wish to implement

Even if you learn the components that make up a programming language, it can often be difficult to figure out how to combine them in a manner that produces the desired result. Just like how even if you know how to use a hammer and saw, you cannot build a house unless you know how a house is built.

It is necessary to thoroughly analyze what you wish to do in plain words and then to write the program after getting a clear understanding of the procedure. You can start by learning from examples of relatively simple applications and gradually working your way up.

4) The program becomes too complicated to understand

When a program becomes very long it gets more and more complex and it can suddenly become difficult to make sense of just what you are programming. For beginners, even a 100 line program can seem very daunting. Learners should work towards a goal of writing a 100 line program as they gradually pick up methods of coding in a way that makes longer programs much easier to navigate.

5) There are techniques for writing large programs

Some of the largest programs in the world can reach hundreds of millions of lines. Naturally, a single person cannot write these programs on his/her own, nor can he/she know all of the lines of code they contain. There are certain methods and tools for writing such huge programs. Some examples can be seen below.

- Create a thorough plan for the entire program
- Partition it into modules so that the work can be divided
- Write and test the program module by module.
- Test the entire program after putting it all together

Once you are able to write programs of around 100 lines, it is a good idea to challenge yourself to write a larger program with these techniques in mind.

1.7.2 How to learn programming

As with all subjects, it is essential for you to learn how to learn.

- How do you pick up a foreign language?
- How do you learn mathematics?

Programming is the skill of actually writing programs with your own hands, so in that sense, it is similar to other practical skills like math and foreign languages. However, programs can be tested as you write them, and they can have all sorts of interesting applications, so they should not be as intimidating as math or foreign languages.

1) Motivation: you should work with things that interest you

- Your motivation for learning is critical if you wish to be able to persist in their studies. There are many people who say vague things like “I want to learn how to program.” However, this alone is too vague to serve as a concrete goal, which makes it easy for you to lose motivation. Fixating on something more concrete, even something more difficult like “I want to make a game” is usually more effective. Choosing a certain subject or application that you are interested in makes it easier for you to preserve your motivation to learn.

2) Lots of reading and writing is fundamental to learning programming

- Type out and execute a bunch of exercises.
  - “Understand through doing” rather than “do after understanding”
  - Learn the patterns in vocabulary, symbols, and notations. It’s important to learn common patterns as a single unit.
  - Typing source code more quickly and accurately helps raise your learning efficiency.

3) Reading aloud/Read while interpreting

- Reading the symbols in the source code out loud while interpreting their meaning is an important ability that helps to facilitate **effective communication** in class.

4) Tinkering: play around with programming exercises.

- Make small changes to programs found in exercises to gain a feel for what is possible.
- Try combining multiple exercises. Doing so will help you understand what adjustments are necessary when combining programs.

5) Tracing

- Follow a program manually by hand in order to see how it will function (called tracing)

6) Become able to deal with errors

- When programming, you are constantly dealing with errors. Becoming able to do so is an important goal in and of itself.
- Actual programming requires you to be able to deal with unforeseen errors, but **intentionally writing programs that contain errors in order to see what happens** can be an effective way to gain experience.
- Read the error messages. There are many beginners who do not read the error messages that are displayed when they encounter an error. Error messages can definitely seem difficult to understand, but they are telling you where or what the error you have encountered is. Error messages for errors that you have intentionally caused should be relatively easy to understand, so please try to familiarize yourself with them.

7) Look up information

- You should strive to become able to search for information on programming via topic or method. A few methods for doing so are given below.
  - ➢ Learn how to use various libraries and advanced programming concepts.
  - ➢ Learn about tools that support programming.
  - ➢ You can look up information in books or on the internet.
  - ➢ Ask others for help with things you don’t understand. This requires the communication skills to be able to ask for help when you need it.

1.7.3 Characters used in programs

Most programming languages are designed with English as a basis, so you will need to keep this in mind when dealing with characters and character encoding. Reference the column titled "Programming & Japanese - The Never-ending Battle against Character Encoding."

- Python (and most programming languages) uses **half-width alphanumeric characters.**

Next Chapter Table of Contents

- Full-width characters are only used in strings and comments[1].
- Python distinguishes between uppercase and lowercase letters (it is “case sensitive”).
- Various symbols are used in programming.
  - It is important to learn the **appropriate names** for these symbols,
  - not just their **location on the keyboard.** This ensures you can communicate about these symbols with others.
- In addition, you will need to be able to press keys, such as the C key, while holding down the Ctrl key. This is written as Ctrl-C. The same is true for the Alt key.

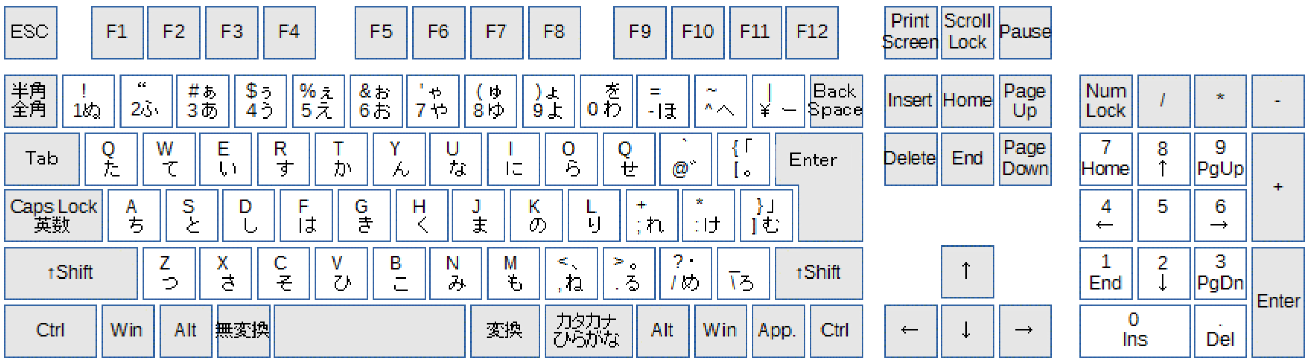

ESC F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12
Print Screen Scroll Lock Pause
半角 全角 1ぬ 2ふ 3あ 4う 5え 6お 7や 8ゆ 9よ 0わ -ほ ^へ ¥ ー Back Space
Insert Home Page Up
Num Lock / * -
Tab Q た W て E い R す T か Y ん U な I に O ら P せ @ ゛ [「 Enter
Delete End Page Down
7 Home 8 ↑ 9 PgUp +
Caps Lock 英数 A ち S と D し F は G き H く J ま K の L り ; れ : け] む
4 ← 5 6 →
↑Shift Z つ X さ C そ V ひ B こ N み M も , ね . る / め ろ ↑Shift
↑
1 End 2 ↓ 3 PgDn Enter
Ctrl Win Alt 無変換 変換 カタカナ ひらがな Alt Win App. Ctrl
← ↓ →
0 Ins . Del

Figure 1-7 JIS keyboard layout

Various symbols are arranged differently than in an English keyboard (ASCII layout).

Pronunciations and notes regarding the usage of the most common symbols can be found in the table below. This table was taken from a literature reference [1] and a portion added with the consent of the authors.

[1] While variable names in Python can use kanji, this is not the case for many programming languages, so it is safer to just avoid using it.

Table 1-1 Symbols used in programming and their pronunciations

Symbol	Pronunciation	Notes
␣	Space	Denoted as ␣ in the symbol column for ease of understanding. Programs use half-width spaces. Note that using full-width spaces in anything other than strings of Japanese text will result in an error that will be difficult to troubleshoot.
!	Exclamation mark	
”	Double quotes	Both ” and ’ can be used to surround strings in Python. Either can be used, but you must use the same type of quotes on both sides of the string.
’	Single quotes, apostrophe	
#	Pound sign, no. sign	
$	Dollar sign	
%	Percent	
&	And, ampersand	
*	Asterisk	
+	Plus	
,	Comma	
-	Minus, hyphen	
.	Period, dot	
/	Forward slash	
:	Colon	Take note of the difference between these two
;	Semicolon	
<	Less than	
>	Greater to	
=	Equals, equal sign	
?	Question mark	
@	At sign, at	
¥ \	Yen symbol Backslash	In JIS encoding, ¥ is assigned the same code as \. The Unicode (UTF-8) encoding used in Python assigns these symbols different codes, but many Windows Japanese fonts display ¥ in place of \. Mac users can simply just use a backslash.
^	Caret, free-standing circumflex	
_	Underscore, underline	In Python, you will often use underscores by combining two in a row like so: __

|	Vertical bar	
~	Tilde	
[	Square bracket (open bracket)	Python and many other programming languages make use of brackets and parentheses for various different uses. It can be easy to make typos.
]	Square bracket (close bracket)	
{	Curly bracket (open bracket)	
}	Curly bracket (close bracket)	
(	Parenthesis (open parenthesis)	
)	Parenthesis (close parenthesis)	
<=	Less than or equal to	2 characters
>=	Greater than or equal to	2 characters
!=	Not equal to	2 characters
==	Double equal sign, equal-to operator	2 characters

Exercise 1-2 Symbols used in Programming

Review the names of the symbols used in programming as well as their locations on the keyboard.

1.8 The fundamental concepts used in programming

The following are the fundamental concepts that make up programs, both in Python and in most other programming languages.

- Arithmetic, strings, logical (true or false) operators
- Variables, variable assignment, evaluating variables (using the assigned value)
- Switching how the program functions based on various conditions (branching)
- Repeating certain lines of the program
- Coding for and calling a fixed set of operations (defining and calling functions)
- Managing complex data
- Input/output (computer terminals, GUIs, files, networks)

1.9 What part of the program do you write?

Nowadays, it is extremely rare for anybody to write an entire application all by themselves. You should understand that coding generally consists of programming things that fall in between the two categories below.

- **Framework:** The GUIs used by applications on personal computers as well as the web servers that web applications run on are almost always pre-existing programs. A program like this is called a framework.

- **Library:** On the other hand, mathematical functions like sine and cosine that many people will find useful are generally used as part of a pre-existing library.

That is to say, you code the specific operations of their program within a framework while using a library that suits your programming needs.

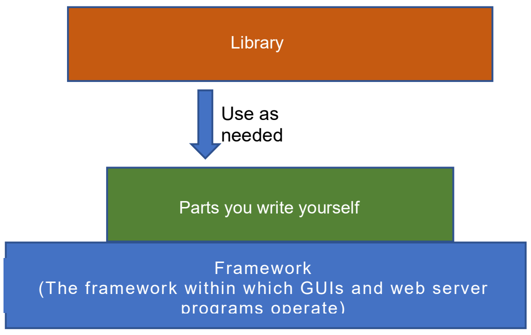

Library
Use as needed
Parts you write yourself
Framework
(The framework within which GUIs and web server programs operate)

Figure 1-8 Frameworks and libraries

References

[1] 喜多 一，岡本雅子，藤岡健史，吉川直人：写経型学習による C 言語プログラミングワークブック，共立出版（2012, in Japanese）

Next Chapter Table of Contents

2. Python: Execution Environment and How to Use It

2.1 Learning goals of this chapter

- Learn how to start Python's integrated development environment, IDLE.
- Learn how to operate the Python Shell within IDLE.
- Learn how to use the editor to edit Python programs (scripts) in IDLE.

2.2 Assumptions regarding the learning environment

This text was written with the assumption that Kyoto University's dedicated learning computers would be used. As such, it assumes Python learning will occur in the following environment.

- Operating system: Windows 10
- Python distributable: Anaconda (made with Python 3)
- Python integrated development environment: IDLE (included with Anaconda)

Learners should install Anaconda on their own PCs. In the case that you do not intend to use the numerical calculation modules introduced in Chapter 11 (NumPy, matplotlib, pandas), the original Python package will suffice.

There are various integrated development environments for Python aside from IDLE, including Jupyter Notebook and Spyder. IDLE is used as the integrated development environment for this text because its functions are limited and thus it is easier for beginners to understand (easier for instructors to teach with). In addition, it is easier to operate the turtle graphics that will be used as examples. However, you should note that its startup method and behavior is slightly different depending on whether you are using Windows or macOS. Important points regarding its use in macOS will be presented later on.

2.3 Setup

Create the folder where you will be saving the Python programs (scripts) that you write for this class. For example, you can make a folder called "Python Scripts" within your "My Documents" folder[1].

[1] If you make a folder on the NextCloud N: drive on one of the university's computers, you will be able to access it on your own PC at home.

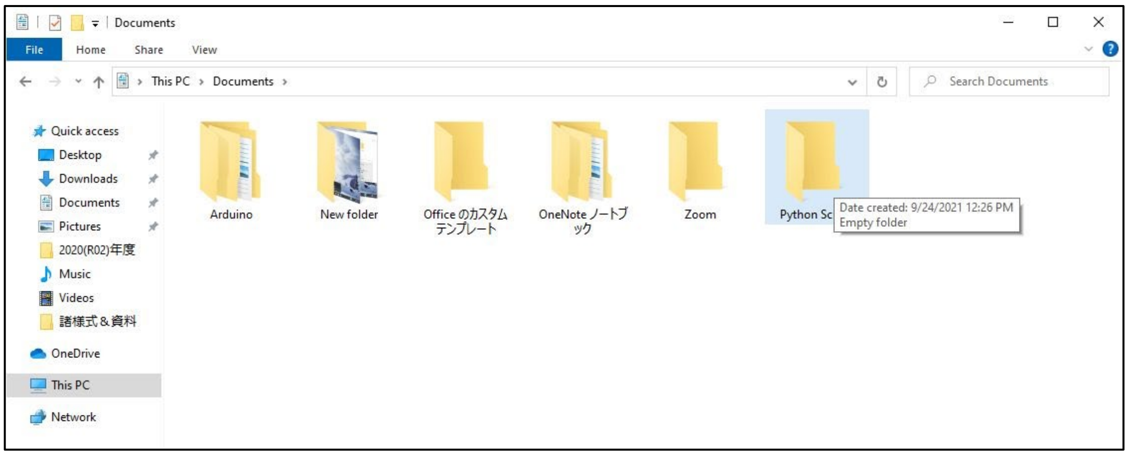
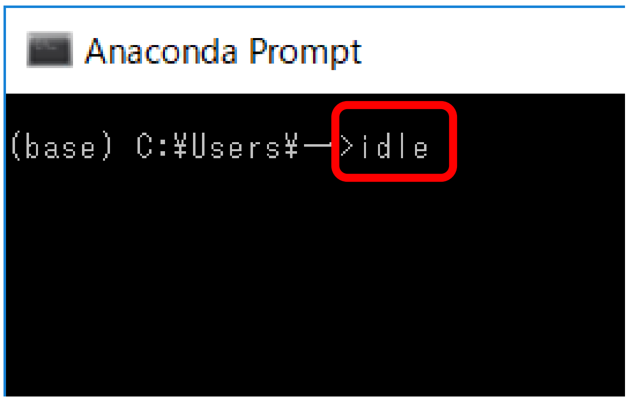


Figure 2-1 Creating a folder to hold your programs

2.4 Launching IDLE

From the start menu, select the 'Anaconda Prompt' found within the 'Anaconda3' folder and double click to start it. Once it starts, launch IDLE by typing idle (in uppercase or lowercase letters) into the window and pressing the ENTER key.



```
(base) C:¥Users¥—>idle
```

Figure 2-2 Starting IDLE from the Anaconda Prompt

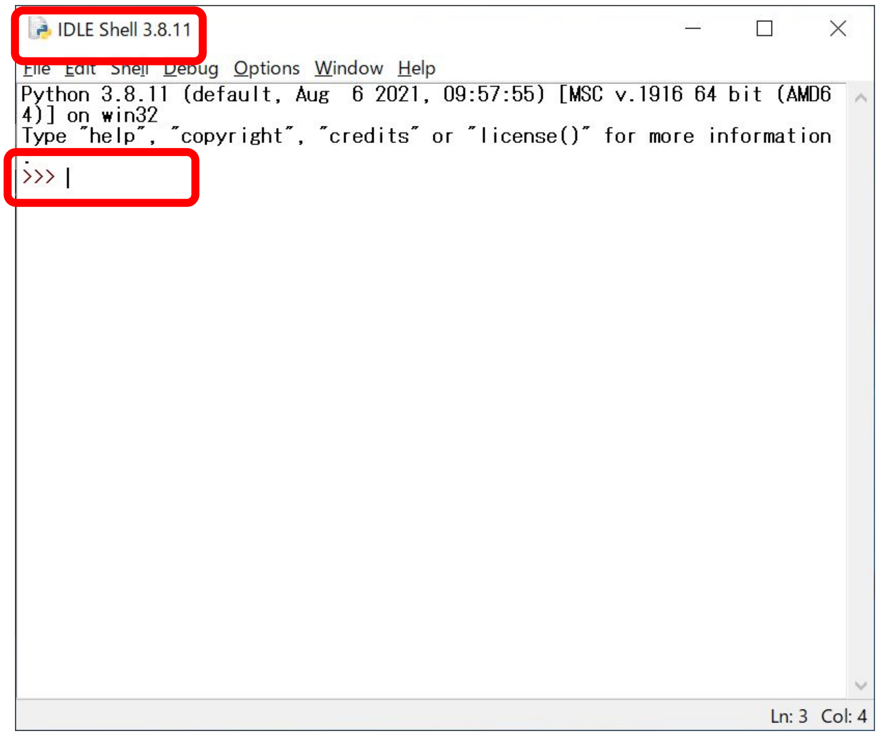
2.5 Python Shell

2.5.1 Confirming that it launched

When you launch IDLE, a Python shell like the one shown in the figure below will appear. This is an environment in which Python can be executed in an interactive manner. Please check the following two things:

- **Double check the window title.** The currently running Python version (IDLE Shell 3.8.11 in this case) is displayed in the window title. When you have multiple versions of Python installed, there are times when the incorrect version (Python 2, for example) is launched. If this happens, double check how to launch IDLE.

- **Double check the prompt.** The “>>>” within the window is a symbol (called a prompt) meant to prompt you to input commands. You can input Python commands here via the keyboard.



```
Python 3.8.11 (default, Aug  6 2021, 09:57:55) [MSC v.1916 64 bit (AMD6
4)] on win32
Type "help", "copyright", "credits" or "license()" for more information
.
>>> |
```

Figure 2-3 IDLE’s Python Shell

2.5.1 Executing Python commands

Input

```
1+2
```

after the prompt in the Python shell and press the ENTER key. Henceforth, **commands to be written into the prompt will be shown in red text.** This is a full-fledged Python program that gives the answer to 1+2. The shell should execute this and return

```
3
```

in response. **Results returned from the prompt will be written below in blue text.**

Arithmetic operations in Python can be performed as shown in the table below. Multiplication uses “*” and division uses “/.” Just like in mathematics, multiplication and division will be prioritized over addition and subtraction. In addition, you can use () in order to specify the desired computation sequence.

It is worth noting that in Python 3, “/” yields a float variable even when used on two integers. When an integer is needed, you should use “//.” Also, there are many cases in which you would need to use the remainder of a division operation in their program. The “%” operator can be used to obtain this remainder.

Table 2-1 Arithmetic operations in Python

Operator	Operation	Notes
+	Addition	
-	Subtraction	
*	Multiplication	
/	Division	Returns a float variable in Python
//	Integer division	
%	Remainder	Gives the remainder of a division operation
**	Exponent	Note that this is 2 characters.
( )	Prioritized operations	Other types of brackets cannot be used for this purpose.

Exercise 2-1 Reviewing arithmetic operations
Practice arithmetic operations in the Python Shell.

Input each of the next two lines (one at a time). Reference the figure to the right.

```
a = 1 + 2
a
```

The first line is a command that assigns the value of the equation “1+2” on the right-hand side of the “=” to the variable “a” on the left-hand side. The shell will not display anything after this command and will simply request the next input.

The second line confirms the value of the variable a.

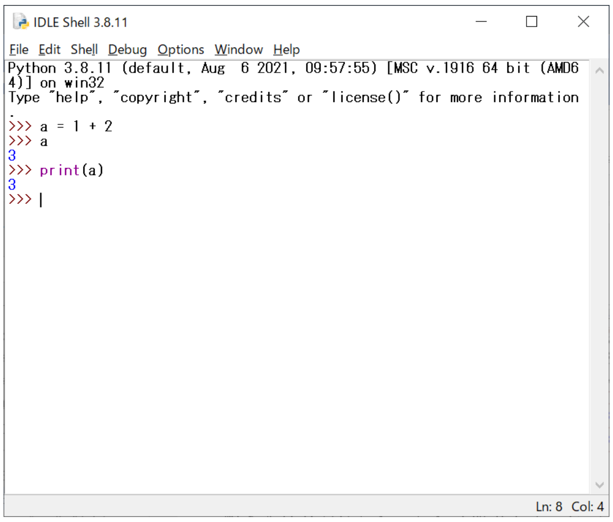
```
3
```

should be what the shell displays. Next, try inputting

```
print(a)
```

into the prompt. print() is a function that outputs the expression within the () onto the shell as characters. As you would expect,

```
3
```



```
Python 3.8.11 (default, Aug  6 2021, 09:57:55) [MSC v.1916 64 bit (AMD6
4)] on win32
Type "help", "copyright", "credits" or "license()" for more information
.
>>> a = 1 + 2
>>> a
3
>>> print(a)
3
>>> |
```

Figure 2-4 Operating the Python Shell

is displayed.

Simply inputting a variable's name into the Python shell will display its value. In the programs that you will code later on, which will be scripts in which multiple lines are executed all at once, will require you to explicitly use the print() function.

2.6 Writing and running scripts

Next, we will learn how to write a multiline Python script and execute it all at once. To do this, you will use the IDLE Editor to edit the commands.

2.6.1 Creating a new file

To create a new program, select "New File" from the "File" menu in the shell window. This should launch the IDLE Editor.

2.6.2 Verifying the IDLE editor

The IDLE Editor and the Python shell look very similar. Use the three points detailed below in order to tell them apart.

- The title of the window will be the name of the file that you're editing. If you have selected "New File," the window title will be "Untitled."
- The menus in this window are different than those in the Python shell. Make sure that there is a menu called "run."
- The interior of the window is blank. There is no ">>>" prompt like in the shell.
- The line and column of the cursor should be displayed in the bottom right if you are using Windows.

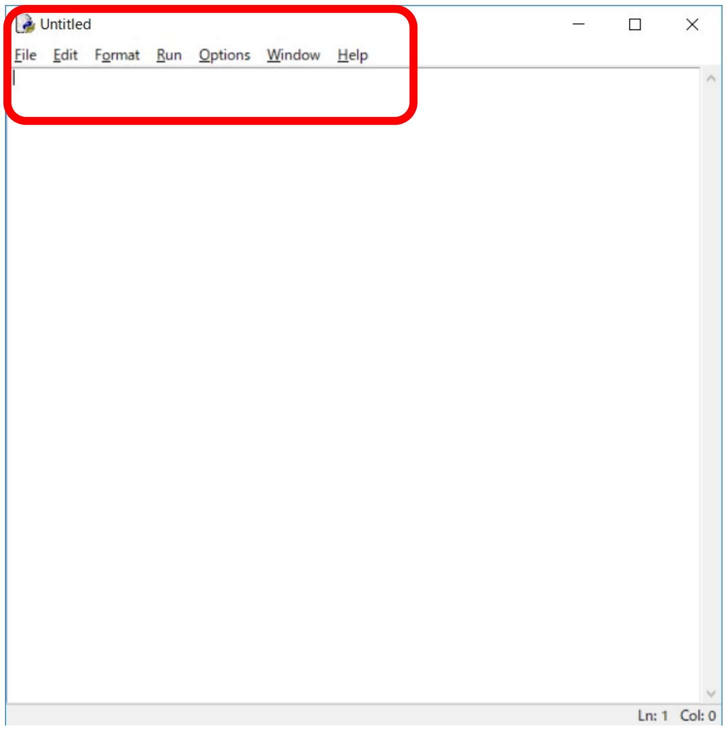


Figure 2-5 Note the differences between IDLE Editor and the Python shell

Exercise 2-2 Differences between the Python shell and IDLE Editor

Please review the differences between the Python shell and IDLE Editor.

2.6.3 Coding, saving, and running a Python program

It's only a two line program, but please input the text found in the yellow section of the table below into the IDLE Editor.

Program 2-1 (p2-1.py)

Row	Source code	Notes
1	a␣=␣1␣+␣2	Assigns the result of the equation "1+2" on the right-hand side to the variable a.
2	print(a)	Outputs the value of the variable a to the screen.

Confirm that there are no typos and then select "Run" and then "Run Module." Run means to execute the program, and module refers to the Python program that is currently being edited.

As this is a new program, you will need to save it. Save it as p2-1.py (.py is the filename extension for Python programs) in the folder that you created for your Python programs. After doing so, it will be run in the Python shell and the results will be displayed.

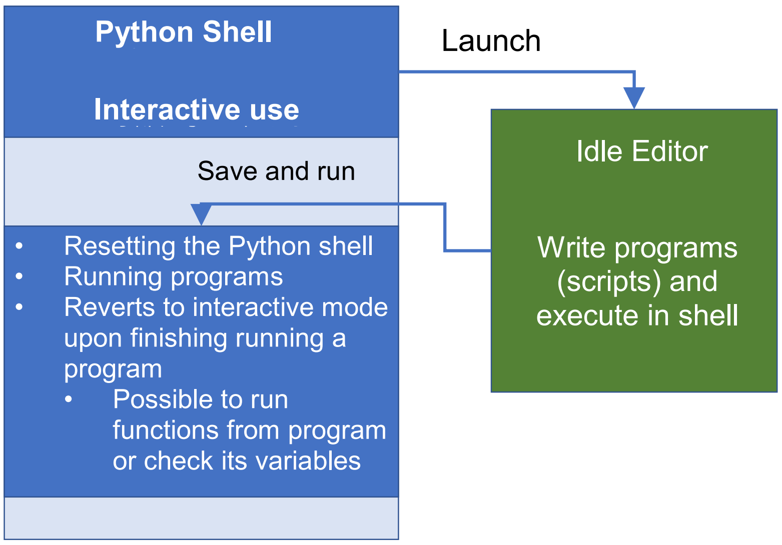

Python Shell
Interactive use
Launch
Idle Editor
Write programs (scripts) and execute in shell
Save and run
Resetting the Python shell
Running programs
Reverts to interactive mode upon finishing running a program
Possible to run functions from program or check its variables

Figure 2-6 Interactions between IDLE shell and Editor

When you tell the Idle Editor to run something, the program is saved in the file, the Python shell is reset, and the program is run[1]. When the program has finished running, the Python shell will revert to interactive mode where it can receive input from the keyboard. In this mode, you can check the values of the program's variables and call its functions.

Exercise 2-3 Confirmation of thr result after execution of Program 2-1

Once p2-1.py has finished running, run the following command to check the value of a.

```
print(a)
```

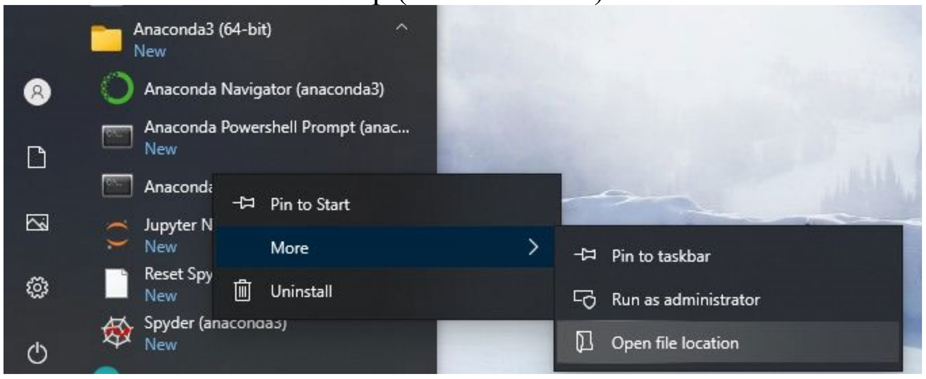
2.7 Setting the working directory with the Anaconda Prompt

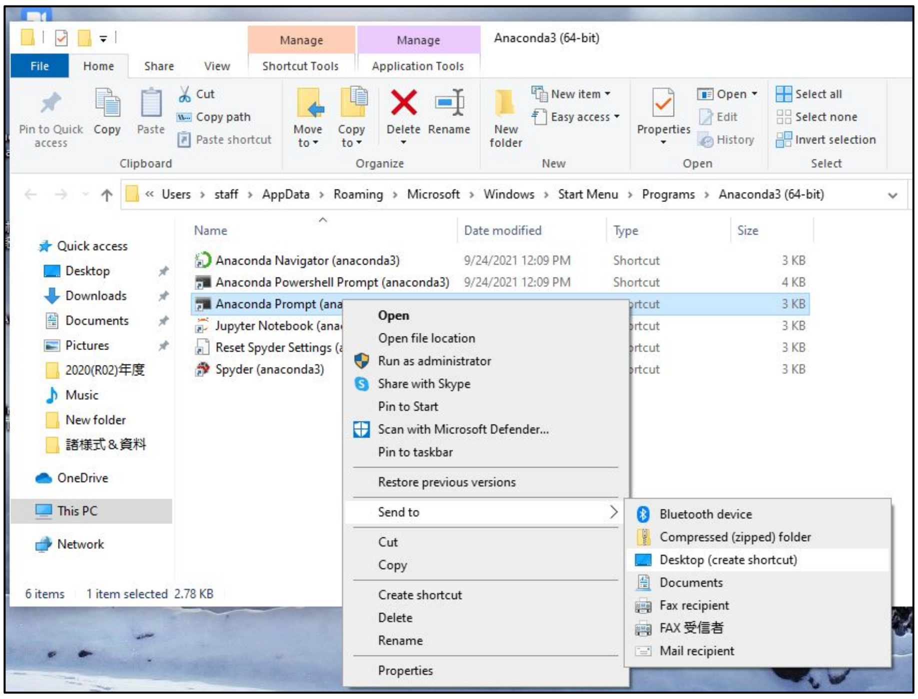
We created a folder to store our Python programs, so let's set it so that this folder is opened by default. You are not able to specify a working directory in IDLE, so you will just set one in the Anaconda Prompt that runs IDLE. Set the working directory using the following procedure.

1. Create a shortcut for the Anaconda Prompt on the desktop
    1) Right click on the Anaconda Prompt in the Start Menu
    2) Select "Other" => "Open file location"
    3) Right click on the Anaconda Prompt icon in the explorer

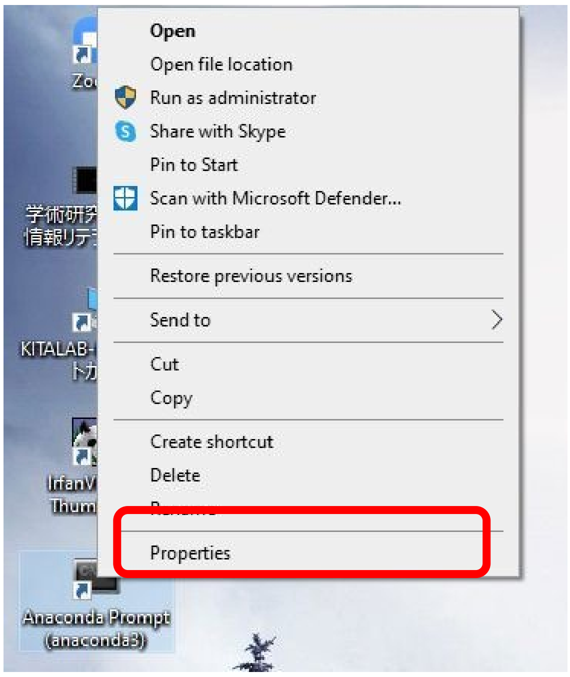
[1]The reason that the shell needs to be reset is to eliminate the potential effects of any lingering variables from when the shell was in interactive mode.

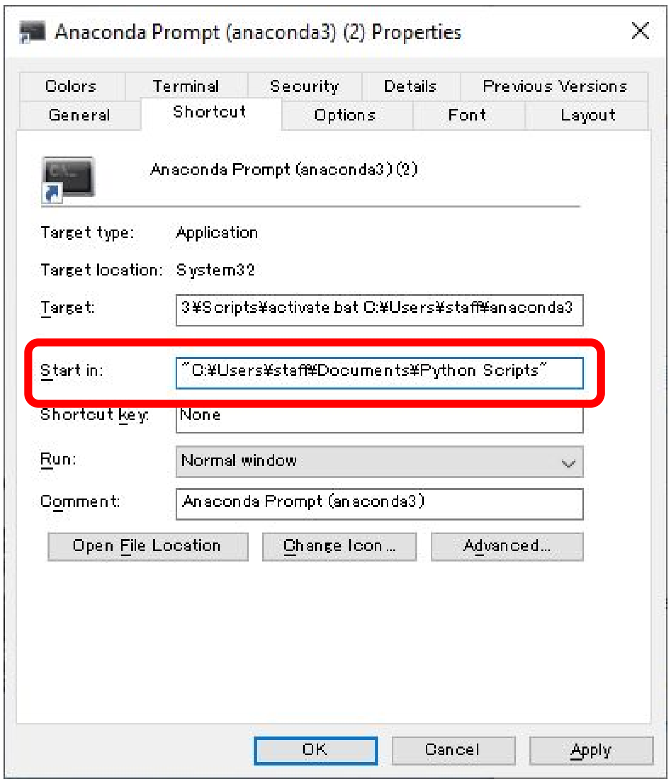
4) Select “Send to” => “Desktop (create shortcut)”




Users › staff › AppData › Roaming › Microsoft › Windows › Start Menu › Programs › Anaconda3 (64-bit)
Open
Open file location
Run as administrator
Share with Skype
Pin to Start
Scan with Microsoft Defender...
Pin to taskbar
Restore previous versions
Send to
Cut
Copy
Create shortcut
Delete
Rename
Properties
Bluetooth device
Compressed (zipped) folder
Desktop (create shortcut)
Documents
Fax recipient
FAX 受信者
Mail recipient

2. Set a working directory using the Anaconda Prompt shortcut on the desktop.
   1) Right click the Anaconda Prompt icon on the desktop
   2) Select “Properties”
   3) Add the folder location of your Python script folder to “Start in”
   4) Click “Ok”





Anaconda Prompt (anaconda3) (2)

Target type: Application

Target location: System32

Target: 3¥Scripts¥activate.bat C:¥Users¥staff¥anaconda3

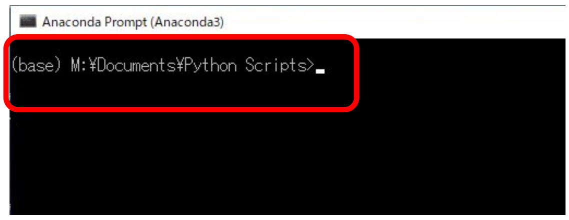
Start in: "C:¥Users¥staff¥Documents¥Python Scripts"

Shortcut key: None

Run: Normal window

Comment: Anaconda Prompt (anaconda3)

From now on, when you double click this desktop icon, the Anaconda Prompt and idle will work from the specified folder.



```
(base) M:¥Documents¥Python Scripts>_
```

2.8 IDLE keyboard shortcuts

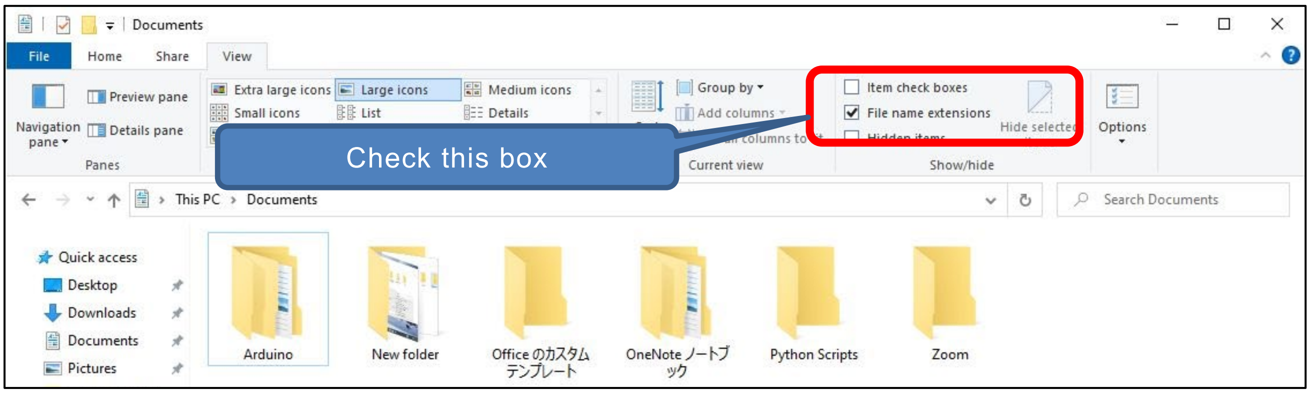
IDLE's Python shell and IDLE Editor are very simple, however they come with some several useful keyboard shortcuts. These can be read in IDLE's online manual, but the most commonly used ones are listed in 16 Useful notes on Python and IDLE.

2.9 Showing file extensions

Filetypes can be determined by looking at the filename extension (a name that follows the period after the filename; ".py" for Python files). Programming requires you to use various types of files, but the Windows file explorer hides filename extensions by default.

Checking the box in the View tab of the ribbon at the top of the file explorer (shown in the figure below) will tell the explorer to display filename extensions.

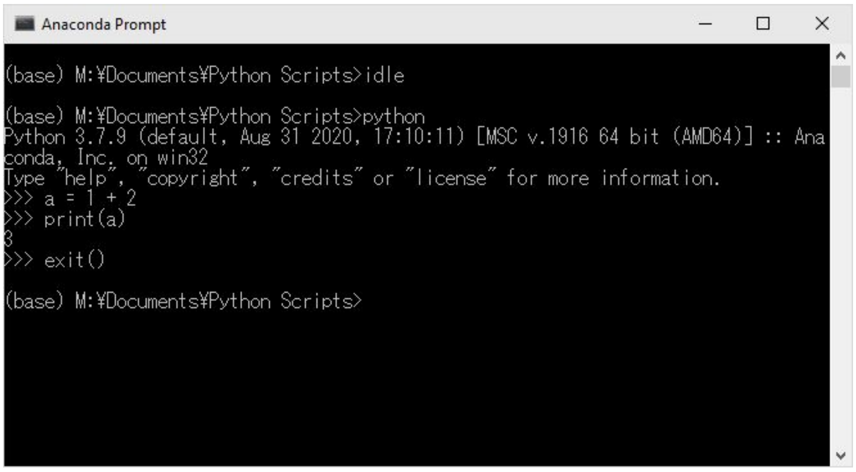
Note: Being able to see filename extensions also enables you to change them. Please note that accidentally changing a filename extension can cause files to no longer be associated with certain applications.



2.10 Executing Python commands

Python programs can also be run directly via the Anaconda Prompt. Please go through the following steps.

1. Close any sessions of IDLE and Anaconda Prompt that are currently open.
2. Specify the working directory as described in the previous section.
3. Launch the Anaconda Prompt from the shortcut on the Desktop.
4. Input “cd” and press the ENTER key to confirm that the working directory has been set properly.
5. Input “dir” and press the ENTER key to see a list of the files that are saved in this folder. Check to make sure that p2-1.py is listed.
6. Input “python” and confirm that the Python shell is launched in the Anaconda Prompt. For the time being, you are merely confirming that it launches, so either input “exit()” or input “C” while holding the CTRL key to close the shell.
7. Specifying the name of a Python program (script) by inputting “python p2-1.py” and running the Python command will execute the program. Please confirm that this command runs as intended.
8. Specifying the -i option by inputting “python -i p2-1.py” causes it to revert to interactive mode after running the program (as it does when running the program in IDLE)

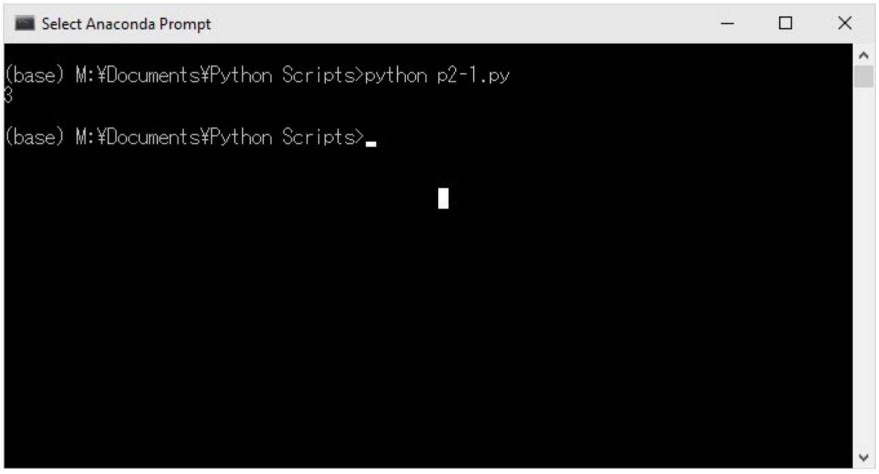


```
(base) M:¥Documents¥Python Scripts>idle

(base) M:¥Documents¥Python Scripts>python
Python 3.7.9 (default, Aug 31 2020, 17:10:11) [MSC v.1916 64 bit (AMD64)] :: Ana
conda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 1 + 2
>>> print(a)
3
>>> exit()

(base) M:¥Documents¥Python Scripts>
```

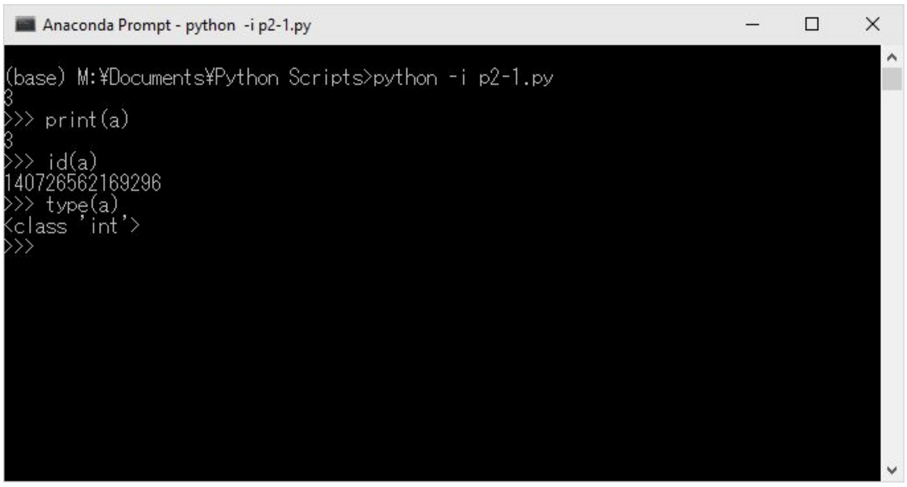
Figure 2-7 Running Python in interactive mode



```
(base) M:¥Documents¥Python Scripts>python p2-1.py
3

(base) M:¥Documents¥Python Scripts>
```

Figure 2-8 Running Python via a specified script



```
(base) M:¥Documents¥Python Scripts>python -i p2-1.py
3
>>> print(a)
3
>>> id(a)
140726562169296
>>> type(a)
<class 'int'>
>>>
```

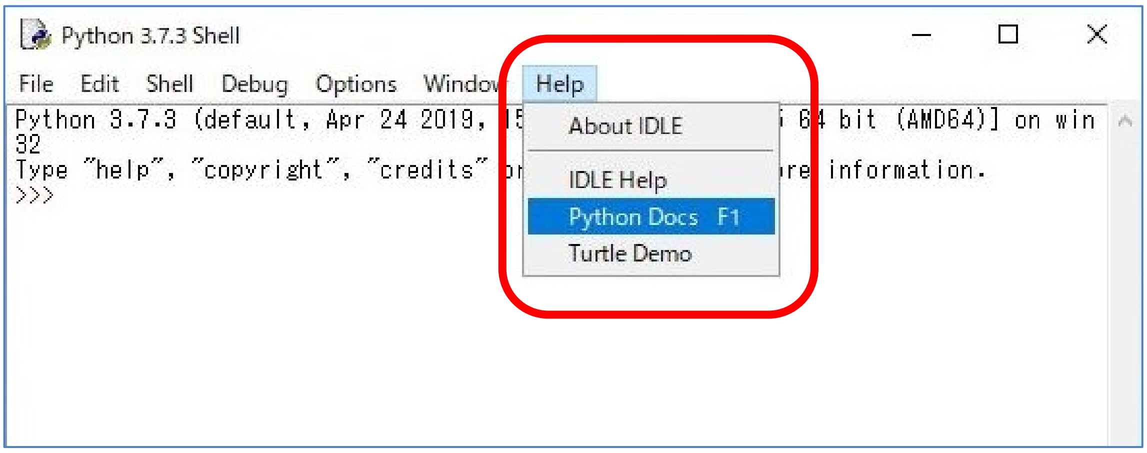
Figure 2-9 Using the -i option to stay in interactive mode once the program is finished running

2.11 Creating a good environment for learning Python

- Python language processor: install Anaconda on your computer
- Look over the Python Language Reference
- Keep a book on Python (one that suits your learning style) on hand
- English dictionary (data confirmation, function and variable names)
- Notebook and writing utensil (PC notepad works too): write down things that spring to mind

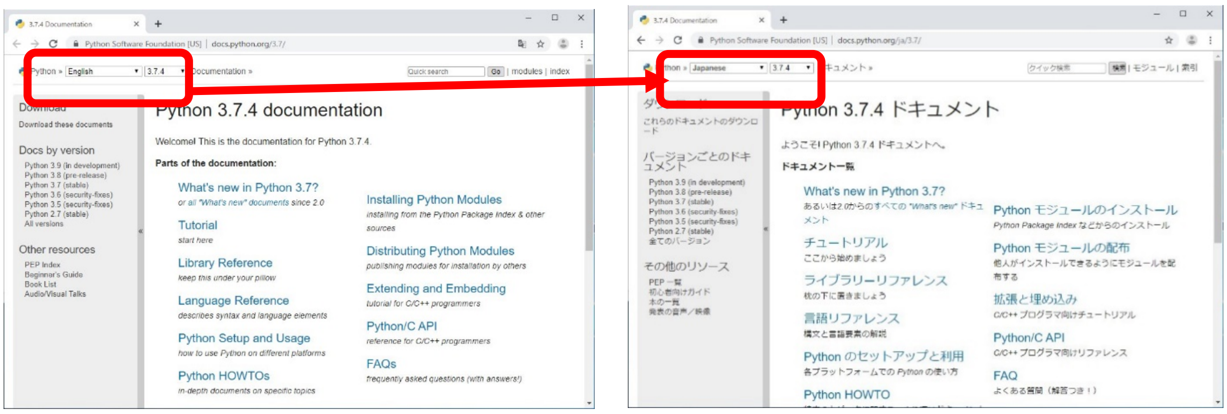
Exercise 2-4 Prepare your own Python learning environment and report it.

Exercise 2-5 Redo today's exercise in your personal learning environment.



```
Python 3.7.3 (default, Apr 24 2019, 15                    4 bit (AMD64)] on win
32
Type "help", "copyright", "credits" or                  re information.
>>>
```

Figure 2-10 Launching the online manual from IDLE


3.7.4 Documentation
Python Software Foundation [US] | docs.python.org/3.7/
Python » English 3.7.4 Documentation »
Quick search Go | modules | index
3.7.4 Documentation
Python Software Foundation [US] | docs.python.org/ja/3.7/
Python » Japanese 3.7.4 ドキュメント »
クイック検索 検索 | モジュール | 索引

Download
Download these documents
Docs by version
Python 3.9 (in development)
Python 3.8 (pre-release)
Python 3.7 (stable)
Python 3.6 (security-fixes)
Python 3.5 (security-fixes)
Python 2.7 (stable)
All versions
Other resources
PEP Index
Beginner's Guide
Book List
Audio/Visual Talks

Python 3.7.4 documentation

Welcome! This is the documentation for Python 3.7.4.

Parts of the documentation:

What's new in Python 3.7?
or all "What's new" documents since 2.0
Tutorial
start here
Library Reference
keep this under your pillow
Language Reference
describes syntax and language elements
Python Setup and Usage
how to use Python on different platforms
Python HOWTOs
in-depth documents on specific topics
Installing Python Modules
installing from the Python Package Index & other sources
Distributing Python Modules
publishing modules for installation by others
Extending and Embedding
tutorial for C/C++ programmers
Python/C API
reference for C/C++ programmers
FAQs
frequently asked questions (with answers!)

ダウンロード
これらのドキュメントのダウンロード
バージョンごとのドキュメント
Python 3.9 (in development)
Python 3.8 (pre-release)
Python 3.7 (stable)
Python 3.6 (security-fixes)
Python 3.5 (security-fixes)
Python 2.7 (stable)
全てのバージョン
その他のリソース
PEP 一覧
初心者向けガイド
本の一覧
発表の音声／映像

Python 3.7.4 ドキュメント

ようこそ! Python 3.7.4 ドキュメントへ。

ドキュメント一覧

What's new in Python 3.7?
あるいは2.0からのすべての "What's new" ドキュメント
チュートリアル
ここから始めましょう
ライブラリーリファレンス
枕の下に置きましょう
言語リファレンス
構文と言語要素の解説
Python のセットアップと利用
各プラットフォームでの Python の使い方
Python HOWTO
Python モジュールのインストール
Python Package Index などからのインストール
Python モジュールの配布
他人がインストールできるようにモジュールを配布する
拡張と埋め込み
C/C++ プログラマ向けチュートリアル
Python/C API
C/C++ プログラマ向けリファレンス
FAQ
よくある質問（解答つき！）

Figure 2-11 Python online manual (the right side is after selecting Japanese)

2.12 For Mac users

This text explains how to use Python in Windows. There are a few differences to consider when using it with a Mac. Please reference the list below.

2.12.1 Installing Python and launching IDLE on Mac

1) Installing Anaconda

Press the download button on the site below or scroll down the page and download the Mac OS installer from the screen that is displayed (if you don't know which to choose, select the 64-Bit Graphical Installer).

https://www.anaconda.com/products/individual/

Run the package file that you downloaded to install it (if you don't understand the choices in the installer, select "install for me only").
If you use a Mac equipped with the new M1 CPUs, you will be asked whether to install Rosetta, which is software that enables you to run programs designed for the older CPUs. Therefore, proceed with the installation.

Also, it is worth noting that when Anaconda is installed, the terminal environment is changed into a Conda environment more suitable for running Python (will display "(base)" at the start of every line). This is all well and good if you only plan to use the terminal for programming exercises, but it may be problematic when trying to run other software. To return the terminal to the default environment, input "conda deactivate." In order to return it to the Conda environment after deactivating it, simply input "conda activate."

If the Conda environment does not seemed to be installed ("(base)" is not displayed), run the line below in the terminal, close it, and then reopen it. The "zsh" at the end is the name of the shell that runs in the terminal. In the event that a shell other than zsh is being used, simply use the name of that shell.

```
/opt/anaconda3/bin/conda init zsh
```

2) Launching IDLE

The terminal is the equivalent to the Windows command prompt. IDLE is run from the terminal.

You can launch it by selecting "Applications" => "Utilities" => "Terminal" in Finder.

Provided that Anaconda is installed, IDLE for Python 3 can be launched by inputting "idle3" using the keyboard and pressing enter (please note that this is slightly different than the previous section in which you needed to enter "idle" into the command prompt in Windows).

3) Navigating IDLE

Sometimes, when launching the IDLE Editor, you will be unable to navigate the menus by clicking. If this happens, just click on another window or the desktop background and then click back to the IDLE window (it will be titled "Python3.8"). This should fix the problem and enable you to use the menus.

4) Japanese input in IDLE

Please note that using Japanese input in IDLE can cause it to become slow and unstable. At present (9/29/2020), using Google Japanese input can lead to problems inputting characters. If this happens, it is recommended to use Mac's standard Japanese input.

For comments and other parts that don't affect the program's operation, one method is to simply avoid using Japanese altogether.

5) Inputting backslashes in IDLE

In this textbook, there are sections in Chapters 11 and 12 in which yen symbols "¥" are input into the programs. Mac users should input backslashes "\"

To input the backslash symbol "\" on Mac, hold down the bottom left "option" key and press the "¥" key.

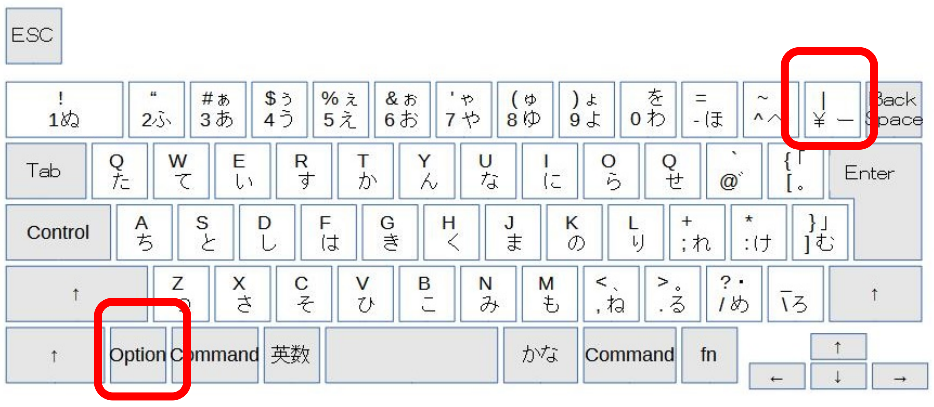


Figure 2-12 Inputting backslash on Mac

Press the ¥ key while holding the "option" key.

6) Closing IDLE

Once idle is closed, you can close the terminal window.

2.12.2 Problems with Tkinter on Mac.

Aside from the above, there are some things that rely on the operating system in which Mac users will see some differences from Windows users. Some examples include the GUI environment Tkinter, which uses a package called Tcl/Tk, and the handling of Japanese fonts in the graphing module matplotlib.

References

Many books about Python have been published in recent years, so it is likely difficult to decide which to buy. As such, we will offer a few recommendations. References [2]-[6] are introductions to Python. While it may be difficult to discern from the title, Reference [7] includes the experiences of the author, who learned Python via self-study and became a programmer. Reference [8] explains subjects that are not often included in introductory texts in the field of programming. References [9] and [10] are geared towards practical applications. References [11] and [12] are texts that discuss the numerical computing libraries, NumPy, matplotlib, and pandas, that will be touched on in this text. For books other than these ones, please confirm that they deal with Python Version 3 before using them.

[2] Bill Lubanovic: Introducing Python: Modern Computing in Simple Package, Oreilly & Associates Inc, 2nd Ed (2019)

[3] 柴田淳：みんなの Python 第 4 版，SB クリエイティブ (2017, in Japanese)

[4] 大津真:基礎 Python，インプレス (2016, in Japanese))

[5] 松浦健一郎，司ゆき：はじめての Python エンジニア入門編，秀和システム (2019, in Japanese))

[6] 大澤文孝：いちばんやさしい Python 入門教室，ソーテック社 (2017, in Japanese))

[7] Cory Althoff: The Self-Taught Programmer: The Definitive Guide to Programming Professionally, Self-Taught Media (2017)

[8] 増井敏克：基礎からのプログラミングリテラシー，技術評論社 (2019, in Japanese))

[9] 日経ソフトウェア編：いろいろ作りながら学ぶ！Python 入門，日経 BP(2019, in Japanese))

[10] Al Sweigart: Automate the Boring Stuff with Python, 2nd Edition: Practical Programming for Total Beginners, No Starch Press, 2nd Ed. (2019)

[11] Wes McKinney: Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython, O'Reilly Media, 2nd Ed. (2017)

[12] Jake VanderPlas: Python Data Science Handbook: Essential Tools for Working with Data, O'Reilly Media, (2016)

3. Assigning and Operating on Variables

3.1 Learning goals of this chapter

- Understand the flow of execution and the flow of information in Python. Learn about sequential execution.
- Learn about naming, assigning, and evaluating variables in Python.
- Learn about the basic data types in Python.
- Learn about the type() function which looks up data (object) types, and the id() function which looks up the location of an object.

3.2 Flow of execution and information in programs

3.2.1 Sequential execution

In the example from the previous chapter

```
a = 1 + 2
print(a)
```

the program is executed one line at a time starting from the top. This is called **"sequential execution,"** and it is a foundation on which programs are built. In addition, programs can:

- Diverge such that certain parts are only executed if certain conditions are met
- Repeat certain processes
- Call functions, effectively delegating the process to the definition of the function

These subjects will be tackled in later chapters. The source code of Python programs is written in accordance with what is known as "the flow of execution."

Exercise 3-1 Programs and sheet music
Computer programs share many similarities with sheet music. A fundamental characteristic of sheet music is that the notes are played in order from the beginning (although they are read from left to right which differs from top to bottom in programs). Also, note that there is notation that enables you to change how certain parts are played or repeat certain parts.

稍々荘重に
mf
f
Sound
水梨弥久 作詞
下総皖一 作曲
(一) コ コ ノ ヘ ニ ハ ナ ゾ ニ ホ ヘ ル セ ン ネ ン ノ ミ
(二) み ど り ふ く く す の は か ぜ に と き の か ね つ

Sample sheet music: Kyoto University school anthem (taken from the link below)

https://www.kyoto-u.ac.jp/ja/about/operation/symbol/song-a.html

3.2.2 Flow of information through variables

Meanwhile, in programs, information is processed at each step **in the form of numbers or strings that have been assigned to variables.** As such, when compared to the flow of execution, **the flow of information can be quite difficult to follow, despite occurring through the assigning and checking of the same variables.** For example, in the code above, the value of the variable a that was set in the first line is used by the print function in line 2.

How about in the following example?

```
a = 1 + 2

a = 3 + 4

print(a)
```

In this program, the value assigned to 'a' in line 1 is immediately overwritten in line 2. As such, you can immediately see that this assignment is meaningless for print(a) in the third line by simply following what happens to 'a'.

3.3 Variable naming

3.3.1 Programs also use variable names with multiple characters

The variable name used in the previous example was 'a.' In mathematics, letters of the alphabet (and even Greek letters) are often used. However, programming languages, which handle various types of data, can use longer variable names. For example,

```
a

x

x2

root

square_root
```

and so on.

3.3.2 Variable naming rules

Please learn the following rules.

- You can use only uppercase letters, lowercase letters, numbers, and underscores.
- Uppercase and lowercase letters are treated as different characters.
- A number cannot be used as the first character in a name.
- Reserved keywords used in Python syntax (words such as "if;" reserved keywords will be displayed in red when using the IDLE Editor) cannot be used.

Variable names can use Japanese (kanji, etc.), however this is not common practice.

3.3.3 Use variable names that are easy to understand

1) Taking mathematics as an example

Using appropriate names facilitates smoother thinking and communication. For example, in mathematics, if you write a linear function as

$y = ax + b$

you can immediately understand that y is a linear function of x, the slope is a, and the intercept is b. This is because it is the standard convention to use x and y as variables, to have y be a function of x, and to use a and b as parameters.

However,

$b = xa + y$

is the same equation that merely switches x and y with a and b. Suddenly it becomes much more difficult to understand.

2) How to name variables in Python programs

In programs, **good variable naming can go a long way towards making a program easier to understand**. Please try to keep the following things in mind.

- Try to choose variable names that represent what the variable does within the program.[1]
- Variables with very short names (like one-character names) should only be used when they are only needed to have a very small scope to bring about their desired effect. In particular, l, o, and

[1] We don't often encounter the concept of "naming" in everyday life; naming kids or pets are generally the only times we name anything. However, when using computers, the act of naming things is critically important when it comes to filenames, folder names, and the like. Naming is one of the most common concepts within programming. It would be best to go in understanding that good naming sense is needed when programming.

O (lowercase L, lowercase O, and uppercase O) can be confused for the numbers 1 and 0, so it is best to avoid using them.

- In general, use lowercase letters instead of uppercase letters. Uppercase letters are generally used for constants that do not change in value.
- Variable names with multiple words should use an underscore (_) as spaces. For example, “street_name”[1]
- Use English when possible. There are times where programs grow much larger than initially expected and come to be used by a large number of people. Often times this can mean people all around the world will be using the program, so it is best to using English naming from the beginning. [2]

Even beyond variable names, no matter how you write your programs, making them easy to understand is of critical importance. PEP8[13] is the recommended coding guideline forPython programs.

Exercise 3-2 Practice using various variable names
Run the program shown in Program 2-1 via the shell and practice changing the variable names to various things.

- Please note that you need to change the variable in both the first and second lines to the same thing.
- Try using a variable name of multiple words connected by an underscore.
- Also, see what error messages result from using variable names with reserved keywords or names starting with a number.

3.4 Variable assignment and evaluation

Run the following program using the Python shell.

```
a = 1
print(a)
a = a + 1
print(a)
```

The variable 'a' is assigned the value of 1 in the first line.

In the third line, 'a' appears on both the left and right hand sides of the equation, so be careful when

[1] Alternatively, it is common to merely capitalize the first letter in each word after the first. In this case, the example would become “StreetName.”

[2] There was a situation in which a FORTRAN program written by a German professor for his research was rewritten in C. Older FORTRAN specifications placed limits on variable name length, so the variables were named using abbreviated German which was completely unintelligible. The professor’s program had thorough English comments accompanying it, so it was still able to fulfill its purpose.

reading it. In Python, this program does the following:

1. First, the expression on the right-hand side (a + 1) of the assignment operator (=) is calculated.
   - The variable 'a' is already assigned the value of 1, so the right-hand side will use “the result of evaluating the value of a,” which is one. This gives us 1 + 1, yielding 2 as the result of the calculation.
2. Next, this result is assigned (overwritten) to 'a,' the variable on the left-hand side.

Variables can be thought of as boxes with names.

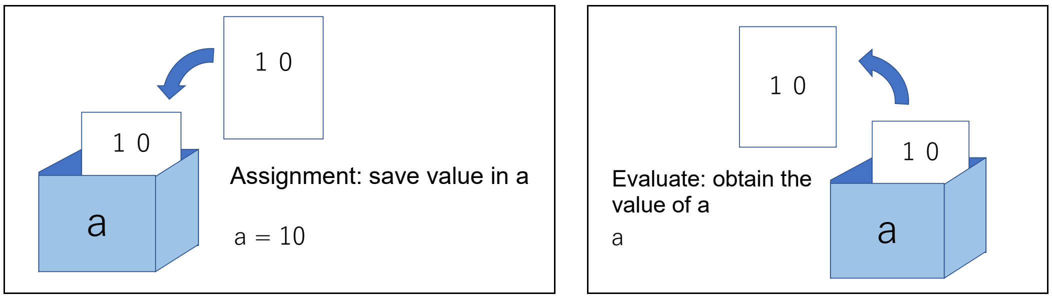


Figure 3-1 Visual representation of variable assignment and evaluation

Exercise 3-3 Explaining the behavior of variables
Below is a program that calculates the price of a ¥1000 product at 15% off.

- This program contains a single error which causes it to yield an error message when run. Please locate and explain the error.
- After correcting the error, explain how the program behaves.

```
kakaku = 1000
nebikiritsu= 15
kakaku = Kakaku*(100-nebikiritsu)/100
print(kakaku)
```

3.5 Assignment operators

In programs, you frequently need to add or subtract a fixed number (for example, 1) to a variable. You can make use of the operators below (beyond just the assignment operator “=”) in order to perform such operations more conveniently.

Table 3-1 Python assignment operators

Operator	Example	Meaning
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a*b
/=	a /= b	a = a/b

Note that the "++" and "--" operators commonly used in the C language are not present in Python.

3.6 Data types used in Python

The previous examples dealt with integers. Python programs can make use of many other data types (seen in the table below), including floating-point numbers, which contain numbers after a decimal point, strings, and Booleans (True and False). One feature of Python is that integers are not limited to a certain number of digits (although at some point they are limited by computer memory and calculation speed). For example,

```
2**200
```

evaluates to

```
1606938044258990275541962092341162602522202993782792835301376
```

As for numerical data types, complex numbers can also be used.

Table 3-2 Data types used in Python

Type Function used to convert	Explanation	Constant (literal) notation example	Notes
Integer int()		12345	No digit limit in Python
Floating-Point Numbers float()	A number that has values after the decimal point	1.0 2.99792458E8	Has a size limit (of significant digits and the range that can be expressed). E8 means x10^8.

String str()	A sequence of characters	'aaa' "日本語"	Strings are surrounded by single or double quotes
Boolean bool()[1]	Used to evaluate conditions	True False	The first letters of constants are uppercase.

It is worth noting that in Python, as with most programming languages, arithmetic using floating-point numbers is carried out in binary. We often express fractions using decimals, but just like how 1/3 cannot be perfectly expressed as a decimal, 1/10 cannot be perfectly expressed using floating-point numbers. For a more detailed explanation, please refer column chapter "What Does Float Mean?"

Exercise 3-4 Confirmaton of Data Type

Run the following in the Python shell.

```
a = 1
b = 1/2
c = "ABC"
print(a)
print(b)
print(c)
print(type(a))
print(type(b))
print(type(c))
```

Although data (generally referred to as "objects") has various "types" in Python, **variables can be assigned values regardless of the type of data they already contain.**

The type() function can be used to find out what type of object is currently assigned to a variable.

[1] Named after George Boole, the man who pioneered the theory behind the algebra of logical operators.

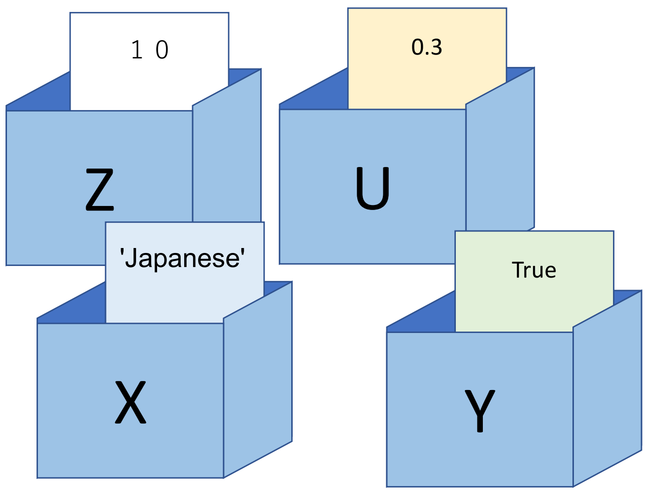


Figure 3-2 In Python, a variable can have any type

3.7 A more accurate understanding of Python variables

In reality, variables in Python do not directly hold data (objects) themselves, rather they contain information (a reference) that points to the whereabouts of the object. You do not need to worry about this very much right now, however it will become important down the line when learning about how complex data like lists are handled.

The information that points to data locations that are held by a variable can be looked up using the id() function.

```
a = 1
b = 2
print(id(a), id(b))
```

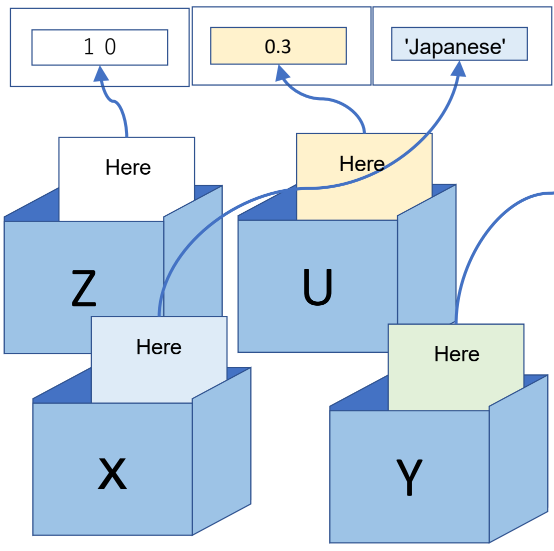


Figure 3-3 Python variables contain information on data (object) location

3.8 Exercise: Find the square root

As a good exercise that deals with both sequential execution and handling variables, let's find **an approximation of the square root** of a given number. While the method of calculating it shown below is rather simple, it uses division which leads to a large number of digits. As such, this method can be quite bothersome to calculate by hand, but using computers makes it easily achievable.

3.8.1 Calculation procedure

In order to solve for the square root $\sqrt{2}$ of a given number, follow the procedure below.

- Let the approximation of the square root be r. It can be initialized to the number whose square root you want to find (2 in this example), or it can even be set to 1. In this case, let $\mathrm{r} = 2$.
- As another approximation of the square root, divide the number whose square root you wish to find by the approximation r. In this example, that would give $2/\mathrm{r} = 2/2 = 1$. If the approximation r is indeed the square root being solved for, 2/r will become the square root as well.
- The new approximation $\mathrm{r}^{\mathrm{NEW}}$ is the midpoint (average) between these two values. In this example, that would be:

$$\mathrm{r}^{\mathrm{NEW}} = \frac{\mathrm{r} + \frac{2}{\mathrm{r}}}{2} = \frac{2 + \frac{2}{2}}{2} = \frac{2 + 1}{2} = 1.5$$

- At this point, simply repeat the process after setting the value of r to $\mathrm{r}^{\mathrm{NEW}}$.

$$r^{NEW} = \frac{1.5 + \frac{2}{1.5}}{2} = \frac{1.5 + 1.33333}{2} = 1.41666$$

As you can see, the number is indeed approaching the square root of 2. Repeating this procedure over and over will further increase the accuracy of the approximation.

Laying out all of the variables in order to make this into a program yields the following:

1. Assign the number (>0) whose square root is being approximated to the variable x.
2. Set an initial value (>0) for the square root approximation and assign this to the variable rnew. In this example, we will set this value equal to x.
3. Assign the value of rnew to the variable r1.
4. You can think of another approximation, x/r1, and assign this value to the variable r2 (r2 = x/r1). If r1 is the square root of x, r2 will also become the square root of x. If it isn't, one will be higher than the true value, and one will be lower, making the true value fall somewhere in between.
5. As such, the new approximation can be set to the average of r1 and r2, (r1 + r2)/2, and update the variable rnew by assigning it this new value.
6. Repeat steps 3-5 a certain number of times.

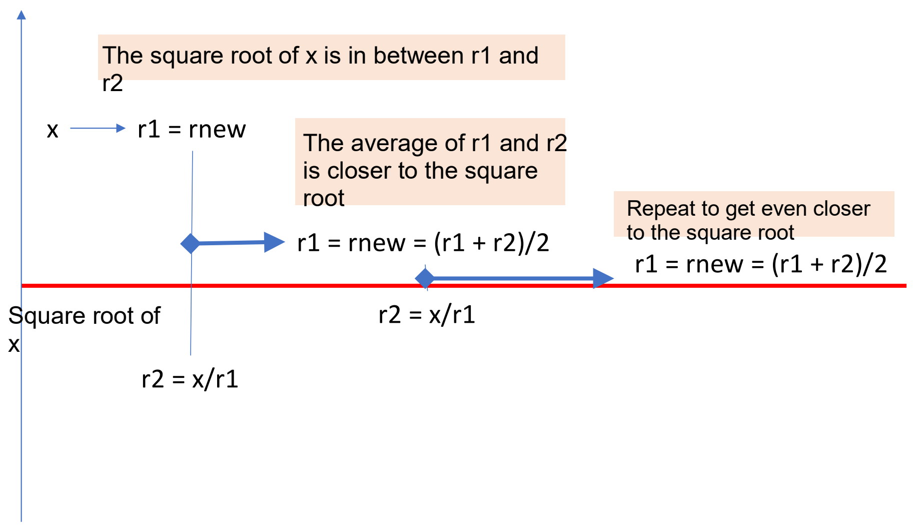

The square root of x is in between r1 and r2
x → r1 = rnew
The average of r1 and r2 is closer to the square root
Repeat to get even closer to the square root
r1 = rnew = (r1 + r2)/2
r1 = rnew = (r1 + r2)/2
Square root of x
r2 = x/r1
r2 = x/r1

Figure 3-4 Intuitive explanation of square root approximation

3.8.2 Python programs

Exercise 3-5 Creation and Execution of a program to obtain square root

Input the source code of Program 3-1 from the following table into the IDLE Editor, save it as p3-1.py, and run it.

Program 3-1 Program that solves for square roots (version 1, p3-1.py)

Row	Source code	Explanation
1	`#␣Find␣the␣square␣root␣of␣x`	Lines starting with # are
2	`x␣=␣2`	comments
3	`#`	
4	`rnew␣=␣x`	
5	`#`	Initial assumption for the
6	`r1␣=␣rnew`	approximation
7	`r2␣=␣x/r1`	
8	`rnew␣=␣(r1␣+␣r2)/2`	
9	`print(r1,␣rnew,␣r2)`	
10	`#`	
11	`r1␣=␣rnew`	
12	`r2␣=␣x/r1`	The code below this section
13	`rnew␣=␣(r1␣+␣r2)/2`	simply repeats the red
14	`print(r1,␣rnew,␣r2)`	portion 3 times.
15	`#`	
16	`r1␣=␣rnew`	
17	`r2␣=␣x/r1`	
18	`rnew␣=␣(r1␣+␣r2)/2`	
19	`print(r1,␣rnew,␣r2)`	
20	`#`	
21	`r1␣=␣rnew`	
22	`r2␣=␣x/r1`	
23	`rnew␣=␣(r1␣+␣r2)/2`	
24	`print(r1,␣rnew,␣r2)`	

If you obtain the following result, the program ran as expected.

```
2 1.5 1.0
1.5 1.4166666666666665 1.3333333333333333
1.4166666666666665 1.4142156862745097 1.411764705882353
1.4142156862745097 1.4142135623746899 1.41421143847487
```

With 4 repetitions, you will obtain the result of 1.4142135623746899.

If you calculate the root by using

```
2**(1/2)
```

you will get

```
1.4142135623730951
```

Note that the result of the program is accurate to the 11th digit after the decimal point.

An intuitive explanation of this approximation method is given above, however this is simply an application of what is known as Newton's method to solving for square roots. For more details, reference the column titled "Newton's Method."

Exercise 3-6 Experience errors (1)

Try a mistake of spelling "rmew" instead of "rnew" in line 4 of Program 3-1, and examine what will happen if you run the program. See also "17 How to Read Error Messages in IDLE/Python."

Exercise 3-7 Solve for the square root of some other numbers.

1. Alter p3-1.py to solve for the square roots of other positive numbers.
2. Also, try and see what happens when this program attempts to solve for the square root of 0. Don't just look at the error message, actually go through the program step-by-step (called tracing) to see where the problem occurs.

3.9 Be careful when using division

In programs, you will frequently run into situations where a variable is used as a divisor. Among the four basic arithmetic operations, division has the unique feature in which it cannot work if 0 is used as the divisor. When programming, be constantly mindful of whether or not the divisor in a division problem will be 0.

3.10 Notation to make equations easy to read

From the 8th line of p3-1.py, the equation

```
rnew = (r1 + r2)/2
```

uses the assignment operator (=), addition (+), division (/), and () which determine operation priority.

- Note that **spaces are placed before and after = and +** in order to make the equation easier to read.
- On the other hand, spaces are not placed directly within the () or on either side of the /. Removing all of the spaces yields

  ```
  rnew=(r1+r2)/2
  ```

  which looks pretty cramped and is difficult to read.
- A general principle to follow would be to forego spaces before and after high priority operations like * and /, while adding spaces before and after low priority operation like +, -, and =.

3.11 Assigning multiple variables

In Python, multiple variables can be assigned in one line by separating variables and expressions with “,” on both the left and right sides of the equation.[1]

```
a = 1
b = 2
c, d = a*2, b*c
print(c, d)
```

Running this gives

```
2 4
```

as the output.

In order to switch the values of two variables, most programming languages assign the value to a temporary variable (tmp in the example below) as shown below.

```
a = 1
b = 2
tmp = a
a = b
b = tmp
```

In Python, this can be done in the following manner.

```
a = 1
b = 2
a, b = b, a
```

References

[13] PEP 8 -- Style Guide for Python Code, https://www.python.org/dev/peps/pep-0008/ (Accessed on 2/12/2020)

[1] Although this is not explained here, a data type called a tuple is being used here behind the scenes.

4. Lists

4.1 Learning goals of this chapter

Until now, we've dealt with simple numbers and simple strings. In Python, there are multiple ways to handle batches of data all at once. Lists are one of these methods. In this chapter, you will learn the following things about using lists in Python.

1. What is a list?
2. How is a list created?
3. Learn how to access the elements of a list.
4. Learn about assigning and copying lists
5. Learn the basics about tuples and dictionaries, which are alternative ways to handle a bunch of data all at once.

4.2 Learning with the Python shell

This chapter is going to have a lot of short code. Rather than inputting it into the editor and running it, you can learn more efficiently by inputting it into the Python shell and seeing what it does.

- Because the Python shell processes code line-by-line, you cannot just copy and paste multiple lines into the shell and run them. Instead, input code one line at a time.

In order to limit the number of pages in this text, **input will be printed in red font** and **output will be printed in blue font** below. Although the Python shell prompt is omitted, please proceed by inputting the red text and confirming that it results in the outputted blue text.

4.3 What is a list?

In everyday life, if you mention a "shopping list," this refers to a short memo containing a list of items you intend to buy. In a similar manner, a Python list is a way for multiple pieces of data to be handled all at once. By ascribing multiple pieces of data, a set order, you are able to handle them as a single entity. For example, inputting

```
a = [5, 1, 3, 4]
```

and then

```
print(a)
```

prints the entire list.

```
[5, 1, 3, 4]
```

Inputting

```
print(a[0])
```

prints the element of the list at index zero.

```
5
```

Inputting

```
print(a[2])
```

prints the third element of the list.

```
3
```

4.4 Generating lists

4.4.1 Generation by specifying elements

Lists are generated by enclosing the elements within [] and separating them using “,” like this:

```
a = [5, 1, 3, 4]
```

Or this:

```
b = ['Sanjyo', 'Shijyo', 'Gojyo', 'Shichijyo']
```

As you can see, strings can be used as elements of a list. Also,

```
c = 5
a = [c, 1, 3, 4]
```

variables and equations can be included as elements of a list.

A long list of the same elements can be made by multiplication of list and an integer:

```
a = [1]*4
a
[1, 1, 1, 1]
```

However, it should be noted that if the element in the list is a complex objects like a list, multiplication makes not make a list of copies of the element but just a list of the identical elements.

4.4.2 Combining with the range() function

An empty list can be generated as an object of the list class like so:

```
e = list()
```

You can combine this with the range() function, which generates a series of numbers, like so:

Next Chapter Table of Contents

```
n = list(range(5))
```

Printing this list using

```
print(n)
```

will show a list of numbers from 0 to 4.

```
[0, 1, 2, 3, 4]
```

4.4.3 Generating lists from strings

It is possible to generate lists using strings instead of the range() function. Inputting

```
s = list('abcde')
```

and then

```
print(s)
```

Printing this list using

```
['a', 'b', 'c', 'd', 'e']
```

As you can see, you get a list containing the string ‘abcde’ split up into individual letters.

The string class contains a method called split() which can split a string into smaller segments using a specified character to delimit the breaks[1]. For example, inputting

```
t =  “a Python textbook”

tlist = t.split()
```

Printing this list using

```
print(tlist)
```

yields

```
['a', 'textbook', 'of', 'Python']
```

which is a word list generated by using spaces as the delimiter.

4.5 Methods

In the previous example, we used the split() method to split a string using spaces as the delimiter. Data handled by Python are generally referred to as “objects,” and the ways you can operate on this data is determined in advance, and they change based on the class, type, and value of the data. These predetermined operations are called **“methods,”** and can be called by placing a period after the variable name and then writing the name of the method. In the previous example, the split() method

[1] Calling this method without an argument will cause it to default to using spaces as the delimiter. You can specify a desired delimiter. You can also input help(str.split) into the Python shell to get an explanation of this method.

Next Chapter Table of Contents

was called by placing a period after the variable t and then including “split()” after the period.

```
t = “a Python textbook”

tlist = t.split()
```

A method can also be called by attaching it to the string itself. For the example above, this would become:

```
tlist =  ”a Python textbook”.split()
```

A few methods for lists will be introduced below, so be sure to remember what a method is and how to call one.[1]

4.6 Accessing elements in a list

Elements of a list can be accessed by including the element’s index within [].

```
a = [5, 1, 3, 4]

print(a[0])
```

yields

```
5
```

Then, by inputting

```
a[1] = 2

print(a)
```

2 is assigned to the second slot (index 1) of a, yielding

```
[5, 2, 3, 4]
```

The length of a list can be obtained using the len() function.

```
print(len(a))
```

yields

```
4
```

Please note that this is a function, **not a method that can be called by “a.len()”**

[1] Methods are very similar to functions, which will be introduced in a later chapter. The major difference is that functions *are not* tied to specific objects, while methods *are* mainly restricted to operating on certain objects. The way to call them is also similar. Classes exist as a way for programmers to set their own data types that include their own method definitions. These will also be introduced in a later chapter.

4.7 Negative indices and slicing

Python allows for list indices to be described in a wide variety of ways.

4.7.1 Negative indices

Using an index of -x (x being an integer) refers to **the element whose index is x elements from the end of the list.**

```
a = [5, 1, 3, 4]
print(a[-1])
```

yields the last element

```
4
```

See the table below for reference.

a[	5,	1,	3,	4	]
Accessing using a positive index	a[0]	a[1]	a[2]	a[3]	
Accessing using a negative index	a[-4]	a[-3]	a[-2]	a[-1]	

4.7.2 Slicing

By making the index [starting index:stopping index], you can take out a select part of a list. This is called a **slice** of the list. Please keep in mind that this includes **up to the index before the stopping index.**

```
a = [5, 1, 3, 4]
b = a[1:3]
print(b)
```

yields

```
[1, 3]
```

4.8 Adding to and combining lists

Lists come with various predetermined methods[1]. In this section, we will introduce the append() method, which adds elements to a list, and the extend() method, which combines multiple lists together.

[1] Inputting “help(list)” into the Python shell will enable you to read an explanation on the methods available for lists.

4.8.1 append method

This method adds the arguments to the end of a list.

```
a = [5, 1, 3, 4]
a.append(2)
print(a)
```

This adds 2 to the end of list a, resulting in

```
[5, 1, 3, 4, 2]
```

4.8.2 extend method

The extend() method is used to combine two lists.

```
a = [5, 1, 3, 4]
b = [2, 6]
a.extend(b)
print(a)
```

This will add list b to the end of list a, resulting in the following output:

```
[5, 1, 3, 4, 2, 6]
```

Note: using the append method will append the entirety of list b as the final element of list a, as seen in the example below.

```
a = [5, 1, 3, 4]
b = [2, 6]
a.append(b)
print(a)
```

The entirety of list b is appended as the final element of list a, leading to the following output:

```
[5, 1, 3, 4, [2, 6]]
```

4.9 List assignment and duplication

First, try running the following program:

```
a = [1, 2, 3]
b = a
print(a)
print(b)
```

This should result in the following output:

```
[1, 2, 3]
[1, 2, 3]
```

Next, try to predict what the result will be from running a program like the one below.

```
b[0] = 0
a[1] = 0
print(a)
print(b)
```

This does not result in

```
[1, 0, 3]
[0, 2, 3]
```

but rather

```
[0, 0, 3]
[0, 0, 3]
```

This is because **variables a and b refer to the exact same list.** Checking this by inputting

```
print(id(a), id(b))
```

reveals that a and b both have the exact same id (which will vary based on your operating environment).

In Python, the variables themselves do not contain the data, but rather the location of the data. Thus,

```
b = a
```

does not assign the data that a represents to b. Rather, it assigns to b the location of the data (list) represented by a. As such, any changes to the elements of either a or b are in fact changes made to the same list.

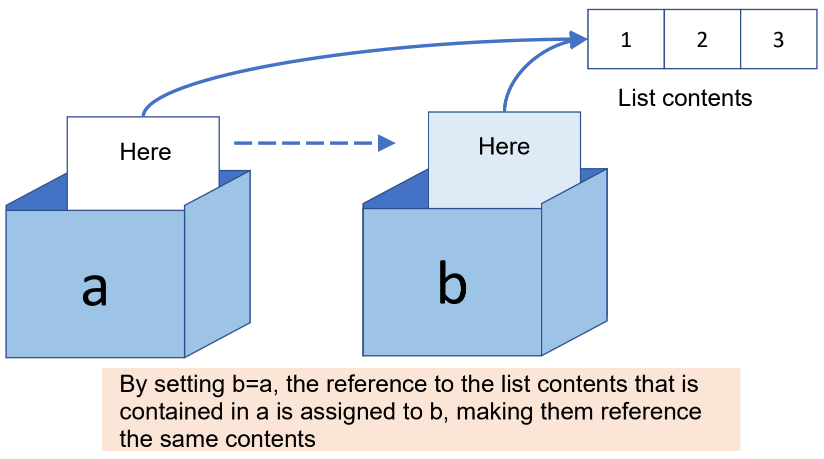

1
2
3
List contents
Here
Here
a
b
By setting b=a, the reference to the list contents that is contained in a is assigned to b, making them reference the same contents

Figure 4-1 List assignment

If you wish to handle b as an entity independent of a, you will need to explicitly create a copy and assign it (when using lists).

```
b = a.copy()
```

This same issue can occur when passing lists into a function as arguments. Reference the column titled "Reference & Duplication."

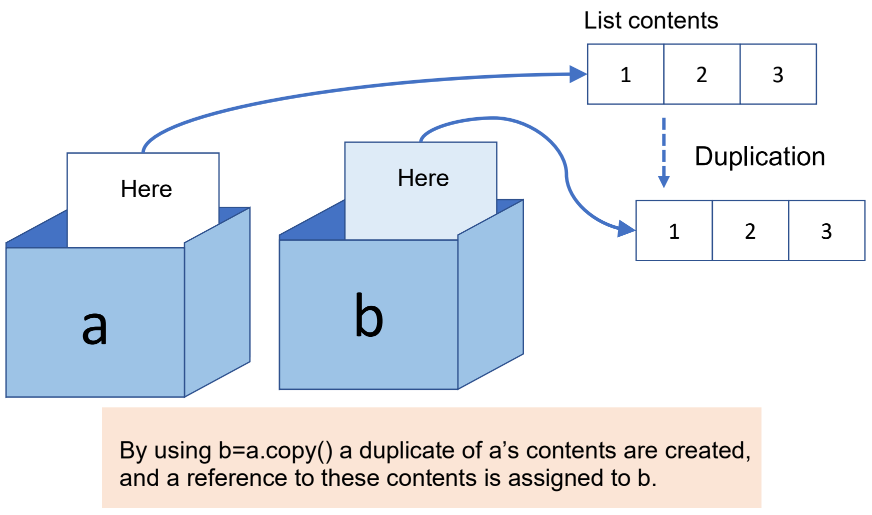

List contents
1
2
3
Duplication
Here
Here
1
2
3
a
b
By using b=a.copy() a duplicate of a's contents are created, and a reference to these contents is assigned to b.

Figure 4-2 Copying and assigning lists

4.10 Mutable and immutable objects

Everyone should now be able to correctly predict how a program with the following code will behave.

```
a = 1
b = a
```

```
b = 2
print(a, b)
```

An important concept regarding data management in Python is the idea of **mutable** versus **immutable** objects.

4.10.1 Numbers and strings are immutable (unchangeable) objects

In Python, numbers and strings are what are known as **immutable objects**. This means that in the third line of the program above, b = 2 does not overwrite the data (with a value of 1) currently referenced by b. Rather, it creates an entire separate entity containing the data "2" and assigns the location of this data to b. You can examine their locations in memory using the code below.

```
a = 1
b = a
print(id(a), id(b))
b = 2
print(id(a), id(b))
```

Running this gives the following:

```
>>> a = 1
>>> b = a
>>> print(id(a), id(b))
1434938848 1434938848
>>> b = 2
>>> print(id(a), id(b))
1434938848 1434938880
```

As you can see, a and b refer to the same location in line 3. However, by line 5 they refer to different locations.

4.10.2 Lists are mutable objects

On the other hand, lists are **mutable objects whose element can be changed**. Because of this, when there are two variables (a and b) containing references to the same list, any changes to the elements of either of these variables will affect the other.

4.11 Shallow and deep copying

Unfortunately, there are additional issues to consider when using lists. What does variable b refer to in the following program?

```
a = [[1, 2], [3, 4]]
```

```
b = a.copy()
```

Let’s see what happens when we do the following:

```
b.appen([5, 6])
print(a)
[[1, 2], [3, 4]]
print(b)
[[1, 2], [3, 4], [5, 6]]
```

As you would expect, adding append() to b works without affecting a because we created a copy of a and assigned it to b. Next, try the following:

```
b[0][0] = 0
print(a)
[[0, 2], [3, 4]]
print(b)
[[0, 2], [3, 4], [5, 6]]
```

This time, the assignment to the element at index [0][0] of b affected a as well. When we check to see what a[0] and b[0] each refer to, we get:

```
print(id(a[0]), id(b[0]))


3188920520008 3188920520008
```

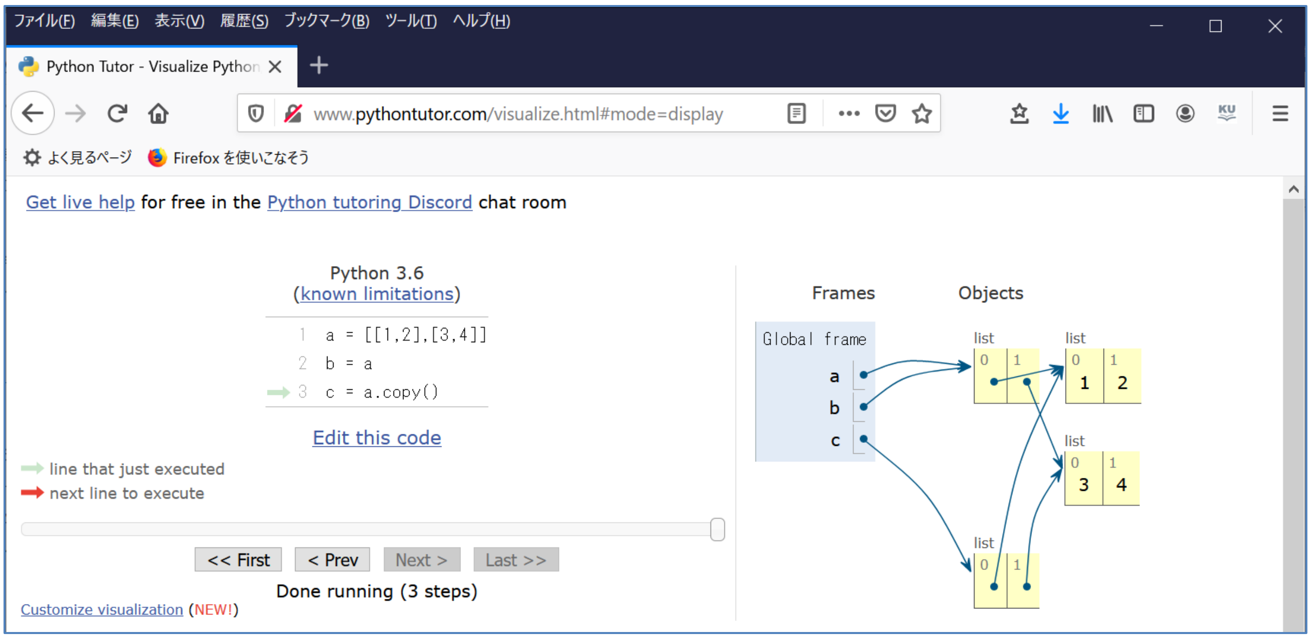
As you can see, they reference the same object.

This is because the copy() method prepares a new list separate from the target list (copy source) and then transcribes the locations of each element over to this new list. Thus, when the elements themselves are lists, the method does not create copies of them. This kind of copy is called a **shallow copy**, and a copy in which the elements themselves are also copied is called a **deep copy**.

4.12 Visualizing lists

There is a website called Python Tutor (http://www.pythontutor.com) where you can input short Python programs and see their behavior (variable usage) visualized. An example can be seen below. Using it to see how lists behave can serve to greatly aid your understanding.

Next Chapter Table of Contents



Get live help for free in the Python tutoring Discord chat room

```
a = [[1,2],[3,4]]
b = a
c = a.copy()
```

Done running (3 steps)

Figure 4-3 Checking how lists behave with Python Tutor

4.13 Keeping Calculation Results in a List

On calculation of square root studied in the previous chapter（p3-1.py）, rewrite the program so as to keep calculation process in a list.

Program 4-1, Keeping calculation process in a list (p4-1.py)

Row	Source code	Explanation
1	`#␣Find␣the␣square␣root␣of␣x`	Lines starting with # are comments
2	`x␣=␣2`	
3	`#`	
4	`rnew␣=␣x`	
5	`rnew_list = [rnew]`	Create a list with rnew as an element
6	`#`	
7	`r1␣=␣rnew`	
8	`r2␣=␣x/r1`	
9	`rnew␣=␣(r1␣+␣r2)/2`	
10	`print(r1,␣rnew,␣r2)`	
11	`rnew_list.append(rnew)`	
12	`#`	The code below this section simply repeats the above 3 times.
13	`r1␣=␣rnew`	
14	`r2␣=␣x/r1`	
15	`rnew␣=␣(r1␣+␣r2)/2`	
16	`print(r1,␣rnew,␣r2)`	
17	`rnew_list.append(rnew)`	
18	`#`	
19	`r1␣=␣rnew`	
20	`r2␣=␣x/r1`	

Next Chapter Table of Contents

`rnew␣=␣(r1␣+␣r2)/2`	
`print(r1,␣rnew,␣r2)`	
`rnew_list.append(rnew)`	
`#`	
`r1␣=␣rnew`	
`r2␣=␣x/r1`	
`rnew␣=␣(r1␣+␣r2)/2`	
`print(r1,␣rnew,␣r2)`	
`rnew_list.append(rnew)`	
`print(rnew_list)`	print the list

Exercise 4-1Store calculation process in a list

Write Program 4-1 and execute it. After execution, examine the length and elements of rnew_list with Python Shell.

4.14 Tuples and dictionaries

Aside from lists, two ways you can handle large amounts of data all at once in Python is by using tuples and dictionaries.

4.14.1 Tuple

Similar to lists, tuples are a data type that consists of multiple elements.

Tuples can be created by simply delimiting the elements on the right-hand side of the equation with commas and assigning them to the variables on the left-hand side.

```
a =1, 2
a
(1, 2)
```

The right-hand side can also be surrounded by parenthesis.

```
a = (1, 2)
a
(1, 2)
```

Actually, data is treated as tuples in assignments of and function return statements containing multiple values. If the left side has a number of variables equal to the number of elements, each element will be assigned to its respective index.

```
(b, c) = a
b
1
```

```
c
2
```

As with lists, the elements of a tuple can be referenced using [] and the correct index.

```
print(a[0])
1
```

However, tuples are immutable objects, so elements cannot be assigned new values after they are created.

```
a[0] = 2
```

The above code will yield the following error:

```
Traceback (most recent call last):
File "<pyshell#9>", line 1, in <module>
a[0] = 2
TypeError: 'tuple' object does not support item assignment
```

4.14.2 Dictionaries

With lists, you access elements using a numerical index. This is useful when the numerical order of the elements is meaningful. Dictionaries are a data type in which elements are accessed using strings (words). The string that functions as the index is called a “key.”

```
age = {"Yamada":18, "Tanaka":19}
age
{'Yamada': 18, 'Tanaka': 19}
age["Yamada"]
18
```

Referencing a key that does not exist will result in an error, but assigning it simply adds it to the dictionary.

```
age["Sato"] = 20
age
{'Yamada': 18, 'Tanaka': 19, 'Sato': 20}
```

The existence of a key in a dictionary can be checked using the 'in' operator.

```
"Okada" in age
False
```

Next Chapter Table of Contents

As with lists, tuples and dictionaries can be used to iterate through for loops.

4.14.3 Notation for lists, tuples, and dictionaries

Lists, tuples, and dictionaries all use different types of brackets. The correct notation for each type can be seen in the table below.

Table 4-1 Notation for lists, tuples, and dictionaries

Data type	Brackets used	Example	Accessing elements	Mutable/ Immutable
Lists	[ ]	d = [0, 1, 2]	d[0]	Mutable
Tuple	( )	d = (0, 1, 2)	d[0]	Immutable
Dictionaries	{ }	d = {"a":1, "b":2, "c":3}	d["a"]	Mutable

5. Control Flow

5.1 Learning goals of this chapter

In this chapter, you will learn the following methods for controlling the execution of a program.

1. Repeated processing and range() functions using for and while loops
2. Branching using if statements
3. Error handling with try statements

In addition, you will learn the following things which are related to the above points

4. How to write conditional expressions
5. Getting input from the keyboard using the input() function
6. Mathematical functions in Python
7. Format designation for outputted text

There is a lot to learn in this chapter, but the material covered will show up in many programs throughout the following chapters. You should be able to pick up the material naturally after seeing many instances of it in use.

It isn't necessary to learn all the fine details at this point. It is important to separate the concept of "**what can this code do**" from the **experience of actually writing it**. If you remember what certain code can do and how to write it in a general sense, you can always check out the fine details in your textbook when the need arises.

5.2 Repeated processing using for loops

One of the main objectives of programming is to have computers automatically do things that are difficult and/or repetitive for people to do by hand. For loops are an important method for programming the repetitive parts. Those of you who are familiar with C will notice Python's for statement has a slightly different syntax, but it is easier to use because you can easily write iterations over elements of a list.

5.2.1 Performing a fixed number of repetitions using a for loop and the range() function

The times that computers can show their true might through programming is when they are able to perform a lot of processing at a very high speed. However, using sequential execution learned in the Chapter 3, you need to write each and every step that the computer executes. Program 3-1 for solving square roots in Chapter 3 repeated the exact same code four times. Here you will learn how to use a for loop with the range() function to automate these repetitions.

Next Chapter Table of Contents

Exercise 5-1 Apply for statement to the program of solving square root

Write and run the program contained in the table below.

Program 5-1 Program that solves for square roots (ver. 2, p5-1.py)

Row	Source code	Explanation
1	`#␣Find␣the␣square␣root␣of␣x`	Lines starting with # are comments
2	`x␣=␣2`	
3	`#`	
4	`rnew␣=␣x`	
5	`#`	
6	`for␣i␣in␣range(10):`	Repeats the code below while iterating i from
7	`␣␣␣␣r1␣=␣rnew`	0 to 9. Note the colon at the end of line 6.
8	`␣␣␣␣r2␣=␣x/r1`	In Python, the repeated portion (block) is
9	`␣␣␣␣rnew␣=␣(r1␣+␣r2)/2`	indented (the recommended amount is 4
10	`␣␣␣␣print(r1,␣rnew,␣r2)`	spaces).

It is possible to indent large portions of text all at once in the IDLE Editor by selecting the text, holding down the Ctrl key, and pressing the] key (Ctrl-]). Text can be unindented by using the Ctrl-[shortcut.

5.2.2 Writing for loops[1]

The word "for" has many different meanings. In the context of programming, thinking of it as "for the sake of" can make things difficult to understand. It is better to think of it as closer to its meaning that is synonymous with "regarding." For loops in Python are written in the following manner.

```
for target variable in range of repetition :
    Repeated block
```

In the p5-1.py example above, the target variable is 'i', the range of repetition is 'range(10),' and the block is the indented code in lines 7-10.

The range(10) function generates 10 values from 0 to 9 (index of -1). The for loop puts the generated values into the variable i and repeats the block.

Translating the for loop into plain English would yield the following.

For values of the target variable within the range of repetition, repeat the repeated block. From here onward, explanations of Python's syntax will be enclosed in a frame like the example above. Fixed expressions will be written in red, and parts that can change based on context will be written in black.

[1]For loops in Python can be written very effectively using lists and other data types. These usages will be introduced later. This chapter will cover using for loops with the range() function.

Exercise 5-2 Checking the block
Unindent line 10 from the block in the previous example (p5-1.py) as shown below. Check and explain how the program behaves.

Program 5-2 Program that solves for square roots (ver. 2, p5-2.py)

Row	Source code	Explanation
1	`#␣Find␣the␣square␣root␣of␣x`	Lines starting with # are comments
2	`x␣=␣2`	
3	`#`	
4	`rnew␣=␣x`	
5	`#`	
6	`for␣i␣in␣range(10):`	
7	`␣␣␣␣r1␣=␣rnew`	
8	`␣␣␣␣r2␣=␣x/r1`	
9	`␣␣␣␣rnew␣=␣(r1␣+␣r2)/2`	
10	`print(r1,␣rnew,␣r2)`	Unindent this line from the block

Exercise 5-3 Prank
The repeated portion in the program above (p5-2.py) can be run very quickly because the part that outputs to the terminal has been removed from the for loop. Change the index of the range() function on line 6 from 10 to 100, 1000, 10000, 100000, 1000000, and 10000000 to see how long it takes to run.[1]

5.2.3 Blocks in Python

Multiple line portions that are all executed as a unit are called **blocks**. This concept is very important in the world of programming.

- Blocks in Python are denoted by indenting them all by the same amount. This is one of Python's defining features.
- For statements and other lines that require a subsequent block of code require that a colon (:) be placed at the end of the line.
- As indentation carries significant meaning in Python programs, you should stick to the standard practice of indenting by 4 spaces.
- When for statements and other lines that require a subsequent block of code are entered into the IDLE Editor, it will automatically indent the next lines.
- **Using full-width spaces in an indent will result in an error. Be careful not to do this, as it can be a very difficult error to spot.** Also, while this can depend on the settings of the editor being used, using the TAB key will result in an error if the TAB code is input as is.

[1] Modern personal computers clock in at the GHz level. At one instruction per clock cycle, this leads to extremely fast processing. However, Python takes longer to process than most programming languages due to the fact that it is run via intermediate representation.

- In other languages, like C, blocks are enclosed in { }. Students who are familiar with other languages will need to take care to use the correct notation.

Exercise 5-4 Experience errors (2)

Statements such as "for" statement that require a succeeding block have to end with a colon ":". Examine what will happen if you forget to type ":", and run the program. Or also try a mistake of writing semicolon ";" instead of ":". See also "17 How to Read Error Messages in IDLE/Python."

Exercise 5-5 Experience errors (3)

In Program 5-1, lines 7 through 10 belong to a same block with a same indentation (four spaces). Examine what will happen if you put less spaces (e.g., three spaces) or more spaces (e.g., five spaces) in line 7, and run the program. See also "17 How to Read Error Messages in IDLE/Python."

5.2.4 Controlling the processing within a for loop

Break and continue statements can be used within for loops in order to abort processing or to skip processing at a specific repetition.

- **break:** Breaks out from the for loop
- **continue:** Skips the remaining code in the block of the for loop and continues to the next iteration of the loop.

These statements are used in conjunction with if statements, which enable conditional branching and will be introduced later.

Program 5-3 continue and break (p5-3.py)

Row	Source code	Notes
1	`for␣i␣in␣range(10):`	
2	`␣␣␣␣if␣i␣==␣1:`	Move to the next iteration
3	`␣␣␣␣␣␣␣␣continue`	if i is 1
4	`␣␣␣␣if␣i==␣8:`	Break from the loop if i is
5	`␣␣␣␣␣␣␣␣break`	8
6	`␣␣␣␣print(i)`	

This program results in the output shown below.

```
0
2
3
4
5
6
```

```
7
```

Exercise 5-6 Explanation of continue and break

Use the source code to explain the output seen above.

5.2.5 range() function

To be precise, range is implemented as a class and not a function. However, as it is often used similarly to a function, we will refer to it as the range() function in order to facilitate easier understanding. range() generates a sequence of numbers over a set interval. However, you cannot know exactly what values are generated by merely looking at the calling of the range() function itself. The generated values can be checked by creating a list.

Inputting

```
list(range(10))
```

into the Python shell yields

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The range() function can be used in three different ways based on what arguments are passed to it. These usages can be seen below.

- Only an ending value is passed as an argument. This takes the form

  ```
  range(ending_value)
  ```

  as seen above. This starts at 0 and stops at the integer before the ending value.
  Students who aren't as familiar with programming languages may feel that it's odd that **it starts with 0** and that the **ending value is not included**. Please refer to the column titled "Starting from 0" for information about this. The **total number of numbers generated still matches the ending value**, so it isn't that difficult to remember. It is standardized throughout Python to start with 0 and stop at the value before the specified end point.

- Both a starting value and an ending value are passed as arguments.

  ```
  range(starting_value, ending_value)
  ```

  The starting value is included, but the ending value is not. A comma must be placed between the two arguments. Placing a half-width space between the arguments will make it easier to read.

- A starting value, ending value, and step width are passed as arguments.

  ```
  range(starting_value, ending_value, step_width)
  ```

Exercise 5-7 range() function

Practice the three usages of the range() function in the Python shell using list() as shown above.

5.2.6 Calculating sums

In the previous example, the range function was merely used to specify the number of repetitions, and the target variable was not used in any particular way. In this example, we will examine a program that actually utilizes the target variable.

Program 5-4 Calculating sum (P5-4.py)

Row	Source code	Notes
1	`sum␣=␣0`	Sets sum to 0
2	`for␣i␣in␣range(10):`	i goes from 0 to 9
3	`␣␣␣␣sum␣=␣sum␣+␣i`	adds i to the sum
4	`print(sum)`	displays the sum

Exercise 5-8 Calculation of sum

Try running Program 5-4. Also, try changing the scope of the sum. Line 3 can also be written as sum += i. Try this out for yourself.

5.2.7 Nested "for" loops

Values for two-dimensional structures such as tables, which contain rows and columns, can be generated by nesting a for loop within the block of another for loop. For example,

Program 5-5 Nested "for" loops (p5-5.py)

Row	Source code	Explanation
1	`for␣i␣in␣range(3):`	
2	`␣␣␣␣for␣j␣in␣range(3):`	
3	`␣␣␣␣␣␣␣␣print(i,j)`	

Running the code above yields the following output:

```
0 0

0 1

0 2

1 0

1 1

1 2

2 0

2 1

2 2
```

Please note that the target variable for the inner for loop is different than that of the outer for loop.

Exercise 5-9 Nested for loops

Try and find out what happens when the variable i is used as the argument for the range() function (change it to range(i)) in line 2.

5.2.8 Using for loops to manage lists

1) How to combine list length with the range function

The elements of a list can be accessed in order with a for loop by using range(len(a)) to generate each element. For example:

```
a = [5, 1, 3, 4]
for i in range(len(a)):
   print(i, a[i])
```

yields

```
0 5
1 1
2 3
3 4
```

2) How to directly use lists with a for loop

If you simply desire to reference the element values, the code below would also suffice.

```
a = [5, 1, 3, 4]
for d in a:
    print(a)
```

yields

```
5
1
3
4
```

In this case, the contents of the elements are passed to d, so changing d does not change the list's contents. When you wish to update the elements of a list, you can access them using their index.

3) How to use the enumerate function

The enumerate function is used in cases where you want to use both the index of an element and its value.

```
a = [5, 1, 3, 4]
for i, d in enumerate(a):
    print(i, d)
```

```
0 5
1 1
2 3
3 4
```

Exercise 5-10 Solving for averages
To solve for the average value of a list whose elements are numbers, you could use the following code:

```
a = [5, 1, 3, 4]
sum = 0
for i in range(len(a)):
    sum += a[i]
average = sum/len(a)
print(average)
```

The result of this code is:

```
3.25
```

Exercise 5-11 Change access to list elements

Rewrite the above program in a manner such that it uses a for loop to use the list directly.

5.2.9 List comprehension using for loops

Consider a list of numbers risen to the second power.

```
[0, 1, 4, 9, 16]
```

Next Chapter Table of Contents

You can make this by explicitly writing it out, as done above, or you could use the following code:

```
a = []
for i in range(5):
    a.append(i*i)
```

In addition, Python has another way to do this, called list comprehension, that entails including a for loop within the list.

```
a = [i*i for i in range(5)]
```

5.3 Repetition using while loops

5.3.1 Calculating the square root with specified precision

Let's try calculating square roots to a set level of precision. Because r1 and r2 lie on opposite sides of the true value, the absolute value of the difference between them, |r1 - r2|, can be taken to be the precision of the calculation. Let's write the program specifying the error to be no more than 10^{-6}.

Accuracy in this manner, in which you specify a certain number of digits past the decimal point, is known as absolute precision. However, in scientific calculations, there are many situations in which it is better to specify a number of significant digits. For more information on this subject, please refer to the column titled "Relative Accuracy."

Program 5-6 Program that solves for square roots (ver. 3, p5-6.py)

Row	Source code	Explanation
1	#␣Find␣the␣square␣root␣of␣x	
2	x␣=␣2	
3	#	
4	rnew␣=␣x	
5	#	
6	diff␣=␣rnew␣-␣x/rnew	Takes the difference between the two
7	if␣diff␣<␣0:	approximations (x and 1 (=x/x))
8	␣␣␣␣diff␣=␣-diff	Changes the sign if negative
9	while␣(diff␣>␣1.0E-6):	Repeats if the difference is larger
10	␣␣␣␣r1␣=␣rnew	than 10^{-6}
11	␣␣␣␣r2␣=␣x/r1	
12	␣␣␣␣rnew␣=␣(r1␣+␣r2)/2	
13	␣␣␣␣print(r1,␣rnew,␣r2)	
14	␣␣␣␣diff␣=␣r1␣-␣r2	
15	␣␣␣␣if␣diff␣<␣0:	Recalculates the difference
16	␣␣␣␣␣␣␣␣diff␣=␣-diff	

Exercise 5-12 Write and run Program 5-6

Running the program should yield the following result:

```
2 1.5 1.0
1.5 1.4166666666666665 1.3333333333333333
1.4166666666666665 1.4142156862745097 1.411764705882353
1.4142156862745097 1.4142135623746899 1.41421143847487
1.4142135623746899 1.414213562373095 1.4142135623715002
```

5.3.2 Calculating the square root with an infinite loop

A program that solves for square roots using an infinite loop is shown in Program 5-7. It calculates precision and uses an if statement to determine whether or not to exit the loop via a break statement. This program differs from p5-6.py in two ways.

- An end condition is not specified at the beginning of the loop.
- The condition of the while statement determines whether to continue, but the condition that determines whether to activate the break statement is a condition that judges whether to stop. So, in this manner, the conditions are inverted.

Program 5-7 Program that solves for square roots (infinite loop method, p5-7.py)

Row	Source code	Explanation
1	`#␣Find␣the␣square␣root␣of␣x`	
2	`x␣=␣2`	
3	`#`	
4	`rnew␣=␣x`	
5	`#`	
6	`while␣True:`	Infinite loop repetition
7	`␣␣␣␣r1␣=␣rnew`	
8	`␣␣␣␣r2␣=␣x/r1`	
9	`␣␣␣␣rnew␣=␣(r1␣+␣r2)/2`	
10	`␣␣␣␣print(r1,␣rnew,␣r2)`	
11	`␣␣␣␣diff␣=␣r1␣-␣r2`	
12	`␣␣␣␣if␣diff␣<␣0:`	
13	`␣␣␣␣␣␣␣␣diff␣=␣-diff`	
14	`␣␣␣␣if␣diff␣<=␣1.0E-6:`	break if the absolute value of the difference is less than or equal to 10^{-6}
15	`␣␣␣␣␣␣␣␣break`	

5.3.3 Structure of a while loop

The format of a while loop, which repeatedly processes a block of code as long as the given condition remains true, is as follows:

```
while conditional expression:
    Block of code
```

The condition is checked before the block of code is executed. This is why, in the example above, the value of diff is calculated both before entering the while loop and inside of it. Details regarding how to write conditional expressions will be introduced in the coming sections.

As with for loops, break and continue statements can be used when you wish to break from the loop or enter the next iteration.

5.3.4 Infinite loops

You will also frequently encounter instances in which while loops are used in the manner seen below.

```
while True:
    Block containing a break statement
```

In this usage, the conditional expression is the constant True, meaning that the condition is always satisfied. This means that the while loop itself will continue infinitely. As such, you need to include a condition to exit the loop in the form of a break statement within the block.

If the break condition is not met, execution of Python will need to be forcibly stopped. **You can do this by using the keyboard shortcut Ctrl-C.**

5.4 Branching using if statements

5.4.1 Structure of an if statement

If statements can be used in a few different ways. The most basic is to execute a block of code if the condition in the if statement is met. In addition to this, you can include a block of code to execute if the condition is not met (else), and you can even include yet another condition to evaluate if the initial condition is not met (elif, meaning else if).

```
if conditional expression:
    Block to be executed if the condition is true
```

```
if conditional expression:
    Block to be executed if the condition is true
else:
   Block to be executed if the condition is false
```

```
if conditional expression 1:
    Block to be executed if conditional expression 1 is true
elif conditional expression 2:
    Block to be executed if conditional expression 1 is false but conditional expression 2 is
true
```

```
else:
    Block to be executed if both conditions are false
```

It is worth noting that Python does not have an equivalent to the switch statement found in C in which you can evaluate three or more conditions to switch between many blocks of code. You can achieve similar branching using multiple elif statements.

5.4.2 How to write conditional expressions

1) Comparing numerical values

Perhaps the most common condition is a comparison of numerical values. The operators that can be used to compare numerical values can be seen in the table below.

Table 5-1 Relational operators in Python

Operator	Meaning	Notes
==	Equivalent	Two equals signs. This is to differentiate it from the assignment operator, which is a single equals sign.
!=	Not equivalent	2 characters
>	Greater than	
<	Less than	
>=	Greater than or equal to	2 characters
<=	Less than or equal to	2 characters

The equivalence operator is two equals signs (==). Please take mental note of this, because it is easy to mix up with the assignment operator (=).

Also, it is worth noting that floating-point numbers are in many cases merely approximations. As such, the equivalence operator can potentially produce unexpected behavior. In order to avoid this, try to make determinations using inequalities when possible.

2) Comparing strings

The operators above can also be used with strings. However, please note that string values are determined by their character encoding (Unicode) number, so make sure to keep this in mind when comparing strings in this manner.

In addition to the above, you can use “in” to check whether the string on the left is included within the string on the right. For example,

Next Chapter Table of Contents

```
'a' in 'abc'
```

returns

```
True
```

3) Logical operations

The logical operators “and,” “or,” and “not” enable you to combine multiple conditions.

4) Prioritizing operations with ()

Python has a defined order of operations:

- Arithmetic operations are prioritized over comparison operations
- Comparison operations are prioritized over logical operations

It is good practice to use () in order to make your programs easier to read by explicitly stating the order of operations. For example:

```
a == 1 and b != 0
```

The code above behaves exactly the same as the code below, but the latter is easier to read.

```
(a == 1) and (b != 0)
```

5.4.3 Nested if statements

Just like how for statements are often nested within other for statements, if statements are also frequently used in this manner. The following two programs make the same determinations but are written in different ways.

Program 5-8 Forking using multiple conditions (p5-8.py)

Row	Source code	Explanation
1	a␣=␣1	
2	b␣=␣0	
3	if␣(a␣==␣1)␣and␣(b␣==␣0):	Forking with multiple
4	␣␣␣␣print("YES␣a==1␣and␣b␣==␣0")	conditions

Program 5-9 Forking with nested if statements

Row	Source code	Explanation
1	a␣=␣1	
2	b␣=␣0	
3	if␣a␣==␣1:	

4 5	␣␣␣␣if␣b==0: ␣␣␣␣␣␣␣␣print("YES␣a==1␣and␣b␣==␣0")	Forking with nested if statements

Exercise 5-13 Experience errors (4).

It is a common mistake of writing an assignment operator "=" instead of the comparison operator "==" in a "if" statement. Examine what will happen if you write "if a = 1: " in the third line of Program 5-9 and run it. See also "17 How to Read Error Messages in IDLE/Python."

5.5 Input from the terminal

So far, we've built the number whose square root we want to find into the program itself. Let's consider adding input from the terminal. Below is the Python shell screen. Red text is input, and blue text is output.

Python shell screen	Notes
`>>> a = input("*** ")`	Change the input prompt string to "***" and then accept input via the input function and assign it to a
`*** sss`	Input sss
`>>> a`	Evaluate a
`'sss'`	
`>>> type(a)`	Retrieves the data type of a using the type function
`<class 'str'>`	Displays that a is a string (str)
`>>>`	

The argument of the input function (the string within the parentheses) is the string that will be displayed. The function's return value (the result of the function call) is a string.

In order to obtain numerical data, convert it to the appropriate type using int() or float().

In the square root program from before, change the part that sets the value of x to

```
x = input("Find the square root of ")
x = float(x)
```

You could even write this in one line like so:

```
x = float(input("Find the square root of "))
```

This enables you to store the numerical value input into the terminal within the variable x.

Exercise 5-14 Input number form terminal

Modify p5-6.py so that it solves for the square root of the number input through the terminal.

Open p5-6.py or p5-7.py and select Save As in the File menu to create a new program named e.g., p5-6-1.py or p5-7-1.py.

5.6 Handling errors

When something that cannot be interpreted as a number is passed through the float() or int() functions as an argument, it results in a ValueError. If you have not specified what should be done in such a case, Python will suspend processing of the program at this point. You can make use of a try statement to deal with errors that may arise in a program.

The following program continuously accepts input and handles errors in order to only output (print(x)) positive numbers. This program contains an infinite loop that lacks an explicit stoppage condition. **Input Ctrl-C to stop the program.**

Program 5-10 Program that checks user input (input_check_en.py)

Row	Source code	Explanation
1	while␣True:	Infinite loop
2	␣␣␣␣x␣=␣input("Enter positive number␣")	
3	␣␣␣␣try:	Put the part that can give an
4	␣␣␣␣␣␣␣␣x␣=␣float(x)	error within the try block
5	␣␣␣␣except␣ValueError:	Handles a ValueError
6	␣␣␣␣␣␣␣␣print(x,␣"can't be converted to a number")	Make sure to capitalize letters when appropriate
7	␣␣␣␣␣␣␣␣continue	Handles other errors
8	␣␣␣␣except:	
9	␣␣␣␣␣␣␣␣print("Unecpected error")	Ends the program. This is the
10	␣␣␣␣␣␣␣␣exit()	end of the try block.
11	␣␣␣␣if␣x␣<=0:	Checks whether the input is
12	␣␣␣␣␣␣␣␣print(x,␣"is not positive")	positive
13	␣␣␣␣␣␣␣␣continue	Continuation of the while
14	#␣Below␣is␣the␣result␣of␣receiving␣a␣proper␣input	loop
15	␣␣␣␣print(x)	

Exercise 5-15 Reviewing Error Handling

Run the above program and see how it responds after receiving various inputs.

5.6.1 Structure of a try statement

Blocks that can generate exceptions are put within try statements, and specified exceptions are placed within except statements with a corresponding block. An except statement that does not specify an exception type will activate for any exception that is encountered (except for exceptions specified above it).

```
try:
    Block which will require exception handling
except exception:
   Block to be processed when the specified exception occurs
except:
   Block designed to handle all exceptions besides the ones specified above it
```

5.6.2 Always be suspicious of external input

The programmer cannot control the external input that they receive. Programs that are only written to be able to handle anticipated input are often unable to properly handle unexpected input. This often leads to incorrect results. It is **very important that you always distrust external input.** You should ideally always check its validity and include appropriate contingencies for any potential errors that can arise from unanticipated input.

5.7 Mathematical functions in Python

In the examples thus far, I've been explicitly calculating the absolute value of the calculation error diff by changing the sign if the number is negative, like so:

```
if diff < 0:
    diff = -diff
```

Python is equipped with an absolute value function, so this can be written in just one line:

```
diff = abs(diff)
```

In addition, Python is equipped with a library (module) containing many sorts of mathematical functions. To use this, simply write

```
import math
```

to import the module before you use it. Constants and functions defined in the math module can be called by writing the module name followed by a period and the name of the function or constant you wish to call. For example:

```
math.pi
math.sqrt(2)
```

The examples above are for mathematical constant pi and obtaining a square root.

Exercise 5-16 Use of math module

Mimic the examples above and play around with the math module yourself within the

Python shell.

5.8 Converting numbers and strings; Combining strings

We have used the input() function to obtain a string followed by the int() or float() functions to convert the string to a number. Additionally, you can also use the str() function to convert numerical data to a string. The format() method, which will be explained below, can be used in cases when you wish to specify the format. These conversion methods are summarized in the table below.

Table 5-2 Conversions between numbers and strings

Conversion	Function	Example	Notes
String to Integer	int()	a = int("123")	An improper string will result in a ValueError
String to Floating-point number	float()	a = float("123.4")	Same as above
Integer / Floating-point number to String	str()	s = str(123.4)	str() can also convert lists and other various types of objects into strings as well.

It is worth noting that numerical values and any other non-string values passed to the print function are automatically converted into strings.

Strings can be connected by using the "+" operator. In addition, strings can be repeated by using the "*" operator in conjunction with an integer. Also, you can combine a number with a string by using the "+" operator and the str() function.

Table 5-3 Connecting and repeating strings

Operation	Operator	Example	Result
Connecting strings	+	"abc" + "def"	"abcdef"
Connecting strings and numbers		"abc"+str(1.2)	"abc1.2"
Repeating strings	*	"abc"*2	"abcabc"

5.9 Format specification when displaying a number

When Python's print() function displays a numerical value, a number of digits that matches the number is automatically selected. It is possible for the user to specify both the number of digits and the format with which the number is displayed. A concrete example can be seen below:

```
c = 2.99792458E8

na = 6.02214076E23

form = "light speed is {0:12.8g} m/s, Avogadro's number is {1:12.8g} mol**(-1)"

print(form.format(c, na))
```

executing the above code, we obtain:

```
light speed is 2.9979246e+08 m/s, Avogadro's number is 6.0221408e+23 mol**(-1)
```

The above code enables you to pick both the number of digits and the format displayed[1].

- The right side of the third line is the string that specifies the format. **The portion enclosed in {}** is the format to which the number will be converted.
  - For example, {0:12.8g} acts upon the 0th element (the number before the colon in the format) of the argument of the **format method** below it,
  - displaying at least 12 digits and going to 8 digits (only 8 digits are displayed to the user) past the decimal point in g format (one of the formats used to display floating-point numbers).
  - Existing formats include 'd' format, which is used to display integers in base 10, 'e' format, which is used to display floating-point numbers using exponent representation, and 'f' format, which displays in fixed-point representation. 'g' format switches between both of these based on the value of the number.
- The form.format(c, na) on the fourth line generates a string in which variables c and na are converted to strings according to the format specified within form. This code calls the format method, which is a method belonging to string variables (form in this case). This method is called by connecting it to the string it acts upon with a period, i.e., ".".

5.10 Test of skills

Exercise 5-17 Test of Skills

Combine input_check_en.py and p5-6.py to create a program satisfying the following conditions that can solve for square roots. Do not try to grapple with everything listed below all at once. Try to sort out which section of the text is related to each item listed and then update your program to satisfy each condition one at a time. Test your program and confirm it is working as intended after adding each one.

1. Use the abs() function to calculate the absolute value.
2. Make the program repeatedly ask for what number's square root to solve for.
3. Make the program notify the user if the input cannot be converted into a number, then have the program ask for another input.
4. Make the program notify the user if the input is below 0, then have the program ask for another input.

[1] The International System of Units (SI) announced with its 2018 amendments that as of May 20, 2019, The International Prototype of the Kilogram would be replaced by a new definition of the kilogram based on physical constants. The values for the speed of light and Avogadro's number used in this example were not determined by actual measurements, rather they are defined to be certain quantities.

If you can, try your hand at the following:

5. Have the program end when the input received from the terminal is “end”
6. Make the calculation accuracy function based on relative accuracy (10^{-6}) instead of absolute accuracy. After implementing this, try solving for the square roots of very small and very large numbers (like 10^{10} and 10^{-10}) and see how the program performs.

Next Chapter Table of Contents

6. Making Kyoto Intersections

6.1 Learning goals for this chapter

In this chapter, we will use for statements and lists to practice. we will be going through an example that deals with Kyoto's intersections, which have a grid-like structure.

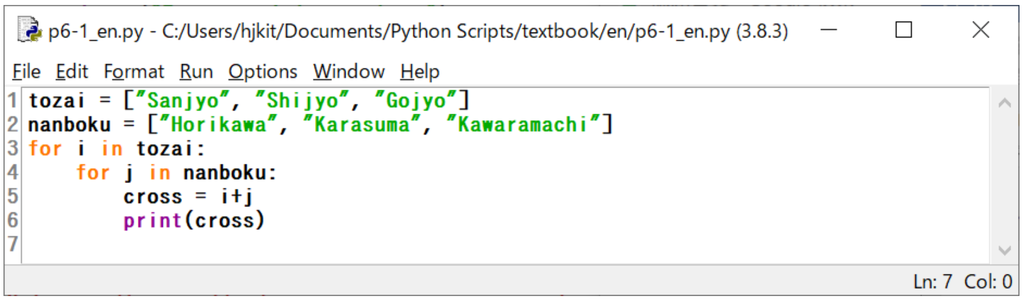
6.2 Creating Kyoto's intersections

Play around with the following example, which uses for statements to handle lists. Be careful if you decide to write the street names in Japanese characters, as the code will then mix half-width and full-width characters. Note that aside from place names, everything is written using half-width characters (including spaces).

Program 6-1 (p6-1_en.py)

Row	Source code	Explanation
1	`tozai␣=␣["Sanjyo",␣"Shijyo",␣"Gojyo"]`	Only the text in red font
2	`nanboku␣=␣["Horikawa",␣"Karasuma",␣"Kawaramachi"]`	uses full-width characters
3	`for␣i␣in␣tozai:`	
4	`␣␣␣␣for␣j␣in␣nanboku:`	
5	`␣␣␣␣␣␣␣␣cross␣=␣i+j`	Concatenate strings using the
6	`␣␣␣␣␣␣␣␣print(cross)`	+ operation

Your screen in the IDLE Editor should look like the screenshot below. Make sure that the reserved keywords for and in, the strings (such as "Sanjyo"), and the print function are all colored appropriately.


p6-1_en.py - C:/Users/hjkit/Documents/Python Scripts/textbook/en/p6-1_en.py (3.8.3)
File Edit Format Run Options Window Help
Ln: 7 Col: 0

```
tozai = ["Sanjyo", "Shijyo", "Gojyo"]
nanboku = ["Horikawa", "Karasuma", "Kawaramachi"]
for i in tozai:
    for j in nanboku:
        cross = i+j
        print(cross)
```

This will give the following result[1]

```
SanjyoHorikawa
```

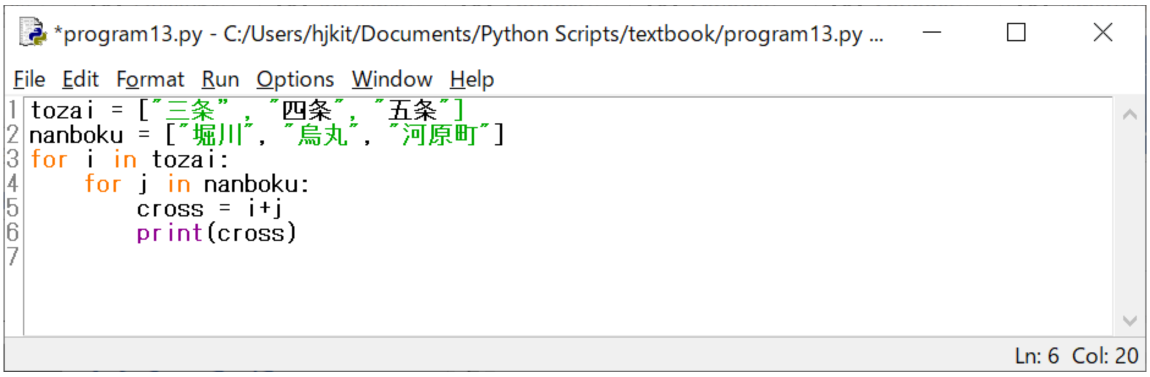
[1] There are cases in which Kyoto intersections are named with the east-west street said first (such as ShijyoKawaramachi) and there are cases in which the north-south street is said first (like HigashiyamaSanjyo). In this example, we will be generating the names mechanically, so they may differ from the way they are actually said.

Next Chapter Table of Contents

```
SanjyoKarasuma
SanjyoKawaramachi
ShijyoHorikawa
ShijyoKarasuma
ShijyoKawaramachi
GojyoHorikawa
GojyoKarasuma
GojyoKawaramachi
```

The following example, Street Name in Kanji because of showing a problem of full-width characters.

If you accidentally use a full-width ” to close a string, the text that is colored will change.

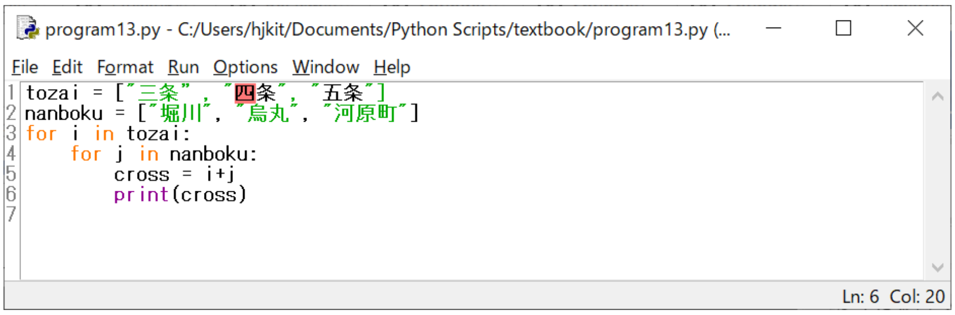


*program13.py - C:/Users/hjkit/Documents/Python Scripts/textbook/program13.py ...

```
tozai = ["三条”, "四条", "五条"]
nanboku = ["堀川", "烏丸", "河原町"]
for i in tozai:
    for j in nanboku:
        cross = i+j
        print(cross)
```

Ln: 6 Col: 20

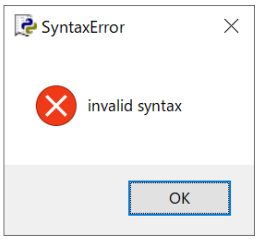
Running this will result in an error (invalid syntax). The 四 is displayed in red in the editor. This is because the double quote closing "三条 (Sanjyo)” is in doble-width character, and hence the double quote opening "四条 (shijyou)" is interpreted as double quotes that end the strings following ‘”三条.’



program13.py - C:/Users/hjkit/Documents/Python Scripts/textbook/program13.py (...

```
tozai = ["三条”, "四条", "五条"]
nanboku = ["堀川", "烏丸", "河原町"]
for i in tozai:
    for j in nanboku:
        cross = i+j
        print(cross)
```

Ln: 6 Col: 20



SyntaxError

invalid syntax

6.3 List of lists and how to scan them

Lists in Python can actually contain lists as elements. This fact can be used to create a **data structure resembling a table.**

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Inserting line breaks to make this easier to read yields the following[1].

```
a = [[1, 2, 3],
[4, 5, 6],
[7, 8, 9]]
```

Display all of list a

```
print(a)
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Display the first element of a (the element is a list).
print(a[0])
[1, 2, 3]
```

Display index 1 of the first element of a.

```
print(a[0][1])
2
```

When tabular data is represented as a list of lists, referencing every element can be achieved through nested for statements. There are multiple ways to do this.

6.3.1 Using indices

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
sum = 0
for i in range(len(a)):
    for j in range(len(a[i])):
        sum += a[i][j]
print(sum)
```

yields the following:

```
45
```

6.3.2 Handling the list directly with for statements

For this example, we will only be referencing the values of the elements, so the code can be written in the manner shown below. The target variable for the for statement is the list itself this time rather

[1] Usually in Python, statements cannot span multiple lines. This is the case in Python 3, but you can clearly see that the example above spans multiple lines. If open brackets such as (do are not closed, then line breaks will not result in an error. In cases where brackets are not used, a \ is required to achieve the same effect.

than a number, so “row” and “element” are used for ease of understanding.

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
sum = 0
for row in a:
    for element in row:
        sum += element
print(sum)
```

6.3.3 Creating a table of Kyoto intersections

Exercise 6-1 Creating a table of Kyoto intersections

Use Program 6-1 as a reference to create the table below as a list of lists.

```
cross_table =[[″SanjyoKawaramachi″, ″SanjyoKarasuma″, ″SanjyoHorikawa″],
    [″ShijyoKawaramachi″, ″ShijyoKarasuma″, ″ShijyoHorikawa″],
    [″GojyoKawaramachi″, ″GojyoKarasuma″, ″GojyoHorikawa″]]
```

This program can be written in a few different ways. For example:

- Assign the null string "" to each element beforehand, create a list of lists containing the necessary number of elements, and assign each intersection name to an element.

Program 6-2 (p6-2_en.py)

Row	Source code	Explanation
1	`tozai␣=␣[″Sanjyo″,␣″Shijyo″,␣″Gojyo″]`	
2	`nanboku␣=␣[″Horikawa″,␣″Karasuma″,␣″Kawaramachi″]`	
3	`cross_table␣=␣[[″″,″″,″″],␣[″″,␣″″,␣″″],␣[″″,␣″″,␣″″]]`	Creates a list of the required size
4	`for␣i␣in␣range(len(tozai)):`	
5	`␣␣␣␣for␣j␣in␣range(len(nanboku)):`	
6	`␣␣␣␣␣␣␣␣cross␣=␣tozai[i]+nanboku[j]`	Generates intersection names
7	`␣␣␣␣␣␣␣␣cross_table[i][j]␣=␣cross`	Assigns intersection names

After running this, you can look at cross_table and see that the table was constructed correctly.

```
>>> cross_table
[[’SanjyoHorikawa’, ’SanjyoKarasuma’, ’SanjyoKawaramachi’],
[’ShijyoHorikawa’, ’ShijyoKarasuma’, ’ShijyoKawaramachi’], [’GojyoHorikawa’,
’GojyoKarasuma’, ’GojyoKawaramachi’]]
```

As the table in the program above is small, it can be directly defined in line 3. For larger tables, you can use something similar to the code below. This example creates a table with 5 rows and 4 columns.

```
cross_table = []
for i in range(5):
    row = []
    for j in range(4):
      row.append(””)
    cross_table.append(row)
```

- Set cross_table to be an empty list and then create a list of intersections for each east-west street. Finally, add (append) this to cross_table.

Program 6-3 (p6-3_en.py)

Row	Source code	Explanation
1	tozai␣=␣[”Sanjyo”,␣”Shijyo”,␣”Gojyo”]	
2	nanboku␣=␣[”Horikawa”,␣”Karasuma”,␣”Kawaramachi”]	
3	cross_table␣=␣[]	
4	for␣i␣in␣range(len(tozai)):	
5	␣␣␣␣street␣=␣[]	List of streets
6	␣␣␣␣for␣j␣in␣range(len(nanboku)):	
7	␣␣␣␣␣␣␣␣cross␣=␣tozai[i]+nanboku[j]	Generates intersection names
8	␣␣␣␣␣␣␣␣street.append(cross)	Appends intersection to streets
9	␣␣␣␣cross_table.append(street)	Appends list of streets to table

6.3.4 Displaying the table of intersections

1) Outputting lists

Let's output the list with commas (a half-width comma and a space) inserted between intersections that start with the same east-west street, but with no commas at the end of each line. This would look like so:

```
SanjyoKawaramachi, SanjyoKarasuma, SanjyoHorikawa
```

The print function defaults to ending with a line break, but you can specify the **end option** to change the string that displays at the end.

```
street = [”SanjyoKawaramachi”, ”SanjyoKarasuma”, ”SanjyoHorikawa”]
```

```
for i in range(len(street)):
    if i < len(street)-1:
        print(street[i], end=", ")  # Adds ", " to all but the final of the line
    else:
        print(street[i]) # Line break at the end
```

While still a bit advanced at this point, you can also pass the entire list to the print function all at once using "*" and specify the delimiter character with the sep option.

```
street = ["SanjyoKawaramachi", "SanjyoKarasuma", "SanjyoHorikawa"]
print(*street, sep=", ")
```

Exercise 6-2 Outputting lists of lists

Create a program that outputs the list of lists of intersection names created in the example above in a manner that conforms to the following conditions:

- Output the intersection names for one east-west street on one line.
- Insert ", " (a half-width comma and a space) in between the intersection names starting with the same east-west street. However, do not put a comma at the end of each line.

Specifically, the program must yield the following output:

```
SanjyoKawaramachi, SanjyoKarasuma, SanjyoHorikawa
ShijyoKawaramachi, ShijyoKarasuma, ShijyoHorikawa
GojyoKawaramachi, GojyoKarasuma, GojyoHorikawa
```

7. Encapsulation of Processing Using Functions

7.1 Learning goals of this chapter

1. In this chapter, we will use the exercise from Chapter 5 to learn how to take a batch of commands and define it as a function so that it can be used when needed.
2. Learn how to pass values as arguments to functions.
3. Learn how to return a value from a function when it is called.
4. Learn about variables used within functions and the scope within which they have influence.
5. Learn about some general situations in which functions tend to be used.

As with Chapter 5, there is a lot to learn in this chapter. However, you will be able to use the material covered in this chapter in concrete examples throughout the rest of the text. As such, you don't need to worry about learning all the fine details right away. Simply **focus on learning what you can do with the concepts** learned in this chapter **as well as on building experience writing code using these concepts.** It is always possible to look back through the text to review the fine details as the need arises.

7.2 Let's make an absolute value function

As a simple exercise in defining functions, let's create our own absolute value function like the one we used in a previous chapter.

Program 7-1 Example of an absolute value function (p7-1.py)

Row	Source code	Notes
1	def␣myabs(x):	Definition of the myabs function, the
2	␣␣␣␣if␣x<0:	argument is x
3	␣␣␣␣␣␣␣␣x␣=␣-x	Flips the sign if x is negative
4	␣␣␣␣return␣x	Returns x
5		
6	while␣True:	This is the main program (the part that runs
7	␣␣␣␣a␣=␣float(input(”>␣”))	first when the program is executed)
8	␣␣␣␣print(a,␣myabs(a))	Calls myabs with a as the argument

Program 7-2 Example of an absolute value function (p7-2.py, returns when possible)

Row	Source code	Notes
1	def␣myabs(x):	Definition of the myabs function, the argument
2	␣␣␣␣if␣x<0:	is x

3	`␣␣␣␣␣␣␣␣return␣-x`	if x is negative, flip the sign and return
4	`␣␣␣␣return␣x`	Returns x
5		
6	`while␣True:`	This is the main program (the part that runs
7	`␣␣␣␣a␣=␣float(input(">␣"))`	first when the program is executed)
8	`␣␣␣␣print(a,␣myabs(a))`	Calls myabs with a as the argument

Exercise 7-1 Create a absolute value function

Write Program 7-1 and Program 7-2 and confirm that they behave as expected.

Exercise 7-2 Experience errors (5).

In Program 7-1 and Program 7-2, function "myabs" assumes that the given parameter is numeral. If you forget calling float() function in line 7 and write as "a = input("> "), the input data is assigned to variable a as string. Examine what will happen in this case. See also "17 How to Read Error Messages in IDLE/Python."

7.3 Format for function definitions

The general format for function definitions is given below.

```
def function name(parameters):
    Explanatory string
    Block executed by the function
    return value returned by the function
```

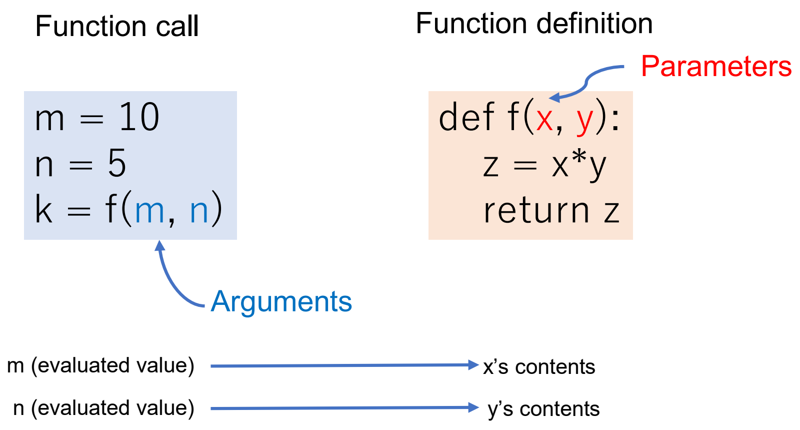
- Function names are governed by the same rules as variable names (referred to as identifiers). As functions tend to be accompanied by specific actions, verbs are often chosen as function names.
- Parameters are the values that the function receives. They are written using the variable names (parameters) with which they will be referred to within the function. When multiple parameters are specified, they must be separated using “,”[1].
- Even if there are no parameters, the () after the function name cannot be omitted. The () are also necessary when calling a function with no parameters.
- The explanatory string (it is called a docstring) can be omitted. Reference the column titled "Documenting a Program."
- return statements are generally written at the end of the function definition.
- return statements can be included anywhere and are not limited to the block of an if statement that checks for certain conditions or to the end of the definition.
- If the function does not need a return value, you do not need to include a return statement.

[1] There are other ways to treat variables in Python that will be omitted here.

7.4 Parameters and arguments

The parameters where the function is actually called are **arguments or actual parameters**, and the parameter used internally within the function are also called **formal arguments**. Reference the column titled " Parameters and Arguments."

- Arguments can be variables, equations, and function calls. The equation or function will be evaluated first before the result is passed to the function.
- Arguments and parameters do not need to have the same names.
- If a number, string, etc. is passed to a function as a argument, the value of the corresponding parameter will not be affected even if another value is assigned to the parameter within the function[1].



```
m = 10
n = 5
k = f(m, n)
```

```
def f(x, y):
    z = x*y
    return z
```

Figure 7-1 Arguments and Parameters

7.5 Return values

The calculated result of a function is returned using a return statement.

- As seen in Program 7-2, it is possible to include multiple return statements within a function definition. However, splitting the return among multiple possible statements can make editing the function difficult.
- **In Python, it is possible to return multiple values by separating them with commas.** When calling the function, multiple variables can be listed, separated by commas, to accept multiple return values. If multiple return values are accepted with a single variable, this variable will contain multiple return values and become a data type known as a tuple.

[1] Lists can be overwritten.

7.6 From an exercise in a prior chapter

In a prior chapter, you learned how to write a program to solve for square roots as well as how to accept the number whose root it would solve as input from the terminal. Combining these concepts, we dealt with the problem of writing a program that would continually ask for numbers from the terminal and solve for the square root of the inputted numbers. Two potential ways to write a program that continually asks for input from the terminal are explained below.

1. Receive input, calculate the square root only when the proper input is received, and repeat by asking for more input.
2. Continually repeat receiving proper input from the terminal followed by solving for the square root of the input.

If you were to program both of these, 1 would need to solve for the square root, and 2 would need to obtain proper input from the terminal before solving for the square root. If you were to assume these actions were performed by the functions get_positive_numeral() and square_root(), the programs could be written as described below.

1.
```
while True:
    Obtain input x from the terminal and check
    whether it is a positive number
    if positive number:
        r = square_root(x)
        print(r)
```

2.
```
while True:
    x = get_positive_numeral()
    r = square_root(x)
    print(r)
```

Writing a fixed batch of processing as a function like this allows you to **encapsulate** said processing. The advantages of doing this include:

- The program that calls the function becomes much shorter and easier to understand.
- The same processing can easily be performed in different parts of the program.
- Modifying the definition of the function becomes easier when you separate the program where it is used from the program where it is defined.

7.7 Let’s write the square_root() function

Let’s take p5-6.py, which solves for square roots, and turn it into the square_root() function.

Program 7-3 Implementation of the square_root() function (p7-3.py)

Row	Source code	Explanation
1	#	
2	def␣square_root(x):	A function that takes a parameter x
3	␣␣␣␣'Calculate square root of argument x'	explanatory string (docstring)
		The function definition block extends
4	␣␣␣␣rnew␣=␣x	until line 17. As with p5-6.py, be
5	␣␣␣␣#	careful of the indentation.
6	␣␣␣␣diff␣=␣rnew␣-␣x/rnew	
7	␣␣␣␣if␣diff␣<␣0:	
8	␣␣␣␣␣␣␣␣diff␣=␣-diff	
9	␣␣␣␣while␣(diff␣>␣1.0E-6):	
10	␣␣␣␣␣␣␣␣r1␣=␣rnew	
11	␣␣␣␣␣␣␣␣r2␣=␣x/r1	
12	␣␣␣␣␣␣␣␣rnew␣=␣(r1␣+␣r2)/2	
13	␣␣␣␣␣␣␣␣print(r1,␣rnew,␣r2)	
14	␣␣␣␣␣␣␣␣diff␣=␣r1␣-␣r2	
15	␣␣␣␣␣␣␣␣if␣diff␣<␣0:	
16	␣␣␣␣␣␣␣␣␣␣␣␣diff␣=␣-diff	'return rnew,' returns the calculated
17	␣␣␣␣return␣rnew	value
18	#␣Main␣program␣from␣here	
19	v␣=␣2	
20	r␣=␣square_root(v)	
21	print("Result is␣",␣r)	

The result of running this program is shown below.

If you include an explanatory string (docstring) with a function definition in Python, you can display this string by calling the help function and passing the name of the function as the argument.

```
2 1.5 1.0
1.5 1.4166666666666665 1.3333333333333333
1.4166666666666665 1.4142156862745097 1.411764705882353
1.4142156862745097 1.4142135623746899 1.41421143847487
1.4142135623746899 1.414213562373095 1.4142135623715002
Result is  1.414213562373095
>>> help(square_root)
Help on function square_root in module __main__:

square_root(x)
    Calculate square root of argument x

>>>
```

Exercise 7-3 Create a function to solve squate root

Make a program that continually solves for square roots with the function square_root().

Exercise 7-4 Create a function to solve squate root (2nd)

Define the function get_positive_numeral(), and make program that continually solves for square roots with it as well as the function squate_root().

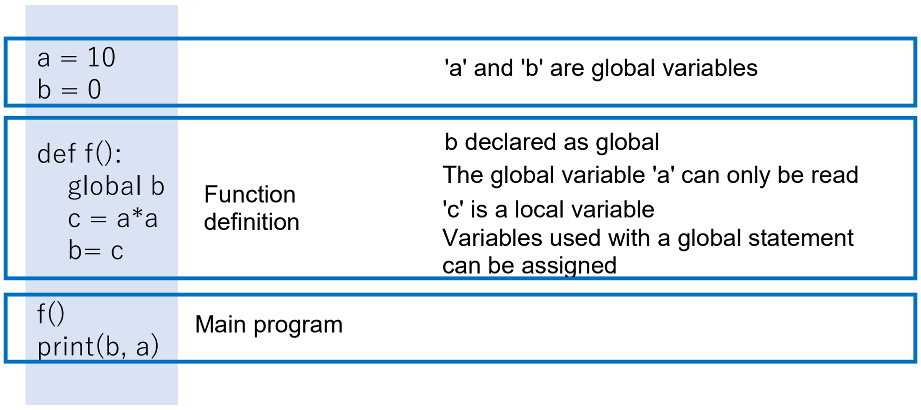
7.8 Treatment of variables within functions

Python treats variables within functions in the following manner. Reference the column titled " Scope of Variables."

- **Local variables**: Variables defined (assigned) within functions can only be used within those functions. Every time the function is run, these variables will be lost when the function ends.
- **Parameters** are treated as local variables.
- **Global variables**: Variables defined outside of the function can be read to obtain their values.
- **Assignments to global variables**: Within functions, only global variables that are declared with 'global' statement can be assigned values.

Because variables are treated in this manner in Python:

- Variables only temporarily needed within functions can be used freely without worrying about their influence on anything else.
- Operating on global variables can be a primary cause of longer programs being difficult to understand. In Python, reading variables, which is relatively safe, is permitted unconditionally. On the other hand, writing to variables requires an explicit global statement. In this way, Python strikes a good balance between convenience and safety.



```
a = 10
b = 0

def f():
    global b
    c = a*a
    b= c

f()
print(b, a)
```

'a' and 'b' are global variables
Function definition
b declared as global
The global variable 'a' can only be read
'c' is a local variable
Variables used with a global statement can be assigned
Main program

Figure 7-2 Global variables and local variables

7.9 Common uses of functions

Unlike mathematical functions, there are multiple ways in which Python functions are commonly used. The information supplied to the function within () is called the argument, and the value returned by the function is called the return value.

- Pass an argument and use the return value. Similar to how mathematical functions are used.

  ```
  y = math.fabs(-2.0)
  ```

- Pass no arguments and do not use a return value. Used to execute a fixed set of commands.
  For example, turtle's up() function seen in the next chapter
- Pass arguments but do not use a return value. Used to execute a set of statements that contains changing values.
  For example, turtle's forward(100) function seen in the next chapter
- Pass no arguments and use a return value. Used to gain information about the state of something.
  For example, turtle's p = pos() seen in the next chapter

There are also functions that **read and write to global variables** as well as functions that **exchange information using lists and other rewriteable arguments**. However, these uses for functions are often referred to as function **side effects**. Because they are often difficult to explicitly state within the source code (particularly the side that calls the function), they have the downside of making programs more difficult to understand.

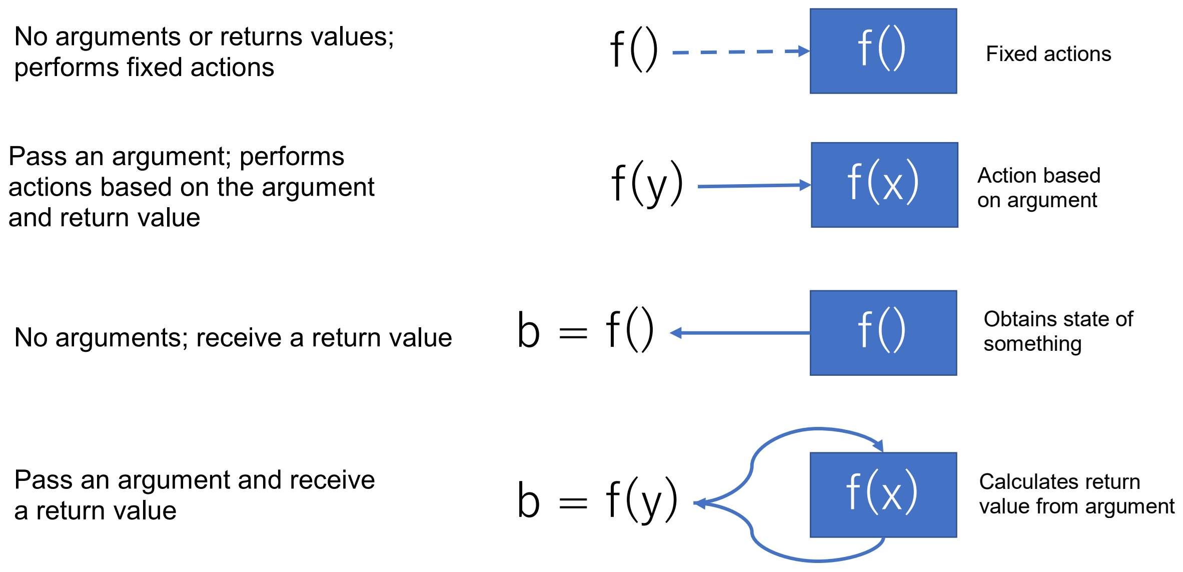
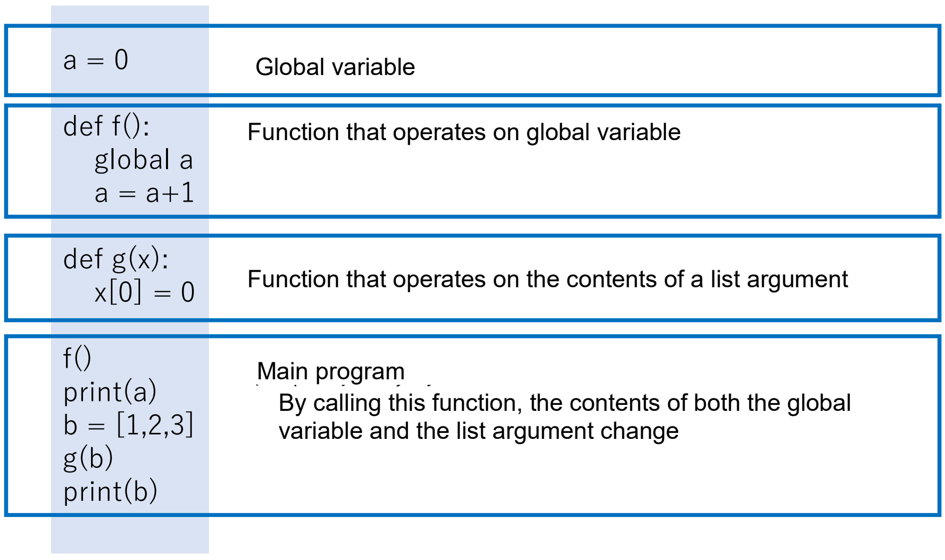

No arguments or returns values; performs fixed actions
f()
f()
Fixed actions
Pass an argument; performs actions based on the argument and return value
f(y)
f(x)
Action based on argument
No arguments; receive a return value
b = f()
f()
Obtains state of something
Pass an argument and receive a return value
b = f(y)
f(x)
Calculates return value from argument

Figure 7-3 Common uses of functions



```
a = 0
def f():
    global a
    a = a+1
def g(x):
    x[0] = 0
f()
print(a)
b = [1,2,3]
g(b)
print(b)
```

Global variable
Function that operates on global variable
Function that operates on the contents of a list argument
Main program
By calling this function, the contents of both the global variable and the list argument change

Figure 7-4 Function calls and "side effects"

7.10 Function calls and passing function objects

When executing a function defined by a def statement, you write something like f() regardless of whether or not the function has arguments. Regarding this, in later chapters, you will run into the function that runs when the mouse is clicked in turtle graphics, or the function that runs when a button is pressed in a GUI program within tkinter in which solely the function name is written. Using this notation does not run the function on the spot, rather it enables you to pass the function in its entirety as an object that can be executed later. In the example below, the function f is passed as an argument to the function F, and f runs within F.

```
def f():
    print("f says Hello")
# Function that takes a function as an parameter and runs it
def F(y):
     print("In F, ", end="")
     y()
# runs f
f()
f says Hello
# passes f to F and runs F
F(f)
In F, f says Hello
```

7.11 Default parameters and keyword parameters

7.11.1 Default parameters

Within function definitions in Python, it is possible to specify a value to be implicitly used (default value) in the case that an argument is not passed. This can be done by setting the argument equal to something (using "=") within the function definition.

7.11.2 Keyword parameters

When calling a function, arguments are written in order from left to right. However, by writing "argument name = value", you can pass a value for only the specified parameter.

7.11.3 Example

```
def f(a, b=2, c=3):
    return a + b + c


f(1,1,1)
3
f(1)
6
f(1, c=2)
5
```

When programming with tkinter in later chapters, there will be a large number of parameters. Thus, we will be using this method of employing keyword parameters to specify only the necessary ones.

8. Playing with Turtle

8.1 Learning goals of this chapter

1. Learn how to use modules in Python with Turtle.
2. Learn how to use class objects in Python with Turtle
3. Review what you have learned so far through making graphical works with Turtle. Independently learn how to use the libraries necessary to make these graphical works.

8.2 Modules

Python offers a variety of libraries in the form of "modules." To use a module, you must first import the module in the Python script. There are a few ways you can do this, as seen below.

8.2.1 Where to put import statements

Import statements are normally all listed at the beginning of a program.

8.2.2 Importing a module by specifying its name

The math module provides access to mathematical functions. To import it, we write:

```
import math
```

In order to call functions and constants from the module, first write the module name, followed by a dot. For example, to call the "pi" constant, we write:

```
math.pi
```

All modules typically follow this style.

8.2.3 Importing a module using a custom name

An alias is used when you want to refer to a module name in a shortened form. For example, the module tkinter, which will be discussed later, is often abbreviated to tk, as follows:

```
import tkinter as tk
```

Then, you can use the alias tk to refer to the module in your code.

8.2.4 Importing elements from within a module

We will import functions from the turtle module used in this chapter simply by writing:

```
from turtle import *
```

This allows us to use all the functions in a module just by calling the function name. However, it is dangerous to overuse this method, because you may lose track of which module the function or variable belongs to. Reference the column titled " Namespaces."

8.2.5 Be careful not to name programs with the same file name as a module

When Python reads the import command, it searches beforehand in the folder containing the libraries and other files for Python programs with the specified module name.

When this occurs, the current working folder is searched first, so if you create a file called turtle.py, for example, it will be interpreted as a module file and an error will occur. Be careful not to create a Python script in the working folder with the same name as a module.

8.3 Turtle — The time-honored turtle

Turtle graphics allows you to give commands to a turtle (robot) on the screen to move forward, rotate, and such, as well as create a graphical representation of its trajectory.

LOGO, a programming language developed at the Massachusetts Institute of Technology (MIT), has integrated graphics functionality which allows for the visualization of the movement of programs for learning. One of the developers of LOGO, Seymour Papert (1928-2016), is famous for his efforts in teaching children programming. He is quoted as saying:

> In many schools today, the phrase "computer-aided instruction" means making the computer teach the child. One might say the computer is being used to **program the child**. In my vision, the **child programs the computer** and, in doing so, both acquires a sense of mastery over a piece of the most modern and powerful technology and establishes an intimate contact with some of the deepest ideas from science, from mathematics, and from the art of intellectual model building.
>
> - Seymour Papert [10], emphasis in bold by the author.

In Japan, a teacher named Mr. Totsuka created a program to process LOGO by himself, and taught it in elementary schools[11]. Other child-friendly programming languages include Squeak (developed by Alan Kay), and Scratch [1], which was developed at MIT. Squeak and Scratch also have turtle graphics functionality. The mascot of Scratch, which is a cat that you can make run around using programming, is descended from turtle graphics. You can also use the Turtle module in Python to have fun with turtle graphics.

Dealing only with numbers can get tiresome, so in this chapter, we will review what you have learned up until now using turtle graphics.

8.4 Python's turtle module

- In Python, turtle graphics is provided as the turtle module.

[1]The MIT Media Lab Group that developed Scratch is called "lifelong kindergarten." Isn't that a wonderful name?

Next Chapter Table of Contents

- It is based on the tkinter GUI environment[1].
- It can be used in two ways: procedurally, to manipulate a single turtle with function calls, and object-oriented, to handle multiple turtles.
- **Note: Do not create Python programs with the same file name as the module name (turtle.py), or Python will not be able to find the correct module.**

8.5 Let's try it out

Write the program in the table below and try running it.

Program 8-1 turtle usage example
(p8-1.py, Do not save this program as turtle.py)

Row	Source code	Explanation
1	from␣turtle␣import␣*	Calls the functions defined in the turtle module.
2	forward(100)	
3	left(90)	
4	forward(100)	
5	left(90)	
6	forward(100)	
7	left(90)	
8	forward(100)	
9	left(90)	
10	done()	End

The forward() function makes the turtle move forward, and the left() function makes the turtle move left.

The turtle holds a pen, and when the pen is pressed down (which is the default setting), it marks the path that the turtle takes when it moves.

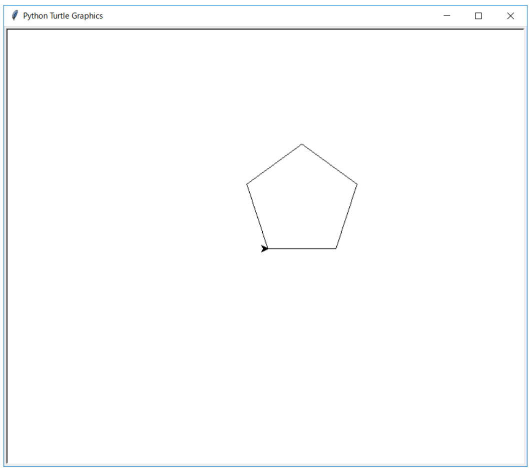
Exercise 8-1 Draw a regular polygon with n faces

Complete the following program so that it draws a regular polygon with n faces.

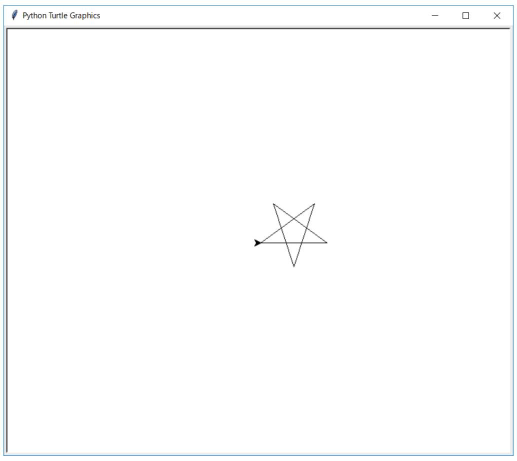
[1]IDLE, which is used in this course, and the GUI environment introduced below are similar. Tkinter uses the Tcl/Tk GUI library.

Program 8-2 Program that draws polygons with n faces (incomplete, to be saved as p8-2.py)

Row	Source code	Explanation
1	`from␣turtle␣import␣*`	Calls the functions defined in the turtle module.
2	`n␣=␣5`	Draw a regular pentagon
3	`for␣i␣in␣range(n):`	Repeat n times
	`done()`	End



Exercise 8-2 How can you draw a 5-pointed star?
Hint: for regular polygons, the direction the turtle was facing changed so that after completing its circuit, it had made a single revolution. What is needed to draw a 5-pointed star?



Next Chapter Table of Contents

Exercise 8-3 Draw regular 7- and 9-sided polygons, and their star equivalents

Make the turtle draw a regular 7-sided polygon, regular 9-sided polygon, and their star equivalents (where the shape made inside of the star has the same number of vertices as the regular polygon equivalent).

8.6 The major functions in the turtle module

The following functions are available. For more details, see
24.1. turtle — Turtle graphics
in the Python documentation.

- forward(d): Move the turtle forward by the specified distance *d*. fd(d) is the same function.
- back(d), bk(d), backward (d):Move the turtle backward
- right(a), rt(a): Turn the turtle right by *a* degrees
- left(a), lt(a) :Turn the turtle left by *a* degrees
- goto(x, y), setpos(x, y), setposition(x, y): Move the turtle to the coordinates *x*, *y*
- setheading(a): Set the orientation of the turtle to *a* degrees
- pendown(), pd(), down(): Pull the pen down (draw when moving)
- penup(), pu(), up(): Pull the pen up
- position(), pos(): Return the turtle's current location as a two-dimensional vector. Two variables are output as return values, as shown below.

```
x, y = pos()
```

- heading(): Return the turtle's current orientation
- isdown(): Returns True if the pen is down, False if the pen is up.

8.7 Moving multiple turtles

8.7.1 Example program

Program 8-3 Moving Multiple Turtles (p8-3.py)

Row	Source code	Explanation
1	`from␣turtle␣import␣*`	Calls the functions defined in the turtle module.
2	`t1␣=␣Turtle()`	Creates the first turtle t1
3	`t2␣=␣Turtle()`	Creates the second turtle t2
4	`t1.color('red')`	Sets the color of t1 to red
5	`t2.color('blue')`	Sets the color of t2 to blue
6	`for␣i␣in␣range(180):`	
7	`␣␣␣␣t1.forward(5)`	Moves t1 forward 5 steps

8	`␣␣␣␣t2.forward(3)`	Moves t2 forward 3 steps
9	`␣␣␣␣t1.left(2)`	Turns each turtle left twice
10	`␣␣␣␣t2.left(2)`	
11	`done()`	End

8.7.2 Using Class Objects

In this example, each turtle is created as an object of the Turtle class.

A turtle has a number of attributes, such as “location” (coordinates), “direction,” “pen up” or “pen down,” and “pen color.” In order to program a turtle, you need to know these attributes and be able to change them.

Objects with a class type are a good way to describe such operations. By using multiple turtles, you will be able to understand the key points of using class objects.

1) 1) Creating a turtle

Objects of the Turtle class are created with the Turtle() command. The function that creates a class object is called a constructor, which in this case creates a turtle (a construct)[1]. In the example below, by assigning the generated object to the variable t1, the turtle can be referred to by the variable t1 thereafter.

```
t1 = Turtle()
```

2) 2) Manipulating the Turtle

The way to know the state of the turtle, or to change its state, is to call a “method.” Methods are invoked by writing “.” followed by the method name (and arguments) of the object variable. Think of it like a function call, except that an object is the target.

```
t1.forward(10)
```

3) 3) Determining the Turtle's Attributes

To find out an attribute of the turtle, call a method that returns the attribute, and assign the result to an appropriate variable.

```
x, y = t1.pos()
```

This can also be written as follows

```
(x, y) = t1.pos()
```

[1] In computer programming, you will often encounter terms with suffixes such as -er, because personifying expressions are useful for expressing the things you ask a computer to do. The word “computer” also used to mean a person in charge of calculations (a calculator) until the advent of mechanical calculators.

8.8 Tips for creating your project

8.8.1 Responding to Mouse Clicks

Program 8-4 Responding to Mouse Clicks in Turtle Graphics (p8-4.py)

Define a function to be executed when the mouse is clicked, and pass the defined function object to the onscreenclick() function.

Row	Source code	Explanation
1	`from␣turtle␣import␣*`	
2	`def␣come(x,y):`	Defines a function to be executed when
3	`␣␣␣␣(xx,yy)␣=␣pos()`	the mouse is clicked, with the parameters
4	`␣␣␣␣newxy␣=␣((xx+x)/2,(yy+y)/2)`	being the position of the mouse cursor
5	`␣␣␣␣print(x,y)`	when clicked.
6	`␣␣␣␣goto(newxy)`	The function definition ends here.
7	`onscreenclick(come)`	Set up the function to be called when the
8	`done()`	mouse is clicked (note that there is no
		() after ”come”).

8.8.2 Converting Coordinates to Angles

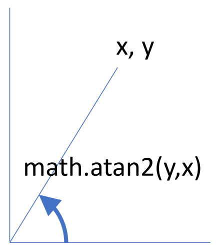
To find the angle from the x and y coordinates in a given direction, Python provides a function called atan2 in the math module[1]. The argument is the previous vertical coordinate. The return value is in radians, so to use it with turtle, where the angle is set in degrees, we convert it as follows.

```
import math

y = 2

x = 1

angle = math.atan2(y, x)*180/math.pi
```



[1] To calculate coordinates from angles, you can use the trigonometric functions cos and sin, but since these are not suitable as inverse trigonometric functions, atan2 has been implemented as an extension of the inverse tangent function (atan).

8.8.3 Using Random Numbers

Random motion is another interesting thing to visualize.

- To use random numbers in Python, use the random module.
- Refer to the column titled "Random Numbers."
- Here is an example of turtle graphics using random numbers (see random_turtle.py for more details).
- The turtle is stopped by a mouse click.

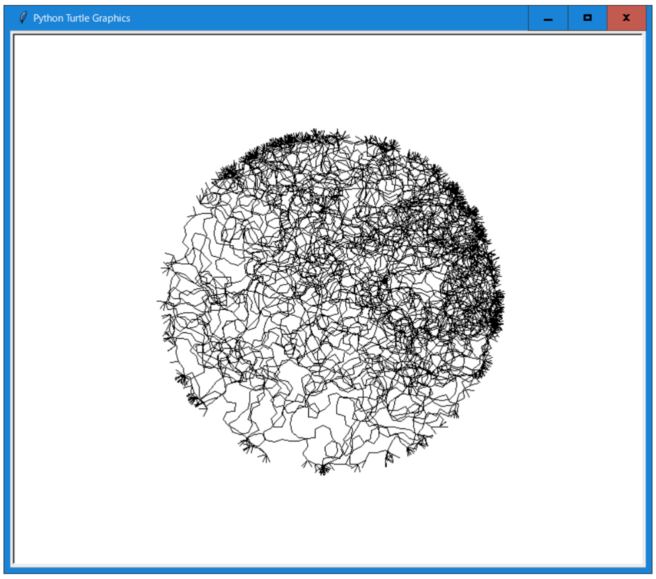


Figure 8-1 A result of executing random_turtle.py

8.8.4 Drawing Fractals

A shape in which one part is similar to the whole is called a fractal. Fractal figures are interesting both for what they are drawn as and for the algorithms that draw them.

- Use recursion (calling a function within the function itself).
- See the column titled "Recursion."
- See detour.py and turtle-tree.py. The recursive calls are marked in red.

Next Chapter Table of Contents

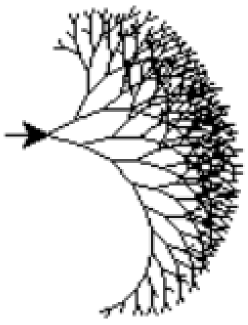
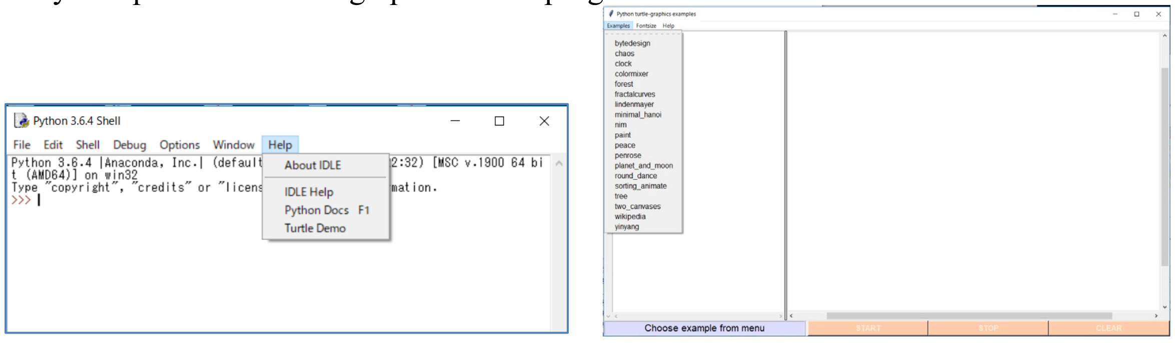


Figure 8-2 A result of executing turtle-tree.py

8.9 Turtle Demo

Python provides a turtle graphics demo program that can be called from the IDLE menu.



```
Python 3.6.4 |Anaconda, Inc.| (default                    2:32) [MSC v.1900 64 bi
t (AMD64)] on win32
Type "copyright", "credits" or "licens                    mation.
>>> 
```

Figure 8-3 How to run the Turtle Demo

8.10 Theme: Creating projects with turtle

To test your skills so far, please create a project using Turtle.

- You will submit the program, screenshots, and your notes.
- The program should be programmed by you.
  - However, it is okay to ask for advice. Please include an acknowledgement of the advice you receive in your notes.
  - Submit your project as an attachment.
- The following information should be included in the memo.
  - Name, affiliation
  - Description of your project

- The functions of Python and turtle graphics that you have learned.
- Bibliographic information about any information you referred to, if applicable, and if it is a website, the title, URL and access date of the website
- If there is someone who helped you, write the name of the person who helped you and a description of the help you received in the acknowledgements.
- This should be written in Word etc. and submitted as a PDF file.

8.11 How to take a screenshot

- To obtain a screenshot of a specific window in Windows, follow the steps below.
  - Select the window.
  - Hold down the ALT key and press the PRTSC key.
    If you press only the PRTSC key, a screenshot of the entire screen will be taken.
  - This will copy the screenshot to the clipboard.
  - You can then save the screenshot by pasting the contents of the clipboard into a program that can manipulate images, such as Paint.
  - See [16] for more details about screenshots on a Mac.

Table 8-1 Key Operation to Take a Schreen Shot

	The entire desktop	A specific window	Location of the shot
Windows	Print Screen key	Alt + Print Screen key	Clipboard (with copy)
	Win + Print Screen key	Win + Alt + Print Screen key	Save the file in PNG format in the Picture Screenshot folder/Video Capture folder
Mac	Command + shift + 3	Command + shift + 4 + space	Save to Desktop as PNG

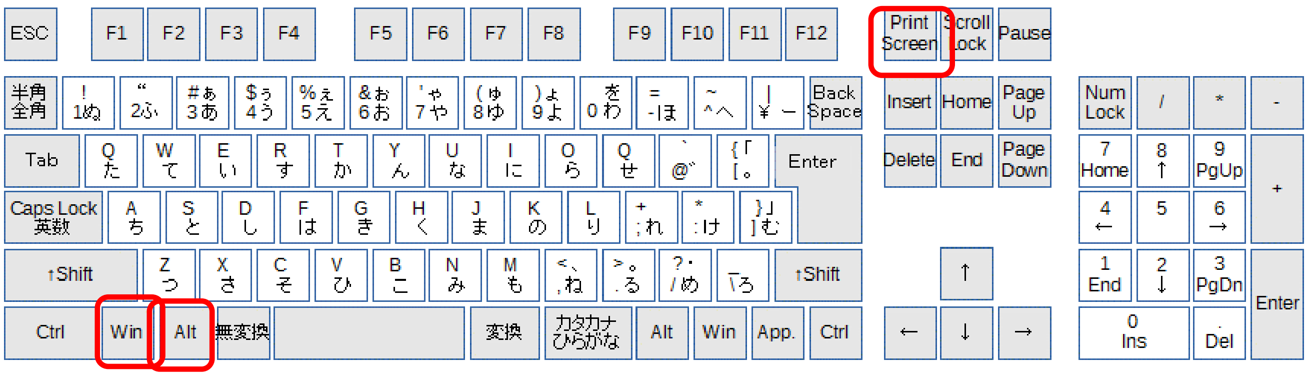


Figure 8-4 Keys used to take a schreen shot in Windows

References

[14] Seymour A. Papert: Mindstorms: Children, Computers, And Powerful Ideas, Basic Books (1993)
[15] 戸塚滝登著: コンピュータ教育の銀河, 晩成書房 (1995, in Japanese)
[16] Taking a screenshot on a Mac, https://support.apple.com/ja-jp/HT201361(accessed May 7, 2021)

Program 8-5 random_turtle.py

```
from␣turtle␣import␣*
import␣random
#␣Import␣random␣module␣as␣you␣will␣be␣using␣random␣numbers

#␣Variable␣(flag)␣to␣stop␣execution
stop_flag␣=␣False

#␣Function␣called␣when␣the␣mouse␣is␣clicked;␣takes␣x,␣y␣parameters
#␣We␣need␣this␣here,␣but␣we␣will␣not␣use␣it
#␣Set␣the␣stop␣execution␣flag␣to␣True

def␣clicked(x,y):
␣␣␣␣global␣stop_flag
␣␣␣␣stop_flag␣=␣True

#
#␣Specify␣what␣to␣do␣when␣the␣mouse␣is␣clicked;
# here, call␣the␣clicked␣function.
#
onscreenclick(clicked)

speed(0)
while(not␣stop_flag):
␣␣␣␣#␣Randomly␣change␣the␣orientation␣in␣the␣range␣of␣-90␣to␣90␣degrees
␣␣␣␣left(random.randint(-90,90))
␣␣␣␣forward(10)
␣␣␣␣#␣If␣the␣turtle's␣position␣is␣more␣than␣a␣certain␣distance␣from␣the␣origin,␣go␣back.
␣␣␣␣if␣position()[0]**2+position()[1]**2␣>␣200**2:
␣␣␣␣␣␣␣␣forward(-10)
```

Next Chapter Table of Contents

Program 8-6 detour.py

```
from turtle import *
def detour(L):
    if L < 10:
        forward(L)
    else:
        LL = L/3
        detour(LL)
        left(60)
        detour(LL)
        right(120)
        detour(LL)
        left(60)
        detour(LL)

for i in range(6):
    detour(100)
    left(60)
```

Program 8-7 turtle-tree.py

```python
from turtle import *

# Recursively draw a tree
def tree(n):
    # If parameter is less than or equal to 1, go forward 5 steps.
    if n<=1:
        forward(5)
    else:
        # If the parameter is greater than 1,
        # move (the trunk) forward according to the value of the parameter
        forward(5*(1.1**n))
        # Record the current position and orientation
        xx = pos()
        h = heading()
        # Rotate 30 degrees to the left
        left(30)
        # Draw a tree of size n-2 (left branch)
        tree(n-2)
        # Raise the pen to stop drawing the trajectory
        up()
        # Return to the previously recorded position (top of the trunk)
        setpos(xx)
        setheading(h)
        # Pull the pen down
        down()
        # Go 15 degrees to the right
        right(15)
        # Draw a tree with size n-1 (right branch)
        tree(n-1)
        # Pull pen up and go back
        up()
        setpos(xx)
        setheading(h)
        # Pull the pen down
        down()

# Use the fastest drawing speed, as it takes a long time
speed(0)

# Draw a tree of size 12
tree(12)
```

9. Creating a GUI Application with Tkinter (1)

9.1 Learning goals of this chapter

In this chapter, you will create a GUI application using tkinter, and learn:

- The role of frameworks in GUI applications, and learn about event-driven programming.
- The MVC architecture in GUI applications.
- To understand how to define functions in Python by implementing callback functions in tkinter.

9.2 GUIs and event-driven programming

In GUI applications, the user selects and uses various operations using menus and buttons. They also expect the computer to respond appropriately to their actions.

- These user operations are called “**events**.”
- Many GUI applications use a GUI “framework.”
- The framework monitors the mouse and keyboard operations, detects events, and calls the event-handling program set up by the programmer.

In a GUI application that uses a framework, the programmer is mainly responsible for programming the following parts.

- Configuration of the screen where buttons, etc. are placed to be used as a GUI application
- Definition of the processes to be run when an event occurs

This type of programming is called **event-driven programming** because it mainly describes responses to events.

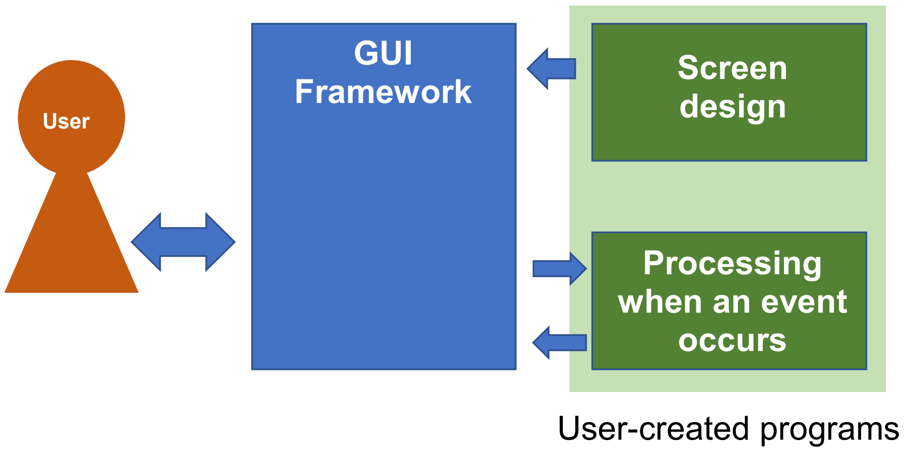


Figure 9-1 Framework for event-driven programming

9.3 Separating the model and user interface

Let's say you want to write a calculator program that handles addition, subtraction, multiplication, and division. Since addition, subtraction, multiplication, and division are operations on two numbers (binary operations), the user wants to delegate the task to the computer as follows:

- **The “first number,” “second number,”** and **“operation to be performed,”** are **“set,”** then
- the set operation is **“applied.”**
- The **“result of the application”** is **“obtained”**

The words written in blue are “things” that act as the targets of operations such as numbers and operations, and are indicated by “nouns (phrases).” On the other hand, the words written in red are “operations” and are indicated by verbs including verbal nouns. These constitute the “**model**” of the “calculator” task.

On the other hand, in order for people to interact with this model, you need a user interface that people can operate. The user interface of a calculator application on a personal computer or a smart phone is one example, as well as a Python shell where expressions can be entered using the keyboard. For visually impaired users, interfaces with audio and braille may be required.

If you focus on creating a calculator, you can say that the “model” is shared and only the user interface changes in various ways. In GUI programming, this concept is called an “MVC architecture.” Let’s look over the definitions of M, V, and C below. Reference the column titled "GUI."

- M: Model - a model to be computed that provides the framework of the application, essentially independent of the GUI.
- V: View - a program that shows the results obtained by the model to the user, which is handled by the GUI.

- C: Control, a program for user operations on the model, which is handled by the GUI.

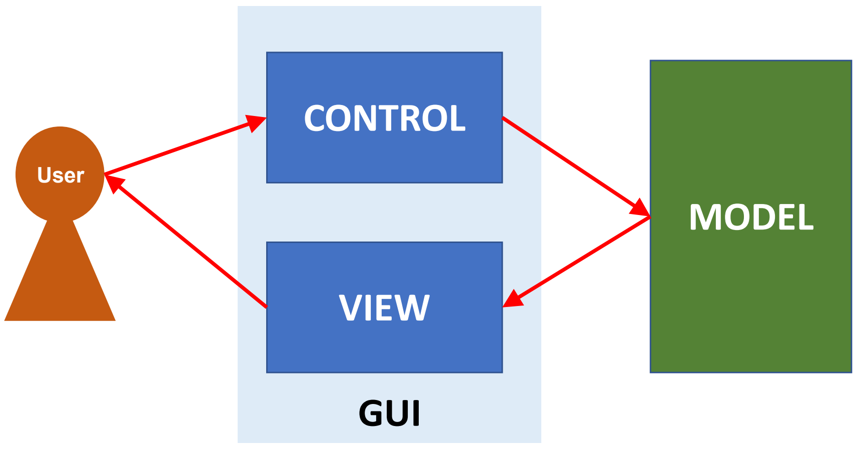


Figure 9-2 Model-View-Control Architecture

9.4 tkinter

There are several basic software and windowing environments for personal computers, such as Windows, macOS, Linux/X-Window. Each of them uses different methods for drawing windows. Tcl/Tk is a GUI framework that absorbs the differences between these OS and windowing environments so that it can be used across different platforms.

tkinter is a package that makes Tcl/Tk available in Python.

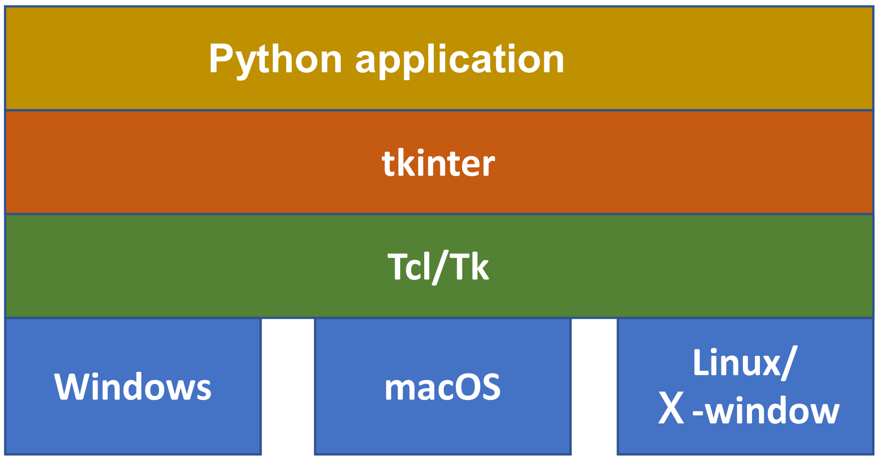


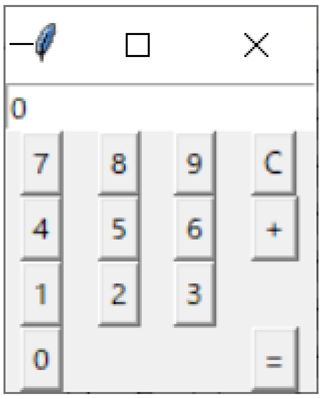
Figure 9-3 Tkinter system configuration

9.4.1 Terms in tkinter

- **Widget**: A generic term for a component such as a button that makes up a GUI.
- **Container**: A receptacle for widgets (groups of widgets).
- **Layout manager/geometry manager**: A mechanism to adjust the geometric arrangement of widgets.
- **Callback function**: A function that is called when a widget is manipulated to perform the necessary processing.

9.5 tkinter example (tkdemo-2term.py)

Consider the following addition-only calculator.



- It consists of 10 keys (0-9), 3 buttons (C (clear), + (add), = (calculate)),
- and a single line of character input/output for numerical values.
- The 0 to 9 buttons insert a digit at the least significant digit of the number being entered, just like a calculator. (Multiply the number entered thus far by 10 and add the number of the pressed key to it.)
- C sets the numerical value to 0.
- The + key registers the input number as the first term of a binary operation and then sets the input number to 0 again.
- The = key registers the second input number as the second term of the binary arithmetic operation, executes the addition, prints the result, and zeros the input number.

Program 9-1 is an example implementation of this.

Program 9-1 Calculator for addition only (tkdemo-2term_en.py)

Row	Source code	Explanation
1	`import␣tkinter␣as␣tk`	Import tkinter using its short form tk
2		
3	`#␣Defines␣variables␣for␣calculation␣functions␣and␣functions␣for␣events.`	
4		Note that there is no dependency on variables, functions, or GUIs to handle binary operations
5	`#␣Model␣for␣binary␣operations`	
6	`#␣Number␣being␣entered`	
7	`current_number␣=␣0`	
8	`#␣First␣number`	
9	`first_term␣=␣0`	
10	`#␣Second␣number`	
11	`second_term␣=␣0`	

```
#␣Result
result␣=␣0

def␣do_plus():
␣␣␣␣"calculation␣when␣+␣key␣was␣pressed,␣set␣the␣first␣term␣and␣clear␣the␣current␣input"
␣␣␣␣global␣current_number
␣␣␣␣global␣first_term

␣␣␣␣first_term␣=␣current_number
␣␣␣␣current_number␣=␣0

def␣do_eq():
␣␣␣␣"calculation␣when␣=␣key␣was␣pressed,␣set␣the␣second␣term,␣executeaddition,␣clear␣the␣current␣input"
␣␣␣␣global␣second_term
␣␣␣␣global␣result
␣␣␣␣global␣current_number
␣␣␣␣second_term␣=␣current_number
␣␣␣␣result␣=␣first_term␣+␣second_term
␣␣␣␣current_number␣=␣0

#␣Number␣key␣callback␣function
def␣key1():
␣␣␣␣key(1)
␣␣␣␣
def␣key2():
␣␣␣␣key(2)

def␣key3():
␣␣␣␣key(3)

def␣key4():
␣␣␣␣key(4)

def␣key5():
␣␣␣␣key(5)

def␣key6():
␣␣␣␣key(6)

def␣key7():
␣␣␣␣key(7)

def␣key8():
␣␣␣␣key(8)

def␣key9():
```

The definition of the widget's callback function starts here

58	␣␣␣␣key(9)	
59		
60	def␣key0():	
61	␣␣␣␣key(0)	
62		The processing of number
63	#␣Functions␣for␣batch␣processing␣number␣keys	keys is the same, only the
64	def␣key(n):	numbers are different
65	␣␣␣␣global␣current_number	
66	␣␣␣␣current_number␣=␣current_number␣*␣10␣+␣n	
67	␣␣␣␣show_number(current_number)	
68		Processing the Clear key
69	def␣clear():	
70	␣␣␣␣global␣current_number	
71	␣␣␣␣current_number␣=␣0	
72	␣␣␣␣show_number(current_number)	
73		Processing the + key
74	def␣plus():	
75	␣␣␣␣do_plus()	
76	␣␣␣␣show_number(current_number)	
77		Processing the = key
78	def␣eq():	
79	␣␣␣␣do_eq()	
80	␣␣␣␣show_number(result)	
81		Function to display a
82	def␣show_number(num):	number entered by the user
83	␣␣␣␣e.delete(0,tk.END)	
84	␣␣␣␣e.insert(0,str(num))␣	
85	␣␣␣␣	
86	#␣Create␣the␣tkinter␣screen	
87		Create a window with Tk()
88	root␣=␣tk.Tk()	Create a frame container
89	f␣=␣tk.Frame(root)	Assign a frame
90	f.grid()	
91		
92	#␣Create␣a␣widget	
93		Create a button with
94	b1␣=␣tk.Button(f,text='1',␣command=key1)	display text and a
95	b2␣=␣tk.Button(f,text='2',␣command=key2)	callback function
96	b3␣=␣tk.Button(f,text='3',␣command=key3)	
97	b4␣=␣tk.Button(f,text='4',␣command=key4)	
98	b5␣=␣tk.Button(f,text='5',␣command=key5)	
99	b6␣=␣tk.Button(f,text='6',␣command=key6)	
100	b7␣=␣tk.Button(f,text='7',␣command=key7)	
101	b8␣=␣tk.Button(f,text='8',␣command=key8)	
102	b9␣=␣tk.Button(f,text='9',␣command=key9)	
103	b0␣=␣tk.Button(f,text='0',␣command=key0)	
104	bc␣=␣tk.Button(f,text='C',␣command=clear)	
105	bp␣=␣tk.Button(f,text='+',␣command=plus)	

```
be␣=␣tk.Button(f,text=”=”,␣command=␣eq)

#␣Lay␣out␣widgets␣using␣a␣grid␣geometry␣manager

b1.grid(row=3,column=0)
b2.grid(row=3,column=1)
b3.grid(row=3,column=2)
b4.grid(row=2,column=0)
b5.grid(row=2,column=1)
b6.grid(row=2,column=2)
b7.grid(row=1,column=0)
b8.grid(row=1,column=1)
b9.grid(row=1,column=2)
b0.grid(row=4,column=0)
bc.grid(row=1,column=3)
be.grid(row=4,column=3)
bp.grid(row=2,column=3)

#␣Widget␣to␣display␣numbers
e␣=␣tk.Entry(f)
e.grid(row=0,column=0,columnspan=4)
clear()

#␣The␣GUI␣starts␣here
root.mainloop()
```

Assign the button to the frame by specifying its position with grid

Create an Entry widget for text input to display numeric values, and define its horizontal size

Go to GUI processing with the mainloop method

9.6 Basic structure of a program using tkinter

Importing the module

```
import tkinter as tk # Import tkinter as the shorter custom name of tk
```

Define the callback function

```
  def key1():
      Contents of key1
```

Create the window

```
root = tk.Tk()
```

Create and allocate a frame. A frame is a kind of container that stores widgets inside it.

```
f = tk.Frame(root)        # Create a frame with root as its parent,
f.grid()                 # and allocate it to a location with grid()
```

Creating a widget (button)

```
b1 = tk.Button(f, text=‘1’, command=key1)
```

```
    # Create a button with f as the parent. The text to display is '1', command
to execute is key1
```

1. Create a widget (entry, to display text)
   ```
   e = tk.Entry(f)
   ```

2. Specify the layout
   ```
   b1.grid(row=3, column=0)
   ```

3. Run the GUI
   ```
     root.mainloop()
   ```

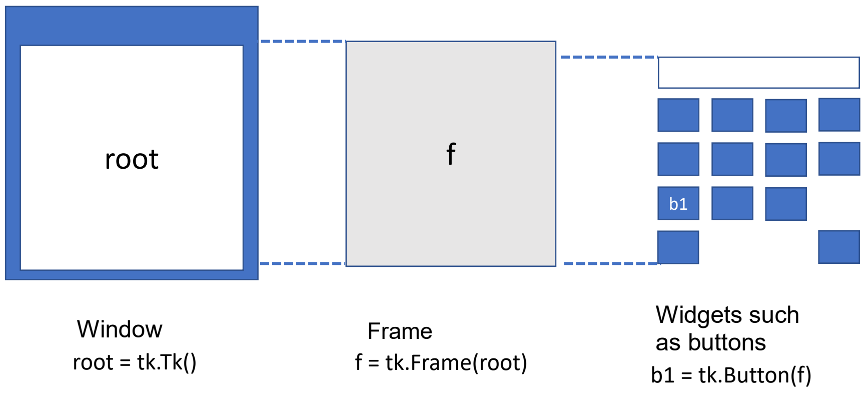


Figure 9-4 Object relationships in tkinter

9.7 Layout on a grid

A widget in tkinter can be assigned to a window or container after specifying a layout manager to manage the layout. There are several layout managers, but one of the simplest is grid, which provides a grid for positioning widgets. It can be used as follows.

- Define a grid layout by specifying its position
  ```
  b1.grid(row=3,column=0)
  ```

- It is also possible to specify how many columns there will be
  ```
  e.grid(row=0,column=0,columnspan=4)
  ```

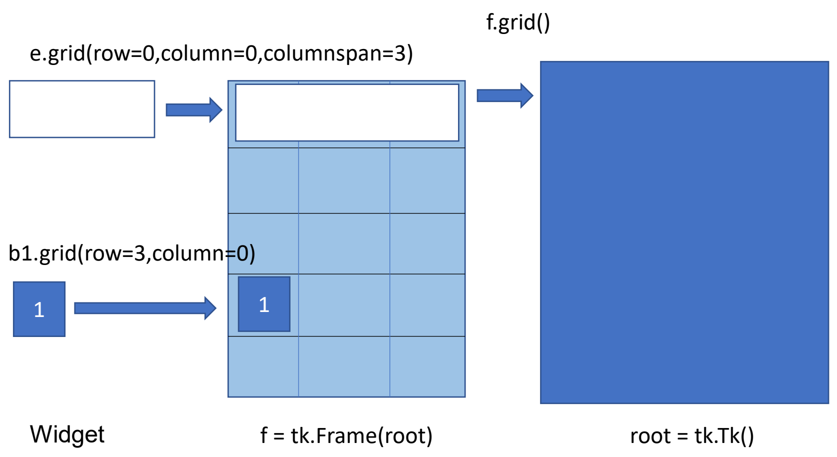


Figure 9-5 Layout with grid

9.8 Writing a callback function using a lambda (λ) notation

In the previous example, the contents of the functions key0() to key9() are calls to the function key() with different arguments. In the definition of the widget, the command=key1 argument on the right side of

```
b1 = tk.Button(f,text='1', command=key1)
```

must be a "function object," and if you write

```
b1 = tk.Button(f,text='1', command=key(1)) # This is wrong
```

then the argument of the key() function will be set to 1, the function is called, and into the command, the return value is assigned. If you designate an argument of 1 to the key() function, it will not be called as a callback function. On the other hand, the function key1() is a dedicated callback function for button b1, and is not used for any other purpose. In this sense, it would be convenient to directly link the definition of the function key1() with the specification of the callback function for button b1, eliminating the need for the name key1.

Python uses the lambda notation [1] to achieve this, where a function is defined and assigned to a variable without naming it on the fly. In the example of b1,

```
b1 = tk.Button(f,text='1', command=lambda:key(1))
```

The key(1) following lambda: is the content of the ad hoc function definition, which is essentially the same as the key1() function.

[1] The name comes from a theoretical model called lambda calculus, which is used in computer science.

Program 9-2 Setting up a callback function with arguments using a lambda notation (tkdemo-2term_lambda_en.py)

Row	Source code	Explanation
1	import␣tkinter␣as␣tk	Import tkinter using its short form tk
2		
3	#␣Defines␣variables␣for␣calculation␣functions␣and␣those for␣events.	
4	#␣Model␣for␣binary␣operations	
5	#␣Number␣being␣entered	
6	current_number␣=␣0	
7	#␣First␣number	
8	first_term␣=␣0	
9	#␣Second␣number	
10	second_term␣=␣0	
11	#␣Result	
12	result␣=␣0	
13		
14	def␣do_plus():	
15	␣␣␣␣"calculation␣when␣+␣key␣was␣pressed,␣set␣the␣first␣term␣and␣clear␣the␣current␣input"	
16	␣␣␣␣global␣current_number	
17	␣␣␣␣global␣first_term	
18		
19	␣␣␣␣first_term␣=␣current_number	
20	␣␣␣␣current_number␣=␣0	
21		
22	def␣do_eq():	
23	␣␣␣␣"calculation␣when␣=␣key␣was␣pressed,␣set␣the␣second␣term,␣executeaddition,␣clear␣the␣current␣input"	
24	␣␣␣␣global␣second_term	
25	␣␣␣␣global␣second_term	
26	␣␣␣␣global␣result	
27	␣␣␣␣global␣current_number	
28	␣␣␣␣second_term␣=␣current_number	
29	␣␣␣␣result␣=␣first_term␣+␣second_term	
30	␣␣␣␣current_number␣=␣0	
31		
32	#␣Functions␣for␣batch␣processing␣number␣keys	
33	def␣key(n):	
34	␣␣␣␣global␣current_number	
35	␣␣␣␣current_number␣=␣current_number␣*␣10␣+␣n	
36	␣␣␣␣show_number(current_number)	
37		
38	def␣clear():	
39	␣␣␣␣global␣current_number	
40	␣␣␣␣current_number␣=␣0	

```
    show_number(current_number)

def plus():
    do_plus()
    show_number(current_number)

def eq():
    do_eq()
    show_number(result)

def show_number(num):
    e.delete(0,tk.END)
    e.insert(0,str(num)) 
    
# Create the tkinter screen

root = tk.Tk()
f = tk.Frame(root)
f.grid()

# Create a widget

b1 = tk.Button(f,text=’1’, command=lambda:key(1))
b2 = tk.Button(f,text=’2’, command= lambda:key(2))
b3 = tk.Button(f,text=’3’, command= lambda:key(3))
b4 = tk.Button(f,text=’4’, command= lambda:key(4))
b5 = tk.Button(f,text=’5’, command= lambda:key(5))
b6 = tk.Button(f,text=’6’, command= lambda:key(6))
b7 = tk.Button(f,text=’7’, command= lambda:key(7))
b8 = tk.Button(f,text=’8’, command= lambda:key(8))
b9 = tk.Button(f,text=’9’, command= lambda:key(9))
b0 = tk.Button(f,text=’0’, command= lambda:key(0))
bc = tk.Button(f,text=’C’, command=clear)
bp = tk.Button(f,text=’+’, command=plus)
be = tk.Button(f,text=”=”, command= eq)

# Lay out widgets using a grid geometry manager

b1.grid(row=3,column=0)
b2.grid(row=3,column=1)
b3.grid(row=3,column=2)
b4.grid(row=2,column=0)
b5.grid(row=2,column=1)
b6.grid(row=2,column=2)
b7.grid(row=1,column=0)
b8.grid(row=1,column=1)
b9.grid(row=1,column=2)
b0.grid(row=4,column=0)
```

```
bc.grid(row=1,column=3)
be.grid(row=4,column=3)
bp.grid(row=2,column=3)

#␣Widget␣to␣display␣numbers
e␣=␣tk.Entry(f)
e.grid(row=0,column=0,columnspan=4)
clear()

#␣The␣GUI␣starts␣here
root.mainloop()
```

9.9 Configuring the appearance of a widget

There are several ways to configure the color, size, font, etc. of a widget, such as a button.

9.9.1 How to set an argument when creating the widget

You can specify arguments when generating the widget. It is convenient to use keyword arguments.

● font=('Helvetica', 14)	Text font and size
● width=2	The size of the widget (For buttons, the number of characters)
● bg = '#ffffc0'	Background color, two hexadecimal digits each in RGB 00 is darker, ff is lighter

Note that bg does not currently change the color on the Mac, so if you set the argument highlightbackground instead of bg, the color of the button itself will not change, but the color around the button will. Please substitute highlightbackground in for bg if you are doing this exercise on a Mac.

Table 9-1 Specifying colors with tkinter

Notation	Red	Green	Blue	Color
'#ffffff'	ff	ff	ff	White
'#000000'	00	00	00	Black
'#ff0000'	ff	00	00	Red
'#00ff00'	00	ff	00	Green
'#0000ff'	00	00	ff	Blue

9.9.2 Configuring the Appearance of a Generated Widget (Part 1)

To change the appearance of a generated widget while it is running, use the configure() method of the widget. The arguments are allocated in the same way as the keyword arguments are at the time of generation. For example, suppose b is a button widget as follows:

```
b.configure(size=2)
```

To reference the current value, call the cget() method with the attribute as an argument. Note that the attribute is given as a string.

```
b.cget(“size”)
```

9.9.3 Configuring the Appearance of the Generated Widget (Part 2)

The appearance of the widget can also be set and referenced in the following way. Note that the settings in square brackets must be a string (enclosed by quotation marks, etc.). Settings are performed by substitution.

```
b[″size″] =2
print(b[“size”])
```

Exercise 9-1 Create an Addition Calculator with Tkinter

Write a program like Program 9-1 or Program 9-2 and see how it works.

Exercise 9-2 Configuring the appearance of a widget

- Set the font size of the addition calculator and the color of the widget as follows. Set the background color to '#ffffc0' (light yellow) for the frame, white for numeric keys, red for clear keys, and green for + and = keys.
- The size of the button should be 2 (the length in characters).
- The font and size of the button and entry should be ('Helvetica', 14).

Exercise 9-3 Extending the Calculator to Four Arithmetic Operations (Skill Test)

Extend the addition calculator so that it can perform the four primary arithmetic operations. However, keep the following in mind.

- You must consider the placement of the buttons.
- For division, since a divide by zero error may occur, when the second term is zero, either do nothing or display an error.

- Omit the decimal point for division. The operator for finding the integer quotient in Python is "//."

Hint: The program needs to be extended in the following two ways.

- Add a button widget to specify the four arithmetic operations.
  (This should not be too difficult.)
- Set up a callback function for when the arithmetic operator button or the "=" button is pressed.
  Consider the following hints.
  - In a calculator, when an operation key such as + is pressed, the operation is not performed; it is performed when the = key is pressed. Thus, when an arithmetic key (for addition, subtraction, multiplication, or division) is pressed, the calculator stores the operation to be performed in a variable until it is executed. For example, you could set up a variable called "operation" and set it to 1 for addition, 2 for subtraction, 3 for multiplication, 4 for division.
  - When the = key is pressed, you need to change the operation that is actually performed to match the stored operation. For example, your code may look like this:

```
if operation == 1:
    Block that performs addition
elif operation == 2:
    Block that performs subtraction
elif operation == 3:
    Block that performs multiplication
else:
    Block that performs division
```

Exercise 9-4 Management of Widgets with List (Skill Test)

In Program 9-1 andProgram 9-2, we use many Button widgets. Try to manage them by using these objects as members of a list. With this list, try to rewrite the program using 'for' statement.

Note: we use a lambda expression in Program 9-2 (for example, `command=lambda:key(1)`). Here, the argument of function key(1) is given as a constant, but with this expression, we can't use a variables (for example, i) whose value should be exaluated at the definition of the lambda expression. It can be written using lambda expression with an argument having a default value like the following:

```
command=lambda x = i:key(x)
```

Exercise 9-5 Differences with an actual calculator

Look for differences in behavior between your program and an actual calculator (or calculator application). For example, what happens when you press an arithmetic key such as + instead of =? You will find that actual calculator products are well designed.

9.10 How to close tkinter

In an application using tkinter, calling mainloop() will result in an infinite loop that waits for user action and then calls the callback function. Methods of closing tkinter other than exiting with the exit button of the window are as follows:

- To escape from mainloop(), call the quit() or destroy() method of the object created by tk.Tk() (e.g., root) in a given callback function. The difference in the behavior of these methods is as follows:
  - quit(): The program exits the loop, but the window or widgets remain.
  - destroy(): exits the loop and removes the window or widgets altogether.

9.11 How to extend the Frame class

In the previous program, Frame and widgets such as the Button widget were configured separately. However, in practical examples of tkinter, Frame is often implemented as an extended class and widgets are generated during its initialization. In this section, we show an example of this. For more information about classes, please refer to the explanation in a later chapter.

Program 9-3 How to extend the Frame class in tkinter (tkdemo-2term_frame_extention_en.py)

Row	Source code	Explanation
1	`import␣tkinter␣as␣tk`	
2		
3	`#␣Defines␣variables␣for␣calculation␣functions␣and␣those␣for␣events.`	
4	`#␣Example␣using␣Frame␣subclasses`	
5		
6	`#␣Model␣for␣binary␣operations`	
7	`#␣Number␣being␣entered`	
8	`current_number␣=␣0`	
9	`#␣First␣number`	
10	`first_term␣=␣0`	
11	`#␣Second␣number`	
12	`second_term␣=␣0`	

```
# Result
result = 0

def do_plus():
    "calculation when + key was pressed, set the first term and clear the current input"
    global current_number
    global first_term
    first_term = current_number
    current_number = 0

def do_eq():
    "calculation when = key was pressed, set the second term, executeaddition, clear the current input"
    global second_term
    global result
    global current_number
    second_term = current_number
    result = first_term + second_term
    current_number = 0
#
# Create a class called MyFrame that inherits from tk.Frame
# Then you can add widgets and callback functions (methods) in it,
# Which is the standard way of using tkinter
#
class MyFrame(tk.Frame):
#
#  __init__ is the initialization method used to create class objects;
# It has two underscores on each side
    def __init__(self, master = None):
        super().__init__(master)
# Create a widget that will not be referenced later (with local variables)
        b1 = tk.Button(self,text='1', command=lambda:self.key(1))
        b2 = tk.Button(self,text='2', command=lambda:self.key(2))
        b3 = tk.Button(self,text='3', command=lambda:self.key(3))
        b4 = tk.Button(self,text='4', command=lambda:self.key(4))
        b5 = tk.Button(self,text='5', command=lambda:self.key(5))
        b6 = tk.Button(self,text='6', command=lambda:self.key(6))
        b7 = tk.Button(self,text='7', command=lambda:self.key(7))
        b8 = tk.Button(self,text='8', command=lambda:self.key(8))
        b9 = tk.Button(self,text='9', command=lambda:self.key(9))
        b0 = tk.Button(self,text='0', command=lambda:self.key(0))
        bc = tk.Button(self,text='C', command=self.clear)
        bp = tk.Button(self,text='+', command=self.plus)
        be = tk.Button(self,text="=", command=self.eq)

```

```
# Lay out widgets using a grid geometry manager
        b1.grid(row=3,column=0)
        b2.grid(row=3,column=1)
        b3.grid(row=3,column=2)
        b4.grid(row=2,column=0)
        b5.grid(row=2,column=1)
        b6.grid(row=2,column=2)
        b7.grid(row=1,column=0)
        b8.grid(row=1,column=1)
        b9.grid(row=1,column=2)
        b0.grid(row=4,column=0)
        bc.grid(row=1,column=3)
        be.grid(row=4,column=3)
        bp.grid(row=2,column=3)

# Widgets and class objects that display numbers that will
# be referenced by other methods are created as an instance
# variable with the "self." prefix
        self.e = tk.Entry(self)
        self.e.grid(row=0,column=0,columnspan=4)
# When defining a class,
# The first parameter of a method is "self."
# Inside of a class, the class object variable and method also
# reference "self"
    def key(self,n):
        global current_number
        current_number = current_number * 10 + n
        self.show_number(current_number)

    def clear(self):
        global current_number
        current_number = 0
        self.show_number(current_number)

    def plus(self):
        do_plus()
        self.show_number(current_number)

    def eq(self):
        do_eq()
        self.show_number(result)

    def show_number(self, num):
       self.e.delete(0,tk.END)
       self.e.insert(0,str(num))
        
#
# Main program from here
```

`#`	
`root␣=␣tk.Tk()`	
`f = MyFrame(root)`	Use the extended class. pack() is a geometry manager.
`f.pack()`	
`f.mainloop()`	

References

There are many resources on the internet explaining how to use Tkinter. The use of tkinter is slightly different between Python 2 and Python 3. For example, the module to import is Tkinter in Python 2, while it is tkinter in Python 3. Please be careful when referring to articles online.

[17] Tkinter 8.5 reference: a GUI for Python
https://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html

10. Creating a GUI Application with Tkinter (2)

10.1 Learning goals of this chapter

In this chapter, you will learn the following skills by creating an analog clock with tkinter:

- How to use autonomous programs such as animations together with a GUI, and
- How to draw graphics using the Canvas widget.

10.2 Conflicts between autonomous programs and GUIs

GUI frameworks such as tkinter observe user operations and delegate processing to a set callback function when an event such as a mouse click occurs. At that time, the callback function is expected to terminate promptly. It waits for the callback function to finish, then goes back to observing, waiting for an event to occur.

At the same time, if the program itself operates continuously (as is the case with an animation) and if the callback function is then called, the observation of events will stop.

In tkinter, there is a method called after that executes the specified callback function after a given period of time to meet both of these needs. Using the after method allows tkinter to register the process after the specified time and terminate the callback function so that the GUI event observation loop is not stopped for a long period of time.

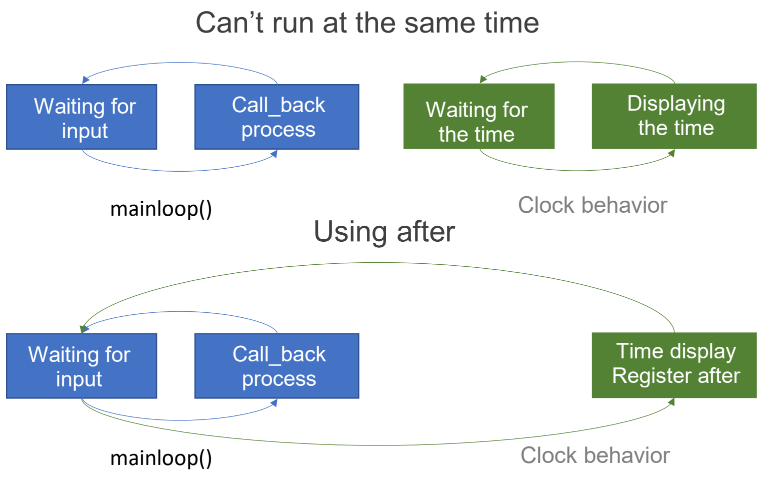


Figure 10-1 Use of 'after' in tkinter

As a caveat, note that this way of doing things is not suitable for applications that require a lot of computation time, such as simulations. You need to look into the use of libraries and concepts such as threads to run programs in parallel.

10.3 Analog clock program using tkinter

In this section, you will create an analog clock as shown in Figure 10-2. It displays the hour, minute, and second hands, and the date display can be turned on and off with a button. Refer to Figure 10-3 for calculating the position of the clock hands.

The following program is implemented by extending the Frame class. Some lines are longer than others; in the list below, the lines without numbers are a part of long single lines that have had the text wrapped onto a new line. Please be careful when you type this code out.

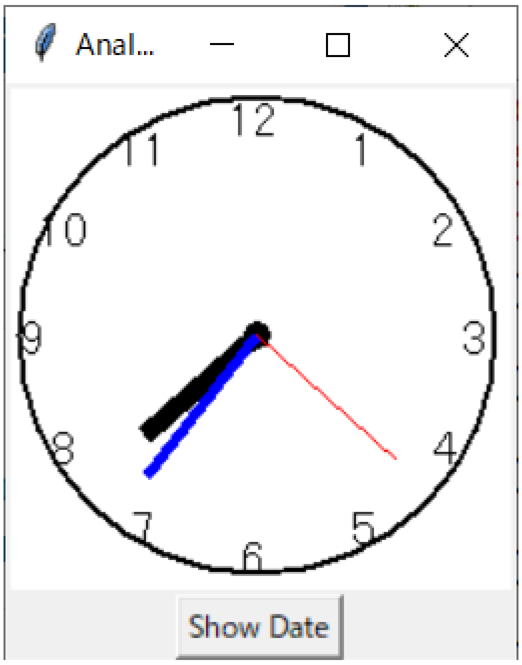


Figure 10-2 The analog clock you will create

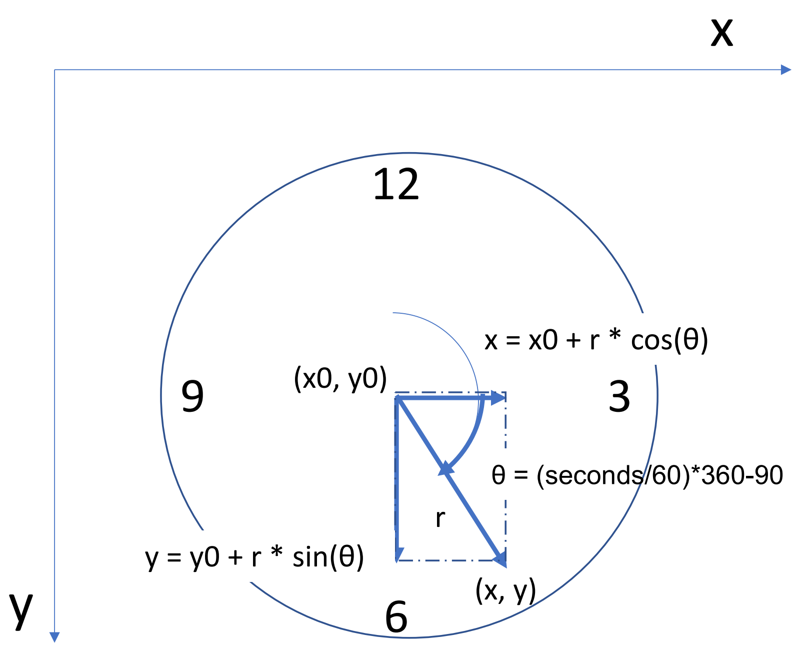


Figure 10-3 Calculation of the position of the hands of the clock

10.3.1 Source code

The program is a bit complicated, so at first, we will look at the code without the button to turn on/off the date display. The code with the button added is shown further below.

Program 10-1 Analog clock with tkinter (without a date button, tkdemo_simple_clock.py)

Row	Source code	Explanation
1	`#`	
2	`#␣Analog␣clock␣with␣no␣date␣button␣using␣tkinter␣canvas`	
3	`#`	
4	`import␣tkinter␣as␣tk`	Import time to
5	`import␣math`	handle time
6	`import␣time`	
7		
8	`#`	
9	`#␣Extend␣the␣Frame␣class`	
10	`#`	
11	`class␣MyFrame(tk.Frame):`	Two underscores on
12	`␣␣␣␣def␣__init__(self,␣master␣=␣None):`	each side
13	`␣␣␣␣␣␣␣␣super().__init__(master)`	

14	#	
15	#␣Create␣a␣canvas	
16	#	
17	␣␣␣␣␣␣␣␣self.size␣=␣200	
18	␣␣␣␣␣␣␣␣self.clock␣=␣tk.Canvas(self,␣width=self.size,␣height=self.size,␣background=”white”)	Widget for drawing
19	␣␣␣␣␣␣␣␣self.clock.grid(row=0,␣column=0)	
20	#	
21	#␣Draw␣the␣dial	
22	#	
23	␣␣␣␣␣␣␣␣self.font_size␣=␣int(self.size/15)	
24	␣␣␣␣␣␣␣␣for␣number␣in␣range(1,12+1):	
25	␣␣␣␣␣␣␣␣␣␣␣␣x␣=␣self.size/2␣+␣math.cos(math.radians(number*360/12␣-␣90))*self.size/2*0.85	
26	␣␣␣␣␣␣␣␣␣␣␣␣y␣=␣self.size/2␣+␣math.sin(math.radians(number*360/12␣-␣90))*self.size/2*0.85	
27	␣␣␣␣␣␣␣␣␣␣␣␣self.clock.create_text(x,y,text=str(number),␣fill=”black”,␣font␣=(””,14))	
28	#	
29	#␣Create␣instance␣variables␣to␣check␣for␣the␣passage␣of␣time	
30	#	
31	␣␣␣␣␣␣␣␣self.sec␣=␣time.localtime().tm_sec	
32	␣␣␣␣␣␣␣␣self.min␣=␣time.localtime().tm_min	
33	␣␣␣␣␣␣␣␣self.hour␣=␣time.localtime().tm_hour	
34	␣␣␣␣#	
35	␣␣␣␣#␣Draw␣a␣dynamic␣display	
36	␣␣␣␣#	
37	␣␣␣␣def␣display(self):	
38	␣␣␣␣␣␣␣␣#	
39	␣␣␣␣␣␣␣␣#␣Draw␣the␣second␣hand	
40	␣␣␣␣␣␣␣␣#	
41	␣␣␣␣␣␣␣␣self.sec␣=␣time.localtime().tm_sec	
42	␣␣␣␣␣␣␣␣angle␣=␣math.radians(self.sec*360/60␣-␣90)	
43	␣␣␣␣␣␣␣␣x0␣=␣self.size/2␣-␣math.cos(angle)*self.size/2*0.1	
44	␣␣␣␣␣␣␣␣y0␣=␣self.size/2␣-␣math.sin(angle)*self.size/2*0.1	
45	␣␣␣␣␣␣␣␣x␣=␣self.size/2␣+␣math.cos(angle)*self.size/2*0.75	
46	␣␣␣␣␣␣␣␣y␣=␣self.size/2␣+␣math.sin(angle)*self.size/2*0.75	
47	␣␣␣␣␣␣␣␣#	
48	␣␣␣␣␣␣␣␣#␣Search␣for␣the␣previous␣drawing␣using␣its␣tag,␣delete␣it,␣then␣redraw␣the␣new␣line	
49	␣␣␣␣␣␣␣␣#	
50	␣␣␣␣␣␣␣␣self.clock.delete(”SEC”)	
51	␣␣␣␣␣␣␣␣self.clock.create_line(x0,y0,x,y,␣width=1,␣fill=”red”,␣tag=”SEC”)	
52	␣␣␣␣␣␣␣␣#	
53	␣␣␣␣␣␣␣␣#␣Draw␣the␣minute␣and␣hour␣hands,␣and␣make␣the␣hour␣hand␣move␣slightly␣every␣minute	

54	␣␣␣␣␣␣␣␣#	
55	␣␣␣␣␣␣␣␣x0␣=␣self.size/2	
56	␣␣␣␣␣␣␣␣y0␣=␣self.size/2	
57	␣␣␣␣␣␣␣␣self.min␣=␣time.localtime().tm_min	
58	␣␣␣␣␣␣␣␣angle␣=␣math.radians(self.min*360/60␣-␣90)	
59	␣␣␣␣␣␣␣␣x␣=␣self.size/2␣+␣math.cos(angle)*self.size/2*0.65	
60	␣␣␣␣␣␣␣␣y␣=␣self.size/2␣+␣math.sin(angle)*self.size/2*0.65	
61	␣␣␣␣␣␣␣␣self.clock.delete("MIN")	
62	␣␣␣␣␣␣␣␣self.clock.create_line(x0,y0,x,y,␣width=3,␣fill="blue",␣tag="MIN")	
63		
64	␣␣␣␣␣␣␣␣self.hour␣=␣time.localtime().tm_hour	
65	␣␣␣␣␣␣␣␣x0␣=␣self.size/2	
66	␣␣␣␣␣␣␣␣y0␣=␣self.size/2	
67	␣␣␣␣␣␣␣␣angle␣=␣math.radians((self.hour%12+self.min/60)*360/12␣-␣90)	
68	␣␣␣␣␣␣␣␣x␣=␣self.size/2␣+␣math.cos(angle)*self.size/2*0.55	
69	␣␣␣␣␣␣␣␣y␣=␣self.size/2␣+␣math.sin(angle)*self.size/2*0.55	
70	␣␣␣␣␣␣␣␣self.clock.delete("HOUR")	
71	␣␣␣␣␣␣␣␣self.clock.create_line(x0,y0,x,y,␣width=3,␣fill="green",␣tag="HOUR")	
72	␣␣␣␣␣␣␣␣#	
73	␣␣␣␣␣␣␣␣#␣Draw␣the␣date␣	
74	␣␣␣␣␣␣␣␣#	
75	␣␣␣␣␣␣␣␣x␣=␣self.size/2	
76	␣␣␣␣␣␣␣␣y␣=␣self.size/2␣+␣20	
77	␣␣␣␣␣␣␣␣text␣=␣time.strftime('%Y/%m/%d␣%H:%M:%S')	
78	␣␣␣␣␣␣␣␣self.clock.delete("TIME")	
79	␣␣␣␣␣␣␣␣self.clock.create_text(x,␣y,␣text=text,␣font=("",12),␣fill="black",␣tag="TIME")	
80	␣␣␣␣␣␣␣␣#	
81	␣␣␣␣␣␣␣␣#␣Call␣again␣in␣100␣milliseconds	
82	␣␣␣␣␣␣␣␣#␣	
83	␣␣␣␣␣␣␣␣self.after(100,␣self.display)	
84		
85	root␣=␣tk.Tk()	Main program
86	f␣=␣MyFrame(root)	starts here
87	f.pack()	
88	f.display()	Call display first
89	root.mainloop()	

Program 10-2 Analog clock with tkinter
(with a date button, tkdemo_clock_with_button.py)

Row	Source code	Explanation

```
#
#␣Analog␣clock␣with␣a␣date␣button␣using␣tkinter␣canvas
#
import␣tkinter␣as␣tk
import␣math
import␣time

#
#␣Extend␣the␣Frame␣class
#
class␣MyFrame(tk.Frame):
␣␣␣␣def␣__init__(self,␣master␣=␣None):
␣␣␣␣␣␣␣␣super().__init__(master)
#
#␣Create␣a␣canvas
#
␣␣␣␣␣␣␣␣self.size␣=␣200
␣␣␣␣␣␣␣␣self.clock␣=␣tk.Canvas(self,␣width=self.size,␣height=self.size,␣background="white")
␣␣␣␣␣␣␣␣self.clock.grid(row=0,␣column=0)
#
#␣Draw␣the␣dial
#
␣␣␣␣␣␣␣␣self.font_size␣=␣int(self.size/15)
␣␣␣␣␣␣␣␣for␣number␣in␣range(1,12+1):
␣␣␣␣␣␣␣␣␣␣␣␣x␣=␣self.size/2␣+␣math.cos(math.radians(number*360/12␣-␣90))*self.size/2*0.85
␣␣␣␣␣␣␣␣␣␣␣␣y␣=␣self.size/2␣+␣math.sin(math.radians(number*360/12␣-␣90))*self.size/2*0.85
␣␣␣␣␣␣␣␣␣␣␣␣self.clock.create_text(x,y,text=str(number),␣fill="black",␣font␣=("",14))
#
#␣Create␣a␣button␣to␣toggle␣the␣date␣display␣on/off
#
␣␣␣␣␣␣␣␣self.b␣=␣tk.Button(self,␣text="Show␣Date",␣font=("",14),␣command␣=␣self.toggle)
␣␣␣␣␣␣␣␣self.b.grid(row␣=␣1,␣column␣=␣0)
#
#␣Create␣instance␣variables␣to␣check␣for␣the␣passage␣of␣time
#
␣␣␣␣␣␣␣␣self.sec␣=␣time.localtime().tm_sec
␣␣␣␣␣␣␣␣self.min␣=␣time.localtime().tm_min
␣␣␣␣␣␣␣␣self.hour␣=␣time.localtime().tm_hour
␣␣␣␣␣␣␣␣self.show_date␣=␣False

␣␣␣␣#
␣␣␣␣#␣Callback␣when␣the␣button␣is␣pressed
```

Import time to handle time

Widget for drawing

```
    #
    def toggle(self):
        if self.show_date:
            self.b.configure(text="show date")
        else:
            self.b.configure(text="hide date")
        self.show_date = not self.show_date

    #
    # Draw a dynamic display
    #
    def display(self):
        #
        #  Draw the second hand
        #
        self.sec = time.localtime().tm_sec
        angle = math.radians(self.sec*360/60 - 90)
        x0 = self.size/2 - math.cos(angle)*self.size/2*0.1
        y0 = self.size/2 - math.sin(angle)*self.size/2*0.1
        x = self.size/2 + math.cos(angle)*self.size/2*0.75
        y = self.size/2 + math.sin(angle)*self.size/2*0.75
        #
        # Search for the previous drawing using its tag, delete it, then redraw the new line
        #
        self.clock.delete("SEC")
        self.clock.create_line(x0,y0,x,y, width=1, fill="red", tag="SEC")
        #
        # Draw the minute and hour hands, and make the hour hand move slightly every minute
        #
        self.min = time.localtime().tm_min
        x0 = self.size/2
        y0 = self.size/2
        angle = math.radians(self.min*360/60 - 90)
        x = self.size/2 + math.cos(angle)*self.size/2*0.65
        y = self.size/2 + math.sin(angle)*self.size/2*0.65
        self.clock.delete("MIN")
        self.clock.create_line(x0,y0,x,y, width=3, fill="blue", tag="MIN")
        self.hour = time.localtime().tm_hour
        x0 = self.size/2
        y0 = self.size/2
        angle = math.radians((self.hour%12+self.min/60)*360/12 - 90)
        x = self.size/2 + math.cos(angle)*self.size/2*0.55
        y = self.size/2 + math.sin(angle)*self.size/2*0.55
```

Is the date displayed?
Change the button text
Invert whether or not the date is shown

```
␣␣␣␣␣␣␣␣self.clock.delete("HOUR")
␣␣␣␣␣␣␣␣self.clock.create_line(x0,y0,x,y,␣width=3,␣fill=
"green",␣tag="HOUR")

␣␣␣␣␣␣␣␣#
␣␣␣␣␣␣␣␣#␣Draw␣the␣date
␣␣␣␣␣␣␣␣#
␣␣␣␣␣␣␣␣x␣=␣self.size/2
␣␣␣␣␣␣␣␣y␣=␣self.size/2␣+␣20
␣␣␣␣␣␣␣␣text␣=␣time.strftime('%Y/%m/%d␣%H:%M:%S')
␣␣␣␣␣␣␣␣self.clock.delete("TIME")
␣␣␣␣␣␣␣␣if␣self.show_date:
␣␣␣␣␣␣␣␣␣␣␣␣self.clock.create_text(x,␣y,␣text=text,␣font=("
",12),␣fill="black",␣tag="TIME")
␣␣␣␣␣␣␣␣#
␣␣␣␣␣␣␣␣#␣Call␣again␣in␣100␣milliseconds
␣␣␣␣␣␣␣␣#␣
␣␣␣␣␣␣␣␣self.after(100,␣self.display)

root␣=␣tk.Tk()
f␣=␣MyFrame(root)
f.pack()
f.display()
root.mainloop()
```

Draw only when the date is displayed

Main program starts here

10.3.2 Key points of this program

- You will need to import modules other than tkinter, including the time module, which is imported to handle time, and the math module, which is imported to use trigonometric functions.
- This program defines the MyFrame class, which is an extension of Frame, then creates widgets, assigns them, and defines callback functions within it.
    - The __init__() method of the MyFrame class is automatically called when the class object is created. In this method, the necessary widgets are created. There are two underscores on each side of the method name.
        - The Canvas widget is created for drawing
        - The dial is drawn by calling the create_text() method of the Canvas widget.
        - A button is created to toggle the display of the time in text. (Only for the code that includes the button)
        - Instance variables are allocated for time elapsing, display switching, etc., and set the values using the seconds, minutes, hours, etc., in the time.localtime() function.

- ➢ A callback function is defined for when the button is pressed. The b.configure() method is called to switch the text displayed on the button (b), and set the variable representing the state. (Only in the code with the button)
- ➢ Method to draw the clock face:
  - ✧ When using Canvas to draw, you can attach a "tag" to the drawn object, so that you can delete it later. First you will need to delete the old drawing (clock hand).
  - ✧ The coordinates of the clock hands are calculated from the time using a trigonometric function, and drawn using the create_line() method.
  - ✧ At the end of this method, the after method is set to make this method call itself 100 milliseconds later to continuously draw the clock.

- Finally, there is the main program. The program creates a window with the Tk() method, creates a MyFrame class object, draws the first instance of the clock with f.display(), and then passes control of the program to tkinter with mainloop().

Exercise 10-1Reviewing the methods to be used

List the methods of the time module, math module, and tkinter Canvas class that are called in this program, and look over these methods in detail in the Python online documentation.

Exercise 10-2 Modifying the analog clock

Make the following modifications to the analog clock program.

1. For the date display, have it display the date and the time using a.m. and p.m. instead of just the date and time.
2. Add another button to toggle the display of the second hand on and off.

Hint: Review the role of the self.toggle method, and figure out how to do the same thing for the second hand display.

Exercise 10-3 Improving the display

This program redraws the clock hands and the date every 100 milliseconds, but it also redraws things that do not change. The date and position of the second hand change every second, but the minute hand and hour hand only change every minute. Think about how you can make it so the minute and hour hands are only redrawn if there has been a change since the last time it was drawn. Also, think about how to avoid the roughly second long delay that occurs when the screen is first displayed or when a button is pressed.

10.4 Coordinating actions using variables

In Program 10-2, when the button to turn on/off the date display is pressed, the value of the show_date variable is simply switched using the toggle() method that is set as the callback function.

On the other hand, the display() method, which runs continuously on a timer, looks at the show_date variable and toggles the date display on and off.

You can use variables to create settings that coordinate the behavior of methods that work independently of each other. Variables such as show_date are called flags because they are used to raise and lower flags.

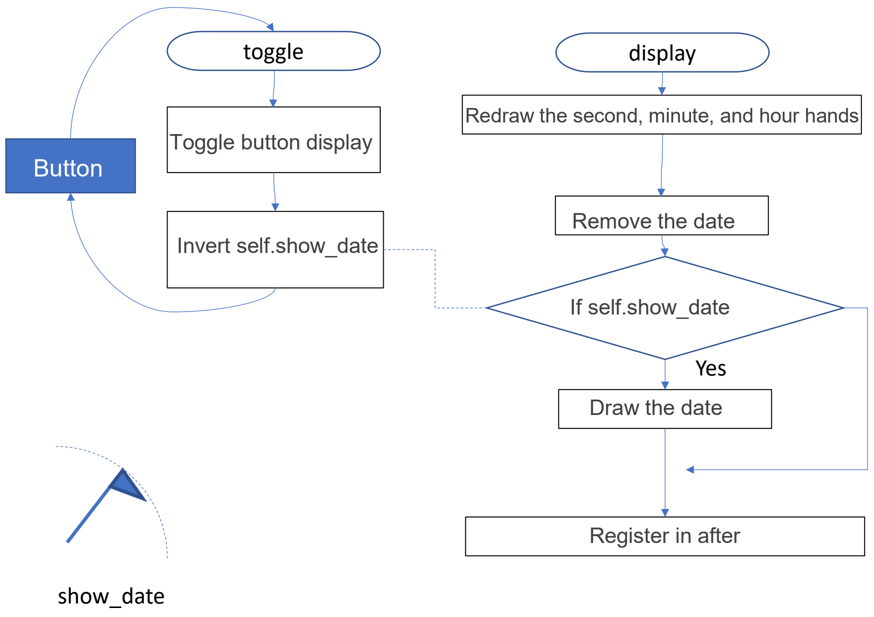


Figure 10-4 Coordinating actions using variables

11. Classes

11.1 Learning goals of this chapter

You have already used class objects in turtle graphics and tkinter. Here, you will learn about classes in more detail.

- You will learn about the concept of object-oriented programming.
- You will define and use classes.
- You will learn about variables that are used in classes.

11.2 Object-oriented programming

To handle multiple turtles in turtle graphics, we did the following.

- Generate as many turtles as needed.
- Call methods on individual turtles to tell them what to do and query their status.

Each turtle had its own state, which included its position, orientation, pen color, and whether or not the pen was down.

Objects are things like these turtles that have their own internal state and can be instructed to behave by calling methods from outside. Programming with objects is called object-oriented programming.

In the tkinter analog clock example, programming was done by extending the Frame class of tkinter. **A "class" is a description of a type that can be used to generate objects with unique states and methods**. Each object generated from a class is called an **"instance."**

In short:

- An object (of a class type) is an element of a program with its own state (variables) and methods, **like a robot** that you can order around (as we did with the turtles).
- A class is a description of what variables and methods an object of that type has; it is a type used to create objects.
- An instance is an individual object generated with a class as its type.
- Object-oriented programming is a method of creating programs by defining classes and using generated instances of them. It is a concept that can be described as **writing programs to coordinate the behavior of robots.**

11.3 How to write and use classes in Python

As in the tkinter example, we will create a character user interface (CUI) type program that performs binary operations. we will use a class to organize the variables that hold the first and second terms, the result of the operation, the operator, and the method that actually performs the operation.

11.3.1 Source code

Program 11-1 CUI calculator program (p11-1.py)

Row	Source code	Explanation
1	`class␣Calculator():`	
2	`␣␣␣␣def␣__init__(self):`	Initialization
3	`␣␣␣␣␣␣␣␣self.first_term␣=␣0`	method
4	`␣␣␣␣␣␣␣␣self.second_term␣=␣0`	
5	`␣␣␣␣␣␣␣␣self.result␣=␣0`	
6	`␣␣␣␣␣␣␣␣self.operation␣=␣"+"`	
7	`␣␣␣␣␣␣␣␣`	
8	`␣␣␣␣def␣do_operation(self):`	
9	`␣␣␣␣␣␣␣␣if␣self.operation␣==␣"+":`	Method to perform
10	`␣␣␣␣␣␣␣␣␣␣␣␣self.result␣=␣self.first_term␣+␣self.second_term`	operations
11	`␣␣␣␣␣␣␣␣elif␣self.operation␣==␣"-":`	
12	`␣␣␣␣␣␣␣␣␣␣␣␣self.result␣=␣self.first_term␣-␣self.second_term`	
13		
14	`#␣Main␣program␣starts␣here`	
15	`calculator␣=␣Calculator()`	Create an object
16	`while␣True:`	
17	`␣␣␣␣f␣=␣int(input("First␣term␣"))`	
18	`␣␣␣␣calculator.first_term␣=␣f`	
19	`␣␣␣␣o␣=␣input("Operation␣")`	
20	`␣␣␣␣calculator.operation=o`	
21	`␣␣␣␣s␣=␣int(input("Second␣term␣"))`	
22	`␣␣␣␣calculator.second_term=s`	
23	`␣␣␣␣calculator.do_operation()`	
24	`␣␣␣␣r␣=␣calculator.result`	
25	`␣␣␣␣print("Result␣",␣r)`	

Example

```
First term 1
Operation +
Second term 2
Result  3
First term 10
Operation -
Second term 5
Result  5
First term
Traceback (most recent call last):
  File "M:/Documents/Python Scripts/class_demo.py", line 17, in <module>
    f = int(input("First term "))
```

Press Ctrl-C to interrupt

Next Chapter Table of Contents

```
KeyboardInterrupt
>>>
```

11.3.2 Overview of the program

- Class definition block (lines 1-12): **classes are defined** as follows.

```
class  ClassName ():

    Definition of methods, etc.
```

- The name of the class (Calculator in this example) can be determined by the same rules as for variables, but the convention is to capitalize the first letter and write the rest in lowercase. When using multiple words in a name, "camel case" is used, whereby the first letter of each word is capitalized (with no spaces in between words) and the rest of the letters are lowercase. (For example, FunctionCalculator)
- The definition of the method __init__(self) (lines 2-6): methods and variables beginning with __ (two underscores) often have special roles in Python.
  __infit__() is a method that is executed whenever a class object is created.
  It is used to initialize variables in the class, and it is also called a "constructor" because it takes a role of making a object. Reference the column titled " Personification." Unlike functions, **class method definitions must always contain a parameter, which is usually named self by convention**. The value of this parameter is automatically given by the system when the method is called. There is no need to write the first argument when calling a class method.
- Instance variables and initialization (lines 3-6): The __init__() method initializes the variables used in the class.
  - Variables that begin with self. are called "instance variables." They are object-specific variables that can always be used within the object whenever an object of that class is created.
  - On the other hand, variables without self. are treated as local variables like in functions, and are discarded when the method is finished processing.
- Definition of the method do_operation() (lines 8-12): This is a method that can be called explicitly, which performs the specified operation on the first and second terms and prints the result. Note the self. in the code; the parameter self is attached to the instance variables when they are processed.
- Main program (line 14 onward): This is a calculator program that receives text input from the terminal and executes it. It is written in an infinite loop, so use Ctrl-C to escape.
- **Generating a class-type object** (line 15). Class-type objects are created by calling the class name like a function and assigning it to a variable.

```
Variable  =  ClassName ()
```

- Manipulation of instance variables and methods of class-type objects (lines 18-24). By appending a "." followed by an instance variable name or method name to the class object variable, you can call these methods or variables. Note that the do_operation() method requires the parameter self in the definition, but does not require it when it is called.

Exercise 11-1 Extention of Calculator Class

Extend the Calculator class to handle multiplication and division. You may return just an integer quotient for division.

Exercise 11-2 Creation and Utilization of Multiple Objects

Create a program that generates and uses multiple objects of the Calculator class. Think about what you can do if you are able to utilize many robots that do addition. For example, how about a robot that does addition and another that monitors it, then checks the figures (does subtraction)?

Exercise 11-3 User Calculator Class in the Tkinter Program

Modify the calculator program created by tkinter so that it uses the Calculator class

11.4 Class variables and access restrictions

We mentioned earlier that Python programs have global variables that are valid for the entire program and local variables that are valid only while executing within a function. You also need to learn about class variables and instance variables when working with classes.

- **Class Variables**
  - Creation: Declared in the class definition, outside of the method definition.
  - Behavior: Acts as a variable shared by the class.
  - Access: Can be referenced without creating a class object by writing ClassName.variablename in the code.
- **Instance Variables**
  - Creation: Declared by adding the self. prefix in the method definition.
  - Behavior: Each instance that is created is treated as an independent variable. The value is retained as long as the instance is in use.
  - Access: In the definition of a method, refer to it by adding the self. prefix as you do when creating it.
    To reference an instance variable in a program that used the created instance, the variable to

which the instance is assigned (e.g., a) is reference by writing the name, followed by a period (.), followed by the name of the instance variable.

Python does not have a very strong variable protection feature. Both class variables and instance variables can be referenced and rewritten externally. One way to restrict access from outside the class is to **use variables that begin with two underscores**. Such variables can be accessed by methods in the class, but cannot be manipulated directly from outside the class.

Program 11-2 Class variables and instance variables (p11-2_en.py)

Row	Source code	Explanation
1	`#␣Class␣practice`	
2	`class␣MyClass():`	Class Definition
3	`␣␣␣␣#␣The␣following␣are␣class␣variables`	
4	`␣␣␣␣a␣=␣"My␣Class"`	__b is an access-
5	`␣␣␣␣__b␣=␣0`	protected variable
6	`␣␣␣␣`	
7	`␣␣␣␣#␣The␣following␣is␣the␣function␣called␣to␣create␣mydata,` `␣giving␣the␣initial␣value␣of␣the␣mydata␣as␣a␣parameter`	This is an
8	`␣␣␣␣`	initialization
9	`␣␣␣␣def␣__init__(self,␣data):`	method that takes
10	`␣␣␣␣␣␣␣␣#␣__number␣is␣a␣serial␣number␣given␣to␣the␣instance`	parameter data
11	`␣␣␣␣␣␣␣␣self.__number␣=␣MyClass.__b`	
12	`␣␣␣␣␣␣␣␣self.mydata␣=␣data`	
13	`␣␣␣␣␣␣␣␣print("MyClass␣Object␣is␣created,␣number:␣",␣` `self.__number)`	
14	`␣␣␣␣#␣Increase␣the␣value␣of␣the␣class␣variable␣by␣1`	
15	`␣␣␣␣␣␣␣␣MyClass.__b␣+=␣1`	
16		
17	`␣␣␣␣#␣Method␣that␣displays␣the␣serial␣number␣␣␣␣`	
18	`␣␣␣␣def␣show_number(self):`	
19	`␣␣␣␣␣␣␣␣print(self.__number)`	
20		
21	`#`	
22	`#␣Main␣program␣from␣here`	
23	`#`	
24	`if␣__name__␣==␣"__main__":`	Does not execute
25	`␣␣␣␣print("Class␣Variable␣a␣of␣MyClass:␣",MyClass.a)`	when imported as a
26		module, according
27	`␣␣␣␣instance1␣=␣MyClass(1)`	to the instructions
28	`␣␣␣␣instance2␣=␣MyClass(10)`	in line 24
29		
30	`␣␣␣␣instance1.show_number()`	
31	`␣␣␣␣instance2.show_number()`	
32		
33	`␣␣␣␣print("mydata␣of␣instance1:␣",␣instance1.mydata)`	

```
␣␣␣␣print("mydata␣of␣instance2:␣",␣instance2.mydata)
␣␣␣␣instance1.mydata␣+=␣1
␣␣␣␣instance2.mydata␣+=␣2
␣␣␣␣print("mydata␣of␣instance1:␣",␣instance1.mydata)
␣␣␣␣print("mydata␣of␣instance2:␣",␣instance2.mydata)
```

If you run this program, you will get the following. You can see that the instances are numbered using class variables, that the instance variable mydata is independent for each instance, and that it can be accessed directly from the main program.

```
Class Variable a of MyClass:  My Class
MyClass Object is created, number:  0
MyClass Object is created, number:  1
0
1
mydata of instance1:  1
mydata of instance2:  10
mydata of instance1:  2
mydata of instance2:  12
```

Also, the following operation will cause an error in the shell. You can see that instance variables that start with "__" are protected.

```
>>> print(instance1.__number)
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    print(instance1.__number)
AttributeError: 'MyClass' object has no attribute '__number'
```

Note that the line:

```
if __name__ == "__main__":
```

in the source code indicates that this code will only be executed when the current file is run as the main program. It is possible to import this source code as a module, but in that case, anything below this line will not be executed.

Next Chapter Table of Contents

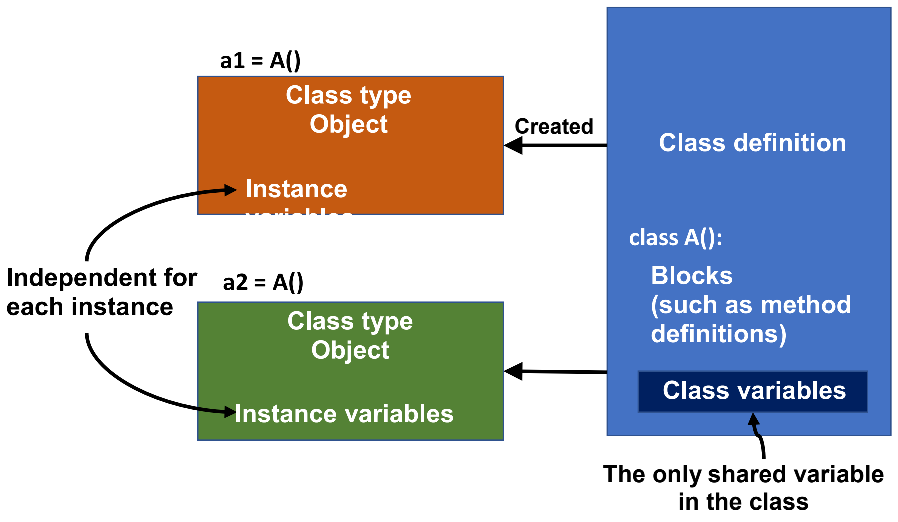


Figure 11-1 Class and Instance Variables

11.5 Inheritance

Inheritance is an important aspect of writing programs using classes. For example, in the example of tkinter implementation, the MyFrame class we defined inherited from tkinter's Frame class. MyFrame inherits the functionality of the Frame class and adds the definition of widgets on the Frame.

11.6 Designing classes starting from instances

Classes are a powerful tool for creating programs that do complex things. As we saw earlier, a class is a "type" that creates objects of a class type (instances). However, **it is difficult to think of things in terms of types**.

When actually designing a class, it is best to think in terms of specific instances, as shown below.

- Consider the data (candidates for instance variables) and operations (candidates for methods) that you want to handle together as a single object.
- Consider a class for each such object.
- Make the classes more widely available.
  - If multiple objects (classes) are the same, then the same class can be used for those objects.
  - If the only difference is values that are set, consider assigning the value to an instance variable and giving it as an argument when the object is created.
  - If shared methods and individual methods are mixed, consider inheritance.

Next Chapter Table of Contents

12. File Input/Output

12.1 Learning goals of this chapter

1. Learn about text files as they are used in Python.
2. Learn how to use spreadsheet software dealing with CSV files to handle the results of Python calculations.
3. Learn how to read and write text files in Python.
4. Learn about filedialog in tkinter to assist in selecting files.

12.2 How to store data permanently

In conventional programs, data set to variables in the program is retained only while the program is running, and is erased when the program is terminated.

In addition, the input and output of a program, whether GUI or CUI, is input by a human, and the results are read by a human.

In order to use data permanently in a program, it is necessary to write and read data in a form that can be saved outside the program. Candidates for this are as follows:

- A file on the computer
- A database on a computer
- A service on a network

In this section, you will learn how to handle files on a computer, which is the basis of all of these methods.

12.3 Regarding files

12.3.1 File Path

Files on computers are managed by operating systems such as Windows, macOS, and linux. These operating systems have a hierarchical folder (directory) [1] structure, in which folders can be placed within folders, and the location of a file is identified by its position in the folder structure. The string describing the location of a file is called the "file path," and is a combination of the "hierarchical folder structure notation" and the file name.

To give an example, in Windows

[1]Windows uses folders as a mechanism for organizing files, while its predecessors, MS-DOS and unix, used the term "directory." Strictly speaking, they are slightly different, but here we will treat folders and directories as the same thing.

M:¥documents¥python scripts¥p3-1.py

is the file path. Depending on a used font, it may be represented as

M:\documents\python scripts\p3-1.py

Here

M: the drive name (corresponding to the disk device or file server)

¥document¥python_scripts: folder path

p3-1.py: file name

.py: everything after the "." in a file name is called the **"extension,"** which indicates the file type.

A full path starts at the drive name and includes all of the folders. A full path is the only way to identify a specific file on a computer.

In addition to this, there is a "current working folder [directory]" that is referred to as the current working directory, (cwd), as well as a "relative path" which is a path that describes only the differences from the specified folder. For example, if your working folder is

M:¥documents¥python scripts

then the relative path notation

p3-1.py

would refer to

M:¥documents¥python scripts¥p3-1.py

12.3.2 Use of Raw String

When you write file path directly in a source code, you should know Python treats '¥' for special meaning, e.g, '¥n' means newline. Since a filepath often uses '¥' for separator of folders, you have to write two '¥'s for each '¥'

```
filepath = "M:¥¥documents¥¥python scripts¥¥p3-1.py"
```

or use prefix 'r' to treat the string as it is:

```
filepath = r"M:¥documents¥python scripts¥p3-1.py"
```

Reference the column titled " Escaping."

12.3.3 Text Files

A text file is a file written in text code (and symbols such as line breaks) in a format that can be read and written by humans using an editor. For example, Python source code and email messages are text files.

On the other hand, a file consisting of data in a computer's internal format is called a “binary file.” The word “binary” means “base-2.” Binary files are written in the computer's internal format with no change; they have the advantage of not losing numerical value precision, and taking up a small amount of data. However, without a description of the file's contents, it’s not possible to tell what is written in a binary file.

In this chapter, you will learn how to read and write text files in Python.

12.3.4 CSV format

Python programs can be used in conjunction with other tools without much effort. The **CSV (comma separated value) format** is an easy way to handle data for this purpose. This is a type of text file in which each line consists of:

    Data entry 1, Data entry 2, Data entry 3

and so on, with commas separating the data. If you add the extension .csv to the file name of a file in this format, it can be read by spreadsheet software such as Microsoft Excel, making it easy to create graphs.

Data in the CSV format is **relatively easy to output**. On the other hand, reading data in the CSV format data can be tricky due to the handling of comma and line break characters; depending on the content of your data, you may want to consider using a library[1] or other method.

12.3.5 Character Encoding Issues

Due to the historical nature of the Japanese language, there are several different character encodings that are used on different operating systems. For example, the character codes used for Japanese file names are as follows

• Mac, Linux: Unicode

• Windows: Shift-JIS

In text files, in addition to the above differences in character encoding, there are also differences in the code used to represent a line break.

- Python 3 uses UTF-8 internally, which is one of the Unicode representations. Python programs (scripts) created in IDLE are coded and stored in UTF-8 as well. If you are running Python on a single OS, you don't need to worry too much about the differences between OSes, as Python will adjust accordingly. You need to be careful when running the program on different operating systems, however.

[1]In Python, the csv module is a library for handling CSV. In a later chapter we will see an example of loading a CSV file with pandas.

12.3.6 Error Handling

Error handling is extremely important for file input/output. This is because the contents of files, the file system, and the data to be read cannot be controlled by the program. When you try to open a file, you must be aware that various things may happen. For example, the file or folder may not exist, you may not have write permission, or you may run out of disk space in the middle of writing a file.

12.4 Let's try to run the code below first.

12.4.1 Source code

Program 12-1 Example of File Input/Output (p12-1_en.py)

Row	Source code	Explanation
1	`#␣Import␣the␣OS␣module␣to␣find␣out␣the␣current␣working␣directory␣(the␣folder␣where␣you␣are␣working)`	
2	`import␣os`	
3		
4	`#␣Get␣the␣current␣working␣directory␣and␣print␣it␣on␣the␣screen`	
5	`print(os.getcwd())`	
6	`#␣Create␣a␣file␣named␣'Japanese␣file.txt'␣and␣write␣fill␣it␣in`	
7	`f␣=␣open('Japanese-File.txt','w')`	
8	`f.write('日本語\n日本語\n日本語\n')`	Write text in Kanji. On Windows, "\" may be shown as "¥"
9	`f.close()`	
10	`#␣Open␣Japanese␣file.txt␣for␣reading␣to␣display␣its␣contents`	
11	`f␣=␣open('Japanese-File.txt','r')`	
12	`s␣=␣f.read()`	
13	`f.close()`	
14	`print(s)`	

12.4.2 Program Notes

- Figure out the current working folder (current working directory) (line 8)
- Open the file named "Japanese file.txt" for writing (w), and assign it to the variable "f" for future use. It is written as a relative path, so it will be created in the working folder with this name. (line 10)
- Write the string to a file (line 11). "\n" means "new line."
- Close the writing file (line 12)
- Open a file with the same name for reading (r). (line 15)
- Assign the entire contents of the file to the variable s. (Line 16)
- Close the file (line 17)

- Output the data (text) (line 18)

12.5 Reading and writing files in Python

12.5.1 Using the open Function

Files can be manipulated as per the following procedure.

1. Open the file with the open function and get the file object in the return value.

   ```
   file = open(file name,mode)
   ```

   The mode can be "r" for reading, "w" for writing, etc. In the above example, the return value is assigned to the variable file.
   In Python, unless otherwise specified, text files are written in the standard character set of the operating system, and the encoding argument can be used to explicitly specify the character set.

   ```
   file = open(file name,mode, encoding= "utf-8")
   ```

   If it fails to open the file, it raises an IOError exception.

2. Reading and writing to a file object

   A. Reading from a file object using the read() method

   ```
   s = file.read()
   ```

   In the above example, the entire text file is read as a string and assigned to the variable s.

   B. Writing to a file object using the write() method

   ```
   file.write(s)
   ```

   The above example writes data in s to the file as a string.

   The data can be added in the same way until the file is closed.

3. Close the file

   ```
   file.close()
   ```

Note: open is a built-in function, and read, write, and close are methods of the file object.

In the above example, the entire contents of the file are read at once; to read a single line, use the readline() method. It is also possible to process the contents of a file line by line using a for statement as follows [1].

```
file = open("filename", "r")
```

[1] Python's for statement can be applied to a variety of objects, such as range() functions, strings (one character at a time), lists (element by element), etc. This is because they are iterable, which means that they can be iterated over. File objects are also iterable, where iterations occur line by line.

```
for line in file:
    A block that works on each line one line at a time
```

12.5.2 Using the with statement - Automating close()

A file opened with the open() function must be closed with the close() method, but in the following cases, the close() function may not be performed.

- You simply forgot to call the close() method.
- The close() method is not executed due to an error in the code where it is written.

To avoid this, Python provides 'with' statement, which automatically closes the file opened by the with statement after the block ends.

```
with open(File name and other arguments of the open function) as variables for the file object :
        A block that manipulates the file
```

12.6 Example 1 Wave approximation

12.6.1 Key points in the example below

- Since it is cumbersome to input the exact file path from a terminal, tkinter (only the filedialog module) is used.
- The calculation results are output in CSV format and linked to spreadsheet software.
- As an example, we will use an example that expresses a periodic function as a sum of trigonometric functions.

12.6.2 Approximation of Periodic Function by Sum of Trigonometric Functions

It is known that a periodic function (a function whose value repeats with a certain period) can be approximated by the sum of sine (sin) and cosine (cos) functions that are integer multiples of the frequency. A sawtooth wave can be approximated as follows (see also the column titled “Trigonometric Functions”).

$$f(x) = \frac{\sin(x)}{1} + \frac{\sin(2x)}{2} + \frac{\sin(3x)}{3} + \frac{\sin(4x)}{4} \cdots$$

The figure below plots the sum of the first through fifth terms.

Next Chapter Table of Contents

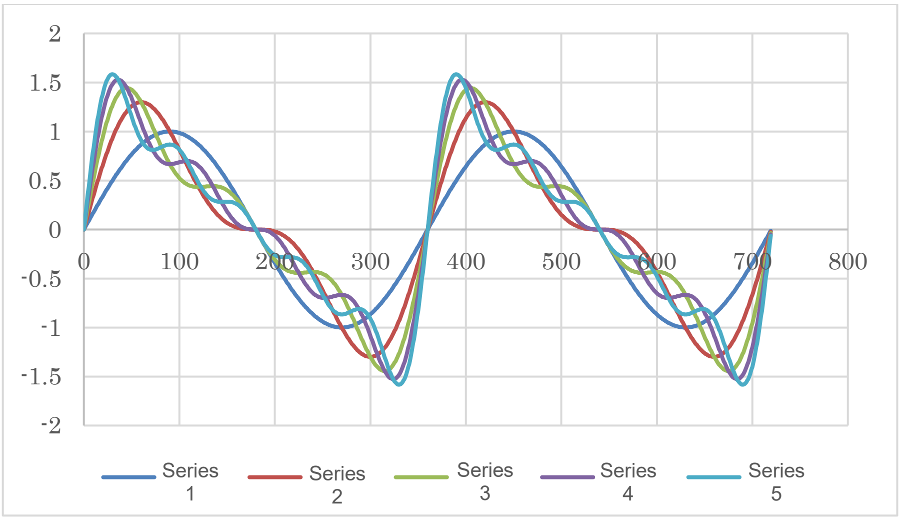


Figure 12-1 Approximation of a sawtooth wave by summing trigonometric functions

For simplicity, the signs of the terms shown above are all the same, but for the sawtooth wave (that has a positive slope at the origin), the signs change alternately as shown below.

$$f(x) = \frac{\sin(x)}{1} - \frac{\sin(2x)}{2} + \frac{\sin(3x)}{3} - \frac{\sin(4x)}{4} \cdots$$

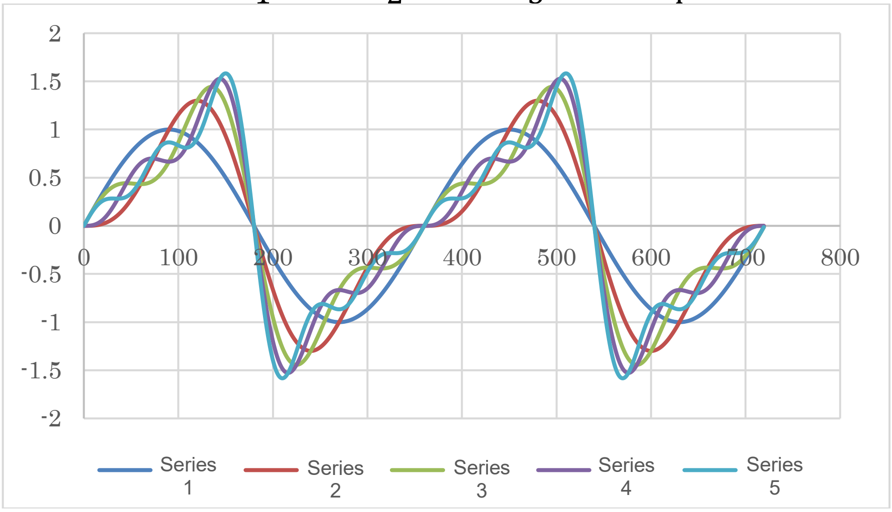


Figure 12-2 Approximation of a sawtooth wave by summing trigonometric functions (with positive slope at the origin)

A sawtooth wave with an amplitude of 1 (maximum and minimum values are ±1) has a coefficient (2/π) applied to the whole wave.

Next Chapter Table of Contents

12.6.3 Source code

Program 12-2 Approximation of a sawtooth wave by summing trigonometric functions (p12-2.py)

Row	Source code	Explanation
1	import␣tkinter␣as␣tk	
2	import␣tkinter.filedialog	Import filedialog as
3	import␣math	well
4	#	
5	#␣Example␣of␣using␣only␣tkinter's␣filedialog	
6	#	
7	#␣Hide␣the␣root␣window␣by␣reading␣the␣withdraw()␣method	
8	root␣=␣tk.Tk()	
9	root.withdraw()	
10	#	
11	#␣Read␣the␣filedialog␣for␣writing␣and␣get␣the␣file␣name.	
12	#	
13	filename␣=␣tkinter.filedialog.asksaveasfilename()	Get the file name
14	#	from the dialog and
15	#␣Close␣tkinter	go back.
16	#	tkinter is no longer
17	root.destroy()	used, so it is
18	#	closed.
19	#␣If␣you␣don't␣get␣a␣file␣name,␣close␣it	
20	#	
21	if␣filename:	
22	␣␣␣␣pass	pass is a command
23	else:	that does not do
24	␣␣␣␣print("No␣file␣specified")	anything.
25	␣␣␣␣exit()␣␣␣␣	
26	#	
27	#␣Approximate␣a␣saw␣wave␣by␣superimposing␣sine␣waves␣upon␣one␣another	
28	#	
29	#␣w␣=␣sin(t)␣+␣sin(2t)/2␣+␣sin(3t)/3␣+␣sin(4t)/4␣...	
30	#	
31	#␣2␣periods␣worth,␣1000␣steps␣overall,␣harmonics␣up␣to␣the␣5th	
32	#	
33	cycles␣=␣2	
34	steps␣=␣1000	
35	harmonics␣=␣5	
36	#␣Error␣handling␣when␣a␣file␣cannot␣be␣opened	
37	try:	
38	#␣Open␣the␣file	
39	␣␣␣␣with␣open(filename,'w')␣as␣file:	
40	␣␣␣␣␣␣␣␣for␣i␣in␣range(steps):	

41	␣␣␣␣␣␣␣␣␣␣␣␣angle_in_degree␣=␣360*cycles*i/steps	
42	␣␣␣␣␣␣␣␣␣␣␣␣angle␣=␣math.radians(angle_in_degree)	
43	␣␣␣␣␣␣␣␣␣␣␣␣s␣=␣str(angle_in_degree)	Express the angle as
44	␣␣␣␣␣␣␣␣␣␣␣␣w␣=␣0	a string
45	␣␣␣␣␣␣␣␣␣␣␣␣for␣j␣in␣range(1,harmonics+1):	
46	␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣w␣+=␣math.sin(angle*j)/j	Concatenate sum w to
47	␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣s␣=␣s+”,␣”+␣str(w)	s, separated by a
48	␣␣␣␣␣␣␣␣␣␣␣␣#print(s)	comma (,)
49	␣␣␣␣␣␣␣␣␣␣␣␣file.write(s+”\n”)	Add a line break
50	␣␣␣␣␣␣␣␣print(”Writing␣to␣file␣”+␣filename␣+␣”␣is␣finished”)	”\n” and write it Append s to the
51	except␣IOError:	file.
52	␣␣␣␣print(”Unable␣to␣open␣file”)	

12.6.4 Program Notes

1) Use of tkinter's filedialog (lines 1-25)

- When you open or save a file in a Windows application, you can find or specify the file in a separate window. tkinter provides a filedialog mechanism for this purpose. In this program, the main window of tkinter is created in line 8. Because we use only the filedialog function, the main window of tkinter is hided in line 9. Also note that the mainloop() method is not called.
- There are several formats for filedialog depending on the intended use, but in this case, the asksaveasfilename() method for "save as" is called in line 13, and the filename is set to the file name (path name) obtained by the return value.
- Since tkinter is not needed when coming back from filedialog, root.destroy() terminates tkinter.
- If the user does something such as cancel the program, the filename will be empty, so the program will be terminated when the if statement is false.
- For more information on using filedialog for reading, refer to the following example.

2) Calculation and file output (lines 23 to 53)

- This part calculates the superposition of trigonometric functions and outputs it in CSV format.
- For file handling, it is necessary to deal with an error such as when the program is not able to open the file. In line 37, a block that manipulates the file with a try statement handles this. The corresponding error handling is in lines 52 and 53.
- In line 40, a with statement is used to open a file with filename obtained from filedialog, and the opened file is handled by the variable file.
- One line of computation and output is as follows:
  angle, first term, sum up to second term, sum up to third term, sum up to fourth term, sum up to fifth term.

The summation is computed in the variable w, and the contents of a line are added to the variable s as a string. To make the CSV format, the value of w is converted to a string and concatenated to s separated by a comma (,) as follows:

```
s = s+", "+ str(w)
```

The comma is followed by a [1] space to make the resulting file easier to read.

- After the computation up to the fifth term in the for statement is completed, lines 49 and 50 are output to a file.

```
#       print(s)
file.write(s+"\n")
```

Line 49 is commented, but if you want to see the result in the Python shell, remove the #. In line 50, file.write() writes to the file, but "¥n" is added to the single-line string s to add a "line break." Mac users should enter a backslash "¥" instead of "¥."

Exercise 12-1 Square wave approximation

A square wave (a periodic function that alternates between values of ±1) can be approximated by a trigonometric function as follows [2]

$$f(x) = \frac{\sin(x)}{1} + \frac{\sin(3x)}{3} + \frac{\sin(5x)}{5} + \frac{\sin(7x)}{7} \cdots$$

Calculate the trigonometric approximation for the square wave in the same way as in the example, output the result to a CSV file, and create a graph using spreadsheet software.

Exercise 12-2 Implementation of the list from Example 1

In the program in Example 1, the results of the calculations were concatenated as strings and written to a file one line at a time. Reimplement this program as follows, separating the calculation and output.

- Write the result of the calculation to a list using a list.
- After the calculation is completed, write a CSV file in the same format as in Example 1 that refers to the list.

There are two ways of constructing a list, as follows. Either implementation method is acceptable.

[1] Turing part of a program that could otherwise be executed into a comment is called "commenting out," and is often used to check the operation of a program.

[2] For a square wave of amplitude 1, a factor of 4/π is applied to the whole wave.

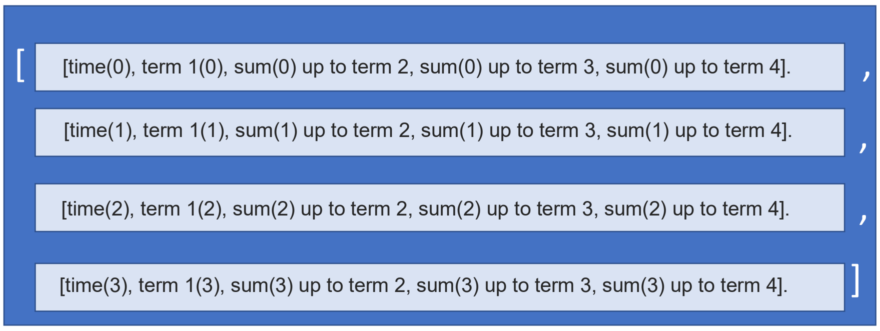

[
[time(0), term 1(0), sum(0) up to term 2, sum(0) up to term 3, sum(0) up to term 4].
,
[time(1), term 1(1), sum(1) up to term 2, sum(1) up to term 3, sum(1) up to term 4].
,
[time(2), term 1(2), sum(2) up to term 2, sum(2) up to term 3, sum(2) up to term 4].
,
[time(3), term 1(3), sum(3) up to term 2, sum(3) up to term 3, sum(3) up to term 4].
]

Figure 12-3 Using a "list of data at different time points"

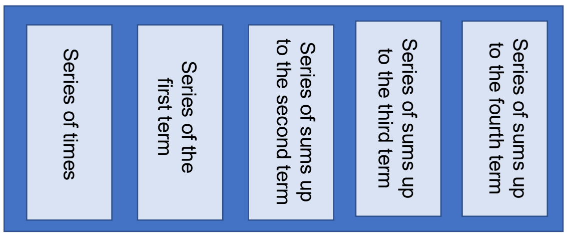

Series of times
Series of the first term
Series of sums up to the second term
Series of sums up to the third term
Series of sums up to the fourth term

Figure 12-4 Using a "list of each series'"

12.7 Example 2 Text Editor

You can create a simple text editor using tkinter. tkinter has all the features you have learned so far, as well as the messagebox dialog for displaying messages, the filedialog method for reading files, the tkinter Menu widget, Text widget, etc. For the geometry manager, we will use 'pack' instead of 'grid' for simplicity.

The Kanji encoding for the files is not specified, so Python assumes the standard encoding for each OS. On Windows, it is assumed that Shift-JIS code (cp932) is used.

Program 12-3 A simple text editor using tkinter (p12-3.py)

```
import␣tkinter␣as␣tk
import␣tkinter.messagebox
import␣tkinter.filedialog
#␣messagebox␣and␣filedialog␣need␣to␣be␣imported␣explicitly
#
#␣Create␣a␣class␣called␣MyFrame␣that␣inherits␣from␣tk.Frame
#␣Set␣up␣widgets␣and␣callback␣functions␣(methods)␣in␣it.
#␣This␣is␣the␣standard␣way␣to␣use␣tkinter.
#
class␣MyFrame(tk.Frame):
#␣__init__␣is␣the␣initialization␣method␣for␣creating␣a␣class␣object
␣␣␣␣def␣__init__(self,␣master␣=␣None):
```

```
␣␣␣␣␣␣␣␣super().__init__(master)
␣␣␣␣␣␣␣␣self.master.title(’Simple␣Editor’)

#␣Create␣a␣menu:␣menubar␣->␣filemenu␣->␣Open,␣Save␣as,␣Exit
␣␣␣␣␣␣␣␣menubar␣=␣tk.Menu(self)
␣␣␣␣␣␣␣␣filemenu␣=␣tk.Menu(menubar,␣tearoff␣=␣0)
␣␣␣␣␣␣␣␣filemenu.add_command(label␣=␣”Open”,␣command␣=␣self.openfile)
␣␣␣␣␣␣␣␣filemenu.add_command(label␣=␣”Save␣as...”,␣command␣=␣self.saveas)
␣␣␣␣␣␣␣␣filemenu.add_command(label␣=␣”Exit”,␣command␣=␣self.master.destroy)
␣␣␣␣␣␣␣␣menubar.add_cascade(label␣=␣”File”,␣menu␣=␣filemenu)
␣␣␣␣␣␣␣␣self.master.config(menu␣=␣menubar)

#␣Create␣a␣Text␣widget␣for␣editing␣as␣a␣class␣variable␣using␣editbox
␣␣␣␣␣␣␣␣self.editbox␣=␣tk.Text(self)
␣␣␣␣␣␣␣␣self.editbox.pack()

#␣Method␣to␣open␣a␣file,␣requires␣the␣parameter␣”self”␣
which␣is␣different␣than␣the␣parameter␣that␣a␣function␣would␣take
␣␣␣␣def␣openfile(self):
#␣Get␣the␣file␣name␣in␣filedialog
␣␣␣␣␣␣␣␣filename␣=␣tkinter.filedialog.askopenfilename()
#␣Process␣the␣filename␣if␣it’s␣not␣empty
␣␣␣␣␣␣␣␣if␣filename:
␣␣␣␣␣␣␣␣␣␣␣␣tkinter.messagebox.showinfo(”Filename”,”Open:␣”+filename)
#␣Open␣a␣file␣with␣a␣variable␣named␣file␣in␣it␣using␣a␣with␣statement
␣␣␣␣␣␣␣␣␣␣␣␣with␣open(filename,’r’)␣as␣file:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣text␣=␣file.read()
#␣Set␣the␣file␣contents␣in␣the␣editbox␣Text␣widget
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.editbox.delete(’1.0’,tk.END)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.editbox.insert(’1.0’,text)
␣␣␣␣␣␣␣␣else:
␣␣␣␣␣␣␣␣␣␣␣␣tkinter.messagebox.showinfo(”Filename”,”Canceled”)

#␣Method␣to␣save␣to␣a␣file
␣␣␣␣def␣saveas(self):
#␣open␣a␣file␣with␣a␣variable␣named␣file␣using␣a␣with␣statement
␣␣␣␣␣␣␣␣filename␣=␣tkinter.filedialog.asksaveasfilename()
␣␣␣␣␣␣␣␣if␣filename:
␣␣␣␣␣␣␣␣␣␣␣␣with␣open(filename,’w’)␣as␣file:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣text␣=␣file.write(self.editbox.get(’1.0’,tk.END))
␣␣␣␣␣␣␣␣␣␣␣␣tkinter.messagebox.showinfo(”Filename”,”Saved␣AS:”+filename)
␣␣␣␣␣␣␣␣else:
␣␣␣␣␣␣␣␣␣␣␣␣tkinter.messagebox.showinfo(”Filename”,”Canceled”)

#␣Main␣program␣from␣here
root␣=␣tk.Tk()
f␣=␣MyFrame(root)
f.pack()
```

Next Chapter Table of Contents

```
f.mainloop()
```

13. Learning Program Development with Tic-Tac-Toe

13.1 Learning goals of this chapter

In this chapter, you will do the following to learn how to develop a program, using tic-tac-toe as an example.

1. Analyze how to play tic-tac-toe, and identify what needs to be expressed in the program.
2. Prepare a game record for testing the program.
3. Create the data and functions that make up the program, starting with the smallest.
4. Assemble the whole program to complete the tic-tac-toe program.

13.2 Developing a program

As seen in the previous programs, it is difficult for beginners to create a program when given a specific task. This is because being able to use the various elements of a programming language and developing a coherent program from scratch are two different skills. For example, **just because you can use a hammer and a saw doesn't mean you can build a house**, because to build a house, you need to know what it consists of and in what order it needs to be designed and constructed. The same applies to programs.

13.3 Design procedure - what to do before using your computer

You often see beginners open their computer to create a program only to end up getting stuck. Why does that happen?

The procedure for designing and creating a program is as follows. You don't necessarily need the computer at first, but because you are facing the computer screen, you may get stuck.

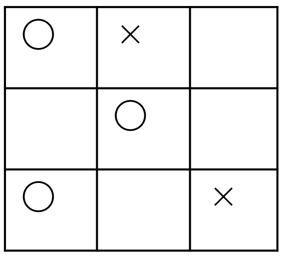
- **What to do before using a computer**
    - ➢ Describe what you want to achieve in words
    - ➢ Identify what you need the program to do
        - ✧ What to express as variables
        - ✧ What values each variable will have

        - What should be expressed as a protocol (function)
        - What should be expressed as an interaction with people
    - Decide the order in which to create the program
    - Decide how to test it
- **This is where the work on the computer begins**
    - Create the program (functions) starting with the parts that do not depend on others
    - Test the functions you create (unit testing)
    - Test the whole program (integration testing)

13.4 Designing a simple tic-tac-toe program

13.4.1 Tic-Tac-Toe

Describe in words the rules of tic-tac-toe and how the game proceeds.



13.4.2 Sentence analysis

Analyze the sentences you made that describe the rules of tic-tac-toe and the progression of the game in terms of parts of speech (nouns, copulas, verbs).[1] You will end up with points similar to those below.

- Items (nouns) that take a specific state: these are candidates for variables
    - 3×3 board, turn
- The state of items (copulas): possible values for the variables
    - The state of each square (empty, O (player one), X (player two))
    - Whose turn it is
- Actions that check states (candidates for functions)
    - Whose turn it is
    - State of the squares

[1]This existence of such tasks is why literacy in the humanities is important for programmers.

  - Player one wins, player two wins, draw
- Changing the state of a square (this is a candidate for a function)
  - Placing O or X in a square
  - Alternating turns

13.4.3 Creating a Game Board Record (to Prepare for Testing)

Before you build your program, you should make some test board game records. It is difficult to cover all cases, but here are some cases to consider.

- Must include cases where player one wins, player two wins, and there is a draw
- Include all the different winning patterns (vertical (3 ways), horizontal (3 ways), diagonal (2 ways))

Preparing test cases first is called "**test first programming;**" it is more effective than "coding first," which is where you start building the program first, in the following reasons

- You're less likely to create test cases if you start later
- You can test these cases at any time during the creation of the program
- Test cases can help you to be aware of the need to test during coding

	Turn	row	column			row	column			row	column		
1	○	0	0			0	0			0	1		
2	×	1	1			1	0			0	0		
3	○	1	0			1	1			2	1		
4	×	2	0			2	2			1	1		
5	○	0	2			0	2			2	2		
6	×	0	1			0	1			2	0		
7	○	2	1			2	0			1	0		
8	×	2	2							0	2		
9	○	1	2										
Result		Draw				Player one wins				Player two wins			
			0	1	2		0	1	2		0	1	2
		0	1○	6×	5○	0	1○	6×	5○	0	2×	1○	8×
		1	3○	2×	9○	1	2×	3○		1	7○	4×	
		2	4×	7○	8×	2	7○		4×	2	6×	3○	5○

Figure 13-1 Example tic-tac-toe board records

13.4.4 Variable design

1) The board

- The 3×3 board is represented by a nested list in which the elements are integers.
  board = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
- The initial state is all 0s (empty)
- The meaning of the values is as follows: 0 is an empty space, 1 is for player one (O), and 2 for player two (X).
- For this purpose, we define constants (there is no way to forbid changing the values in Python). We use uppercase letters to make it clear that they are constants. we also use the following constants to consider the turn and result.

```
OPEN = 0

FIRST = 1

SECOND = 2

DRAW = 3
```

2) Turn

- An integer variable will indicate which player's turn it is. The initial value is FIRST (the first move)

```
turn = FIRST
```

- The values use the previous constants (FIRST, SECOND), where FIRST indicates that it is player one's turn, and SECOND indicates that it is player two's turn

3) Game board record

- The turn order is always "player one, then player two, then player one..." The move made by each player is represented in the list of rows and columns.
- We then append the result (undecided, player one wins, player two wins, draw) to this list [1].
- Board Record = [[first move (row, column)], [second move (row, column)], ..., [last move (row, column)], [result]]
- The row and column can be represented by an integer value of either 0, 1, or 2. The result can be represented by an integer value of 0, 1, 2, or 3.

13.4.5 Functions concerning the board and turns

Create functions to "operate" on the state, "check" the state, "display" it on the screen, initialize it (a

[1] Because the board record is represented by a single list, different list elements actually represent two different things (where the players moved, and the result). This is admittedly not a very straightforward implementation.

type of operation), etc.

The board and the turn are shared as global variables, so a global declaration is required to change the values of variables outside the function from within a function. Functions such as those below may be needed.

1) Turns

- Operation: Initialize the turn
- Operation: Change whose turn it is
- Display: Display the turn (generate a string for it)

2) The Board

- Operation: Initialize the board
- Operation: Mark the specified square on the board with the marker of the player whose turn it is
- Check: Determine the state of individual squares on the board
- Check: Determine which player wins the game, based on the board
- Check: Determine if all squares on the board are occupied
- Display: Output the entire board (generate it as a string)

3) Board Records

- Operation: Replay a game using the board record.

4) Algorithm for determining the winner

If you try to explicitly write out a method of determining the winner of a game of tic-tac-toe as you play it in your daily life, it would be as follows.

- Determine whether the turn (denoted by t) results in victory in a given row or column.
    - If all the spots in the three positions in row/column of interest are t, then t wins.
    - Otherwise, t does not win
- Determine if the game is won in a certain direction (horizontal, vertical, diagonal, reverse diagonal)
    - If horizontal or vertical
        - if the game is won in row 0 (column 0), t wins
        - If not, then if the game is won in row 1 (column 1), t wins
        - If not, then if the game is won in row 2 (column 2), t wins
        - Otherwise, t does not win
    - If the game is won in the diagonal or reverse diagonal line, then t wins

- Determining victory based on the above
  1. If the game is won horizontally, then t wins.
  2. If not, then if the game is won vertically, t wins.
  3. If not, then if the game is won diagonally, t wins.
  4. If not, then if the game is won reverse diagonally, t wins.
  5. Otherwise, t does not win

Also, while it is not possible on a board where victory is checked for every turn, on a randomly generated board, it is possible to have a case where both player one and player two win.

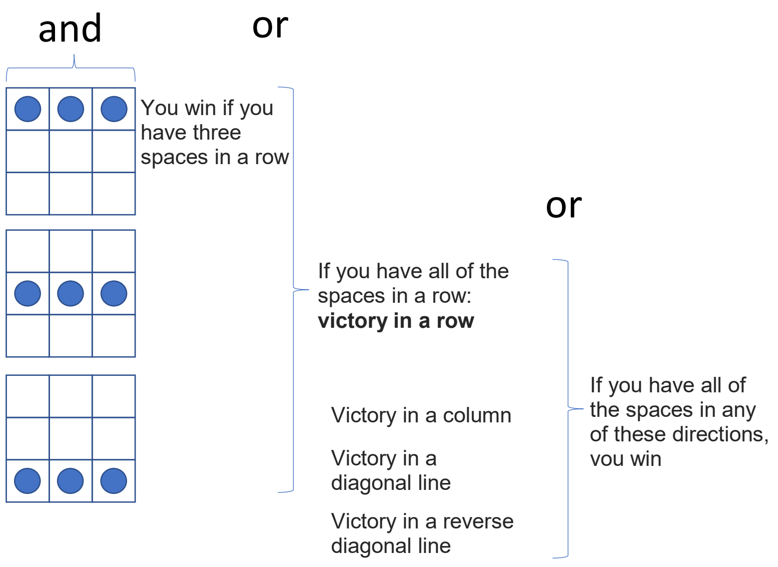

and
or
You win if you have three spaces in a row
or
If you have all of the spaces in a row: **victory in a row**
Victory in a column
Victory in a diagonal line
Victory in a reverse diagonal line
If you have all of the spaces in any of these directions, you win

Figure 13-2 Determining victory for the turn in question

- Determining the winner
  Using the above procedure (function), you can determine the winner of a game of tic-tac-toe as follows.

1. Check whether player one has won or not; if player one has won, then it is player one’s victory.
2. If not, then check whether player two has won or not; if player two has won, then it is player two’s victory.
3. If not, then if there is still an empty space on the board, then the game is unfinished.
4. If not (if there are no empty spaces on the board), the game is a draw.

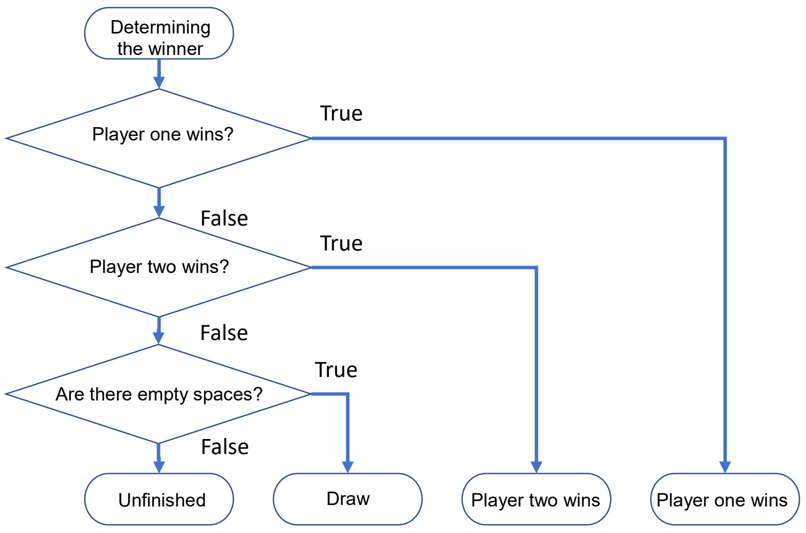


Figure 13-3 Determining the winner

13.4.6 How to write complex conditional decisions

As explained in Section 4, the determination of the winner of a tic-tac-toe game is actually quite complicated. For example, suppose you have three conditions: is_A, is_B, and is_C (think of these as variables or functions that take a value of True or False). The part of the function that decides if all of them are True is written as:

```
return is_A and is_B and is_C:
```

You could also nest the if statements as follows:

```
if is_A:
    if is_B:
        if is_C:
            return True
return False
```

Also, you could write it as:

```
if not is_A:
    return False
if not is_B:
    return False
if not is_C:
    return False
return True
```

If you use that last way of writing it, then you could also write it as:

```
conditions = [is_A, is_B, is_C]
for c in conditions:
  if not c:
    return False
return True
```

In the last method, even if the number of conditions to be checked increases, the decision part can be written succinctly by using a for statement.

Exercise 13-1 Programming Branching with Complicated Condition

Based on the examples above, write a function that returns True if any of is_A, is_B, or is_C is True.

13.4.7 Progression of the game

Let's consider the flow of the main program. Input and output are done in the Python shell in the form of characters. You'll see how easy it is to implement using the functions you defined earlier.

- Initialize the game
- Display the board
- Repeat the following until the game is won, lost, or drawn:
  - Prompt the user for the input of the active player until a valid input is obtained.
    - Get input from the active player
  - Update the board
  - Display the board
  - Determine if the game is won, lost, or drawn
    - Show the result and escape the main loop if the game is over
  - Change the player's turn

13.5 Implementation of the program

1) Source code structure

Once the design is complete, you can start fleshing out the program. Check to see that your Python source code looks something like the following.

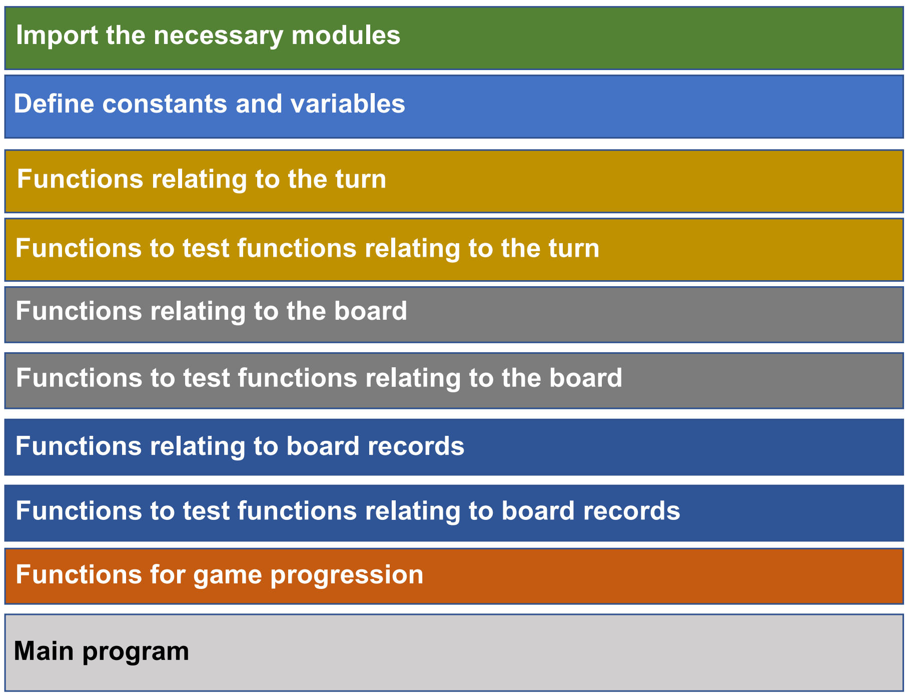

Import the necessary modules
Define constants and variables
Functions relating to the turn
Functions to test functions relating to the turn
Functions relating to the board
Functions to test functions relating to the board
Functions relating to board records
Functions to test functions relating to board records
Functions for game progression
Main program

Figure 13-4 Overall source code structure

2) Example code (tic_tac_toe_en.py)

An example implementation of the program is shown below. Some notes concerning the way things were implemented and the notation are explained below.

- The program is implemented in the order shown in Figure 13-4.
- The program has nearly 500 lines in total, but the code with the yellow background is for testing functions.
- Functions come with a docstring. Multi-line docstrings start and end with ''' (three single quotes).
- There are no modules that need to be imported.
- The display functions do not "print to the screen," but generate a string that can be passed to the print() function.
- The main program simply prints "tic-tac-toe." All the functions are loaded, so you can call them in Python shell for testing or for actual play.

Program 13-1 Tic-tac-toe program example (Part 1: Global variables)

Row	Source code
1	#
2	#␣Tic-tac-toe
3	#
4	#␣There␣are␣no␣modules␣that␣need␣to␣be␣imported.
5	#
6	#␣Define␣constants
7	#
8	#
9	#␣Create␣a␣game␣record␣in␣play()␣(needs␣to␣be␣completed).
10	#
11	'This␣is␣a␣tic-tac-toe␣program'
12	OPEN␣=␣0
13	FIRST␣=␣1
14	SECOND␣=␣2
15	DRAW␣=␣3
16	#
17	#␣Constant␣variable
18	#
19	turn␣=␣1
20	board␣=␣[[0,0,0],[0,0,0],[0,0,0]]
21	#
22	#␣Test␣board␣records
23	#
24	log1␣=␣[[0,␣0],␣[1,␣1],␣[1,␣0],␣[2,␣0],␣[0,␣2],␣[0,␣1],␣[2,␣1],␣[2,␣2],␣[1,␣2],␣[DRAW]]
25	log2␣=␣[[0,␣0],␣[1,␣0],␣[1,␣1],␣[2,␣2],␣[0,␣1],␣[2,␣0],[FIRST]]
26	log3␣=␣[[0,␣1],␣[0,␣0],␣[2,␣1],␣[1,␣1],␣[2,␣2],␣[2,␣0],␣[1,␣0],␣[0,␣2],[SECOND]]

Program 13-2 Tic-tac-toe program example (Part 2: Turn-related functions)

```
#
#␣Functions␣related␣to␣turns
#
#␣Convert␣the␣move␣number␣to␣a␣string
#
def␣show_turn():
␣␣␣␣'Return␣a␣string␣showing␣current␣turn'
␣␣␣␣if␣turn␣==␣FIRST:
␣␣␣␣␣␣␣␣return('First')
␣␣␣␣elif␣turn␣==␣SECOND:
␣␣␣␣␣␣␣␣return('Second')
␣␣␣␣else:
␣␣␣␣␣␣␣␣return('Vaule␣of␣turn␣is␣not␣adequate')
#
#␣Initialize␣the␣turn
#
def␣init_turn():
␣␣␣␣'Initialize␣turn'
␣␣␣␣global␣turn
␣␣␣␣turn␣=␣1
#
#␣Change␣the␣turn
#
def␣change_turn():
␣␣␣␣'Change␣turn'
␣␣␣␣global␣turn
␣␣␣␣if␣turn␣==␣FIRST:
␣␣␣␣␣␣␣␣turn␣=␣SECOND
␣␣␣␣elif␣turn␣==␣SECOND:
␣␣␣␣␣␣␣␣turn␣=␣FIRST
#
#␣Test␣turn-related␣functions
#
def␣test_turn():
␣␣␣␣'Test␣program␣of␣turn'
␣␣␣␣init_turn()
␣␣␣␣print(show_turn(),"␣is␣the␣current␣turn")
␣␣␣␣change_turn()
␣␣␣␣print(show_turn(),"␣is␣the␣current␣turn")
␣␣␣␣change_turn()
␣␣␣␣print(show_turn(),"␣is␣the␣current␣turn")
```

Program 13-3 Tic-tac-toe program, example (Part 3: Board-related functions part 1)

```
#
#␣Board-related␣functions
#
#␣A␣string␣that␣displays␣the␣board
#
def␣show_board():
␣␣␣␣’Return␣a␣string␣showing␣the␣current␣board’
␣␣␣␣s␣=␣’␣:0␣1␣2\n---------\n’
␣␣␣␣for␣i␣in␣range(3):
␣␣␣␣␣␣␣␣s␣=␣s␣+␣str(i)␣+␣’:␣’
␣␣␣␣␣␣␣␣for␣j␣in␣range(3):
␣␣␣␣␣␣␣␣␣␣␣␣cell␣=␣’’
␣␣␣␣␣␣␣␣␣␣␣␣if␣board[i][j]␣==␣OPEN:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cell␣=␣’␣’
␣␣␣␣␣␣␣␣␣␣␣␣elif␣board[i][j]␣==␣FIRST:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cell␣=␣’O’
␣␣␣␣␣␣␣␣␣␣␣␣elif␣board[i][j]␣==␣SECOND:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cell␣=␣’X’
␣␣␣␣␣␣␣␣␣␣␣␣else:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣cell␣=␣’?’
␣␣␣␣␣␣␣␣␣␣␣␣s␣=␣s␣+␣cell␣+␣’␣’
␣␣␣␣␣␣␣␣s␣=␣s␣+␣’\n’
␣␣␣␣return␣s
#
#␣Initialize␣the␣board
#
def␣init_board():
␣␣␣␣’Set␣all␣the␣places␣on␣board␣OPEN’
␣␣␣␣for␣i␣in␣range(3):
␣␣␣␣␣␣␣␣for␣j␣in␣range(3):
␣␣␣␣␣␣␣␣␣␣␣␣board[i][j]␣=␣OPEN
#
#␣Return␣the␣value␣of␣position␣i,␣j␣on␣the␣board
#
def␣examine_board(i,j):
␣␣␣␣’Return␣state␣of␣the␣i-th␣row␣j-th␣column␣place␣on␣the␣board’
␣␣␣␣return␣board[i][j]
#
#␣Register␣the␣turn␣t␣to␣i,␣j␣on␣the␣board,␣and␣return␣its␣status␣as␣a␣string
#
def␣set_board(i,j,t):
␣␣␣␣’’’
set␣turn␣t␣on␣the␣i,␣j␣place␣of␣the␣board,␣and␣return␣the␣status
returned␣value␣will␣be
␣␣’ok’␣if␣successfully␣places
␣␣’Not␣empty’␣the␣place␣is␣not␣empty
```

```
  'illegal turn' if turn value is not adequate
  'illegal slot' if place is not adequate
'''
    if (i>=0) and (i<3) and (j>=0) and (j<3):
        if (t>0) and (t<3):
            if examine_board(i, j) == 0:
                board[i][j] = t
                return 'OK'
            else:
                return 'Not empty'
        else:
            return 'illegal turn'
    else:
        return 'illegal slot'
```

```
#
# Testing function for the board
#
def test_board1():
    'The first test program on the board'
    init_board()
    print(show_board())
    print(set_board(0,0,1))
    print(show_board())
    print(set_board(1,1,2))
    print(show_board())
    print(set_board(1,1,1))
    print(show_board())
```

Program 13-4 Tic-tac-toe program, example (Part 4: Board-related functions part 2)

```
#
# Determine if a turn t wins in the horizontal direction
#
def check_board_horizontal(t):
    'Check whether turn t is win in horizontal direction'
    for i in range (3):
        if (board[i][0] == t) and (board[i][1] == t) and (board[i][2] == t):
            return True
    return False
#
# Determine if a turn t wins in the vertical direction
#
def check_board_vertical(t):
    'Check whether turn t is win in vertical direction '
    for j in range (3):
        if (board[0][j] == t) and (board[1][j] == t) and (board[2][j] == t):
            return True
    return False
#
# Determine if a turn t wins in the diagonal direction
#
def check_board_diagonal(t):
    'Check whether turn t is win in diagonal direction'
    if (board[0][0] == t) and (board[1][1] == t) and (board[2][2] == t):
        return True
    return False
#
# Determine if a turn t wins in the reverse diagonal direction
#
def check_board_inverse_diagonal(t):
    ' Check whether turn t is win in inverse diagonal direction '
    if (board[0][2] == t) and (board[1][1] == t) and (board[2][0] == t):
        return True
    return False
#
# Simple determination of victory for a turn t
#
def is_win_simple(t):
    'Check win of turn. Do not check win of the other turn'
    if check_board_horizontal(t):
        return True
    if check_board_vertical(t):
        return True
    if check_board_diagonal(t):
        return True
    if check_board_inverse_diagonal(t):
```

```
␣␣␣␣␣␣␣␣return␣True
␣␣␣␣return␣False
#
#␣Determine␣the␣winner␣by␣confirming␣the␣opponent␣has␣not␣won.
#
def␣is_win_actual(t):
␣␣␣␣'Check␣win␣of␣turn␣t.␣It␣also␣check␣whether␣the␣other␣turn␣do␣not␣win'
␣␣␣␣if␣not␣is_win_simple(t):
␣␣␣␣␣␣␣␣return␣False
␣␣␣␣if␣t==FIRST:
␣␣␣␣␣␣␣␣␣if␣is_win_simple(SECOND):
␣␣␣␣␣␣␣␣␣␣␣␣␣return␣False
␣␣␣␣else:
␣␣␣␣␣␣␣␣if␣is_win_simple(FIRST):
␣␣␣␣␣␣␣␣␣␣␣␣return␣False
␣␣␣␣return␣True
#
#␣Determine␣whether␣the␣board␣is␣full
#
def␣is_full():
␣␣␣␣'Confirm␣all␣the␣places␣are␣not␣empty'
␣␣␣␣for␣i␣in␣range(3):
␣␣␣␣␣␣␣␣for␣j␣in␣range(3):
␣␣␣␣␣␣␣␣␣␣␣␣if␣board[i][j]␣==␣OPEN:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣False
␣␣␣␣return␣True
#
#␣Determine␣whether␣a␣draw␣has␣occurred
#
def␣is_draw():
␣␣␣␣'Check␣wheter␣board␣is␣draw'
␣␣␣␣if␣is_win_simple(FIRST):
␣␣␣␣␣␣␣␣return␣False
␣␣␣␣if␣is_win_simple(SECOND):
␣␣␣␣␣␣␣␣return␣False
␣␣␣␣if␣not␣is_full():
␣␣␣␣␣␣␣␣return␣False
␣␣␣␣return␣True
```

Program 13-5 Tic-tac-toe program, example (Part 5: Board testing functions 1)

```
#
#␣Second␣board␣testing␣function,␣which␣tests␣the␣victory␣determination
#
def␣test_board2():
␣␣␣␣'The␣second␣test␣program␣of␣the␣board'
␣␣␣␣init_board()
␣␣␣␣board[0][0]␣=␣FIRST
␣␣␣␣board[1][0]␣=␣FIRST
␣␣␣␣board[2][0]␣=␣FIRST
␣␣␣␣print(show_board())
␣␣␣␣print("HORIZONTSL␣FIRST:␣"␣,check_board_horizontal(FIRST))
␣␣␣␣print("HORIZONTSL␣SECOND:␣",check_board_horizontal(SECOND))
␣␣␣␣print("VERTICAL␣FIRST:␣"␣␣␣,check_board_vertical(FIRST))
␣␣␣␣print("VERTICAL␣SECOND:␣"␣␣,check_board_vertical(SECOND))
␣␣␣␣init_board()
␣␣␣␣board[0][0]␣=␣SECOND
␣␣␣␣board[1][0]␣=␣SECOND
␣␣␣␣board[2][0]␣=␣SECOND
␣␣␣␣print(show_board())
␣␣␣␣print("HORIZONTSL␣FIRST:␣"␣,check_board_horizontal(FIRST))
␣␣␣␣print("HORIZONTSL␣SECOND:␣",check_board_horizontal(SECOND))
␣␣␣␣print("VERTICAL␣FIRST:␣"␣␣␣,check_board_vertical(FIRST))
␣␣␣␣print("VERTICAL␣SECOND:␣"␣␣,check_board_vertical(SECOND))

␣␣␣␣init_board()
␣␣␣␣board[0][0]␣=␣FIRST
␣␣␣␣board[0][1]␣=␣FIRST
␣␣␣␣board[0][2]␣=␣FIRST
␣␣␣␣print(show_board())
␣␣␣␣print("HORIZONTSL␣FIRST:␣"␣,check_board_horizontal(FIRST))
␣␣␣␣print("HORIZONTSL␣SECOND:␣",check_board_horizontal(SECOND))
␣␣␣␣print("VERTICAL␣FIRST:␣"␣␣␣,check_board_vertical(FIRST))
␣␣␣␣print("VERTICAL␣SECOND:␣"␣␣,check_board_vertical(SECOND))
␣␣␣␣init_board()
␣␣␣␣board[0][0]␣=␣SECOND
␣␣␣␣board[0][1]␣=␣SECOND
␣␣␣␣board[0][2]␣=␣SECOND
␣␣␣␣print(show_board())
␣␣␣␣print("HORIZONTSL␣FIRST:␣"␣,check_board_horizontal(FIRST))
␣␣␣␣print("HORIZONTSL␣SECOND:␣",check_board_horizontal(SECOND))
␣␣␣␣print("VERTICAL␣FIRST:␣"␣␣␣,check_board_vertical(FIRST))
␣␣␣␣print("VERTICAL␣SECOND:␣"␣␣,check_board_vertical(SECOND))

␣␣␣␣init_board()
␣␣␣␣board[0][0]␣=␣FIRST
␣␣␣␣board[1][1]␣=␣FIRST
```

```
␣␣␣␣board[2][2]␣=␣FIRST
␣␣␣␣print(show_board())
␣␣␣␣print("DIAGONAL␣FIRST:␣"␣,check_board_diagonal(FIRST))
␣␣␣␣print("DIAGONAL␣SECOND:␣",check_board_diagonal(SECOND))
␣␣␣␣print("INV␣DIAGONAL␣FIRST:␣"␣␣␣,check_board_inverse_diagonal(FIRST))
␣␣␣␣print("INV␣DIAGONAL␣SECOND:␣"␣␣,check_board_inverse_diagonal(SECOND))
␣␣␣␣init_board()
␣␣␣␣board[0][0]␣=␣SECOND
␣␣␣␣board[1][1]␣=␣SECOND
␣␣␣␣board[2][2]␣=␣SECOND
␣␣␣␣print(show_board())
␣␣␣␣print("DIAGONAL␣FIRST:␣"␣,check_board_diagonal(FIRST))
␣␣␣␣print("DIAGONAL␣SECOND:␣",check_board_diagonal(SECOND))
␣␣␣␣print("INV␣DIAGONAL␣FIRST:␣"␣␣␣,check_board_inverse_diagonal(FIRST))
␣␣␣␣print("INV␣DIAGONAL␣SECOND:␣"␣␣,check_board_inverse_diagonal(SECOND))

␣␣␣␣init_board()
␣␣␣␣board[0][2]␣=␣FIRST
␣␣␣␣board[1][1]␣=␣FIRST
␣␣␣␣board[2][0]␣=␣FIRST
␣␣␣␣print(show_board())
␣␣␣␣print("DIAGONAL␣FIRST:␣"␣,check_board_diagonal(FIRST))
␣␣␣␣print("DIAGONAL␣SECOND:␣",check_board_diagonal(SECOND))
␣␣␣␣print("INV␣DIAGONAL␣FIRST:␣"␣␣␣,check_board_inverse_diagonal(FIRST))
␣␣␣␣print("INV␣DIAGONAL␣SECOND:␣"␣␣,check_board_inverse_diagonal(SECOND))
␣␣␣␣init_board()
␣␣␣␣board[0][2]␣=␣SECOND
␣␣␣␣board[1][1]␣=␣SECOND
␣␣␣␣board[2][0]␣=␣SECOND
␣␣␣␣print(show_board())
␣␣␣␣print("DIAGONAL␣FIRST:␣"␣,check_board_diagonal(FIRST))
␣␣␣␣print("DIAGONAL␣SECOND:␣",check_board_diagonal(SECOND))
␣␣␣␣print("INV␣DIAGONAL␣FIRST:␣"␣␣␣,check_board_inverse_diagonal(FIRST))
␣␣␣␣print("INV␣DIAGONAL␣SECOND:␣"␣␣,check_board_inverse_diagonal(SECOND))
```

Program 13-6 Tic-tac-toe program, example (Part 6: Board testing functions 2)

```
#
#␣Third␣board␣testing␣function,␣which␣determines␣whether␣it␣is␣a␣victory␣or␣draw
#
def␣test_board3():
␣␣␣␣’␣The␣third␣test␣program␣of␣the␣board␣’
␣␣␣␣init_board()
␣␣␣␣board[0][0]␣=␣FIRST
␣␣␣␣board[1][0]␣=␣FIRST
␣␣␣␣board[2][0]␣=␣SECOND
␣␣␣␣board[0][1]␣=␣SECOND
␣␣␣␣board[1][1]␣=␣SECOND
␣␣␣␣board[2][1]␣=␣FIRST
␣␣␣␣board[0][2]␣=␣FIRST
␣␣␣␣board[1][2]␣=␣FIRST
␣␣␣␣board[2][2]␣=␣SECOND
␣␣␣␣print(show_board())
␣␣␣␣print(”HORIZONTSL␣FIRST:␣”␣,check_board_horizontal(FIRST))
␣␣␣␣print(”HORIZONTSL␣SECOND:␣”,check_board_horizontal(SECOND))
␣␣␣␣print(”VERTICAL␣FIRST:␣”␣␣␣,check_board_vertical(FIRST))
␣␣␣␣print(”VERTICAL␣SECOND:␣”␣␣,check_board_vertical(SECOND))
␣␣␣␣print(”DIAGONAL␣FIRST:␣”␣,check_board_diagonal(FIRST))
␣␣␣␣print(”DIAGONAL␣SECOND:␣”,check_board_diagonal(SECOND))
␣␣␣␣print(”INV␣DIAGONAL␣FIRST:␣”␣␣␣,check_board_inverse_diagonal(FIRST))
␣␣␣␣print(”INV␣DIAGONAL␣SECOND:␣”␣␣,check_board_inverse_diagonal(SECOND))
␣␣␣␣print(”IS␣WIN␣SIMPLE␣FIRST”,␣is_win_simple(FIRST))
␣␣␣␣print(”IS␣WIN␣SIMPLE␣SECOND”,␣is_win_simple(SECOND))
␣␣␣␣print(”IS␣WIN␣ACTUAL␣FIRST”,␣is_win_actual(FIRST))
␣␣␣␣print(”IS␣WIN␣ACTUAL␣SECOND”,␣is_win_actual(SECOND))
␣␣␣␣print(”IS␣FULL”,␣is_full())
␣␣␣␣print(”IS␣DRAW”,␣is_draw())

␣␣␣␣init_board()
␣␣␣␣board[0][0]␣=␣FIRST
␣␣␣␣board[1][0]␣=␣SECOND
␣␣␣␣board[2][0]␣=␣FIRST
␣␣␣␣board[0][1]␣=␣SECOND
␣␣␣␣board[1][1]␣=␣FIRST
␣␣␣␣board[2][1]␣=␣OPEN
␣␣␣␣board[0][2]␣=␣FIRST
␣␣␣␣board[1][2]␣=␣OPEN
␣␣␣␣board[2][2]␣=␣SECOND
␣␣␣␣print(show_board())
␣␣␣␣print(”HORIZONTSL␣FIRST:␣”␣,check_board_horizontal(FIRST))
␣␣␣␣print(”HORIZONTSL␣SECOND:␣”,check_board_horizontal(SECOND))
␣␣␣␣print(”VERTICAL␣FIRST:␣”␣␣␣,check_board_vertical(FIRST))
␣␣␣␣print(”VERTICAL␣SECOND:␣”␣␣,check_board_vertical(SECOND))
```

```
    print("DIAGONAL FIRST: " ,check_board_diagonal(FIRST))
    print("DIAGONAL SECOND: ",check_board_diagonal(SECOND))
    print("INV DIAGONAL FIRST: "   ,check_board_inverse_diagonal(FIRST))
    print("INV DIAGONAL SECOND: "  ,check_board_inverse_diagonal(SECOND))
    print("IS WIN SIMPLE FIRST", is_win_simple(FIRST))
    print("IS WIN SIMPLE SECOND", is_win_simple(SECOND))
    print("IS WIN ACTUAL FIRST", is_win_actual(FIRST))
    print("IS WIN ACTUAL SECOND", is_win_actual(SECOND))
    print("IS FULL", is_full())
    print("IS DRAW", is_draw())

    init_board()
    board[0][0] = SECOND
    board[1][0] = FIRST
    board[2][0] = SECOND
    board[0][1] = FIRST
    board[1][1] = SECOND
    board[2][1] = FIRST
    board[0][2] = SECOND
    board[1][2] = OPEN
    board[2][2] = FIRST
    print(show_board())
    print("HORIZONTSL FIRST: " ,check_board_horizontal(FIRST))
    print("HORIZONTSL SECOND: ",check_board_horizontal(SECOND))
    print("VERTICAL FIRST: "   ,check_board_vertical(FIRST))
    print("VERTICAL SECOND: "  ,check_board_vertical(SECOND))
    print("DIAGONAL FIRST: " ,check_board_diagonal(FIRST))
    print("DIAGONAL SECOND: ",check_board_diagonal(SECOND))
    print("INV DIAGONAL FIRST: "   ,check_board_inverse_diagonal(FIRST))
    print("INV DIAGONAL SECOND: "  ,check_board_inverse_diagonal(SECOND))
    print("IS WIN SIMPLE FIRST", is_win_simple(FIRST))
    print("IS WIN SIMPLE SECOND", is_win_simple(SECOND))
    print("IS WIN ACTUAL FIRST", is_win_actual(FIRST))
    print("IS WIN ACTUAL SECOND", is_win_actual(SECOND))
    print("IS FULL", is_full())
    print("IS DRAW", is_draw())
```

Program 13-7 Tic-tac-toe program example (Part 7: Board record-related functions)

```
#
#␣Log␣replay
#
def␣replay_log(log):
␣␣␣␣’Replay␣a␣game␣log.␣It␣shows␣replay␣on␣screen␣with␣print()␣function’
␣␣␣␣init_board()
␣␣␣␣init_turn()
␣␣␣␣print(show_board())
␣␣␣␣for␣m␣in␣log:
␣␣␣␣␣␣␣␣if␣len(m)␣==␣2:
␣␣␣␣␣␣␣␣␣␣␣␣print(show_turn(),”␣is␣the␣current␣turn”)
␣␣␣␣␣␣␣␣␣␣␣␣print(set_board(m[0],␣m[1],␣turn))
␣␣␣␣␣␣␣␣␣␣␣␣print(show_board())
␣␣␣␣␣␣␣␣␣␣␣␣print(”IS␣WIN”,␣turn,␣”:␣”,␣is_win_actual(turn))
␣␣␣␣␣␣␣␣␣␣␣␣change_turn()
␣␣␣␣␣␣␣␣else:
␣␣␣␣␣␣␣␣␣␣␣␣print(”RESULT␣IN␣LOG:␣”,m[0])
␣␣␣␣print(”IS␣WIN␣FIRST:␣”,␣is_win_actual(FIRST))
␣␣␣␣print(”IS␣WIN␣SECOND:␣”,␣is_win_actual(SECOND))
␣␣␣␣print(”IS␣DRAW:␣”,␣is_draw())
#
#␣Log␣test
#
def␣test_log():
␣␣␣␣’Test replay of logs’
␣␣␣␣print(”LOG1”)
␣␣␣␣replay_log(log1)
␣␣␣␣print(”LOG2”)
␣␣␣␣replay_log(log2)
␣␣␣␣print(”LOG3”)
␣␣␣␣replay_log(log3)
#
#␣Test␣all
#
def␣test_all():
␣␣␣␣’Do all the test programs’
␣␣␣␣test_turn()
␣␣␣␣test_board1()
␣␣␣␣test_board2()
␣␣␣␣test_board3()
␣␣␣␣test_log()
```

Program 13-8 Tic-tac-toe program, example (Part 8: The play() function and the main program)

```
#
# Actual gameplay
#
def play():
    'Conduct an actual tic-tac-toe game interactively on terminal'
    init_turn()
    init_board()
    print(show_board())
# Create an empty list for the game board record. 
# Declare it as a global variable if you want to access it outside of play()
#    global log
    log = []
    while True:
        print(show_turn()," is the current turn")
        while(True):
            row = int(input("Input the number of row you want to place: "))
            column = int(input("Input a number of column: "))
            result = set_board(row, column, turn)
            print(result)
            if result == "OK":
                break
            print("Inadequate value(s), try again")
        # Add a turn to the log here (outside of the inner while)
        # Additional code required here
        #
        print(show_board())
        if is_draw():
            print("Game is draw")
# Add the game result (a draw) to log here.
            break
        if is_win_actual(turn):
            print(show_turn(), "won the game")
# Add the game result (current player's win) to log here.
            break
        change_turn()
    # This is a replay of the game
    # Currently the log is empty, so determine victory, and process
    if len(log)>0:
        replay_log(log)
    else:
        print("Game log was not recorded")
if __name__ == '__main__':
    print('Tic-Tac-Toe')
```

3) Testing the program

After reading the above program and running it in the Python shell, run the test functions from the shell to see how the program works. The test functions provided are as follows, in order from top to bottom.

```
test_turn()
test_board1()
test_board2()
test_board3()
test_log()
test_all()
```

4) Running the program

After running the above program, call the play() function from the Python shell and try playing tic-tac-toe.

```
play()
```

Exercise 13-2 Getting the board record

Extend the above program to collect the board record of the game in the play() function. You can also replay the board record after the winner is determined.

13.6 Test of skills

Using the methods you have learned so far, try the following as a test of your skills.

- Create a GUI-based tic-tac-toe game instead of a CUI based one, using tkinter.
- By using classes, implement functions to control the turn, the board, the game record, etc. as methods in the classes.
- Extend the game to be played by a computer on one side of the board, rather than by two humans.
- Develop a similar program for the game of Reversi instead of tic-tac-toe.

13.7 Various topics related to program development

13.7.1 What you can understand from a test

Computer programs usually have to work properly in a virtually infinite number of cases. For example, the number of possible games of tic-tac-toe is quite large. For this reason, you must consider the following points concerning tests:

- It is easy to see that **a program that does not pass the test has a mistake in it**; however,

- you must remember that **if a program passes a test, it does not guarantee that it will run properly in all cases where it is expected to have passed the test.**

It is easier to test component functions than it is to test their combined functions. You can build more confidence in your skill at using complex functions by building them with correctly working components.

13.7.2 Refactoring

There are two ways in which a program can be improved, as below

- Enhancing the functionality of the program
- Maintaining the functionality of the program, but rearranging how it is implemented in order to make it easier to maintain and extend.

The latter kind of improvement is called "**refactoring**." An example of this is reprogramming your tic-tac-toe program using classes. There are several reasons why refactoring is necessary.

- Programs are used for a long time, so they must be easy to maintain.
- Also, the developers of the program may be replaced.
- There is a constant demand for additional functionality in programs.

13.7.3 The V-model of software development

In the first half of the development of a program, even a small example like the tic-tac-toe program you made above, the requirements of the final product (specifications) are clear. Then, you flesh out the specifics of the program step by step, including the functions to be implemented. In the second half of development, you repeatedly perform tests on non-interdependent functions in sequence, which eventually leads to the conclusion of programming after many iterations.

This process is represented by a V-shape as shown in the figure below, and is called the V-model of software development.

Because of the V-shape structure, the distance from design to implementation and testing is much greater in the upper part of the V-shape. This means that inappropriate design in the upper part of the V takes longer to be discovered in implementation and testing, and requires more diligent reworking.

There are two possible directions for successful software development detailed below. The latter is called agile development.

- Define required specifications, and reduce reworking
- Narrow down the specifications and develop quickly, then add features if necessary.

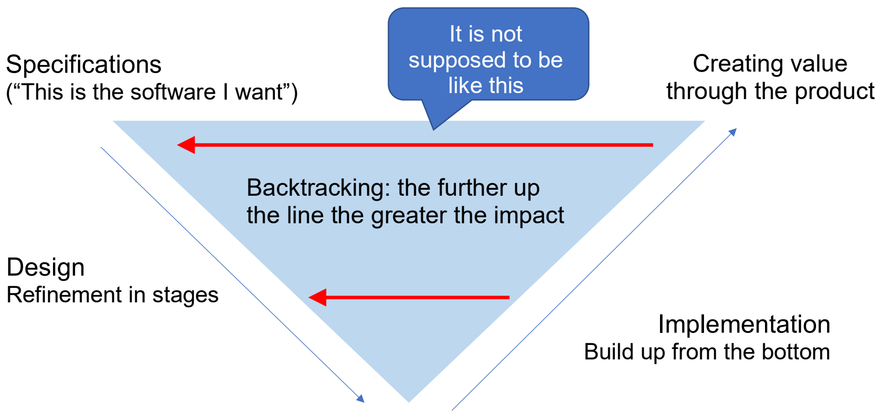

Specifications
("This is the software I want")
It is not supposed to be like this
Creating value through the product
Backtracking: the further up the line the greater the impact
Design
Refinement in stages
Implementation
Build up from the bottom

Figure 13-5 The V-Model of Software Development

Next Chapter Table of Contents

14. Academic Use of Python

14.1 Learning goals of this chapter

One of the reasons why Python has attracted so much attention is that it offers a wide range of libraries suitable for numerical computation, as well as for other academic applications. We will use the following three libraries in this chapter to learn the basics of working with data (in NumPy and pandas), as well as the basics of graphing (in Matplotlib).

Each of them is a quite sophisticated package, and you need to have some expertise in the fields in which these packages are applied in order to fully utilize them, so you will learn only the basics here.

1. NumPy: A basic package for numerical computation in the field of science and technology
   SciPy: A library for more advanced numerical computations
2. Matplotlib: A package for plotting data on graphs
3. pandas: a data analysis package

These are interrelated, as shown in the figure below.

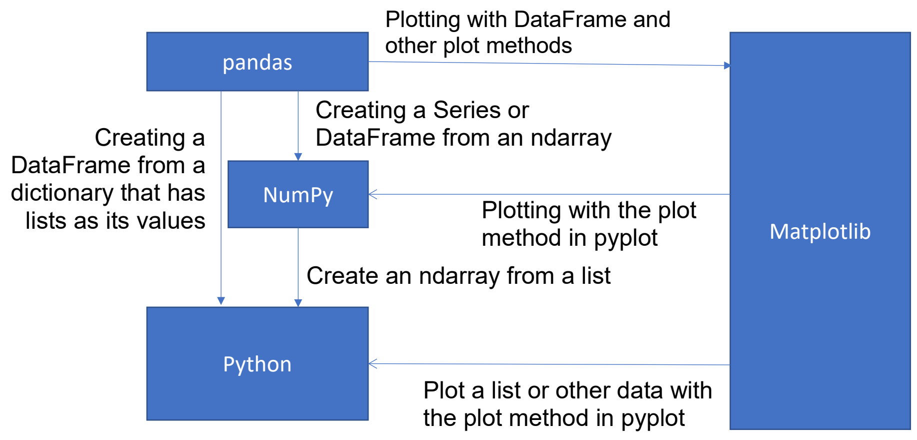

pandas
Plotting with DataFrame and other plot methods
Creating a Series or DataFrame from an ndarray
Creating a DataFrame from a dictionary that has lists as its values
NumPy
Plotting with the plot method in pyplot
Matplotlib
Create an ndarray from a list
Python
Plot a list or other data with the plot method in pyplot

Figure 14-1 The relationship between NumPy, Matplotlib and pandas

14.2 Using a custom name when importing

The following custom names are often used for importing packages such as NumPy, Pandas, Matplotlib. You will use these custom names in this text.

```
import numpy as np

import matplotlib.pyplot as plt

import pandas as pd
```

14.3 NumPy

Python is a slow programming language, but NumPy is written in C, and can perform vector and matrix operations very quickly.

14.3.1 Generating multi-dimensional arrays

While Python uses lists to group data together, NumPy uses its own data format, the ndarray (aka array).

1) Generating from a list

```
import numpy as np
data1 = [1, 2, 3]
arr1 = np.array(data1) # 1-dimensional data
data2 = [[1,2,3],[4,5,6]]
arr2 = np.array(data2)
```

2) Create an array in which all elements are zero

```
    np.zeros(5)   # A 1D array of with 5 elements
    array([0., 0., 0., 0., 0.])
    np.zeros((2,2))    # A 2D array of with a size of (2,2); note the double
parentheses ()
    array([[0., 0.],
    [0., 0.]])
    a = np.array([[1,2,3],[4,5,6]])
    np.zeros_like(a)    # Same size as array a
    array([[0, 0, 0],
    [0, 0, 0]])
```

An array consisting only of 1's can be created using ones and ones_like.

14.3.2 ndarray attributes

You can read up on your own about the following attributes of ndarrays.

- ndim：ndarray dimension
- shape: ndarray size
- dtype: Data type

Next Chapter Table of Contents

```
import numpy as np
arr2 = py.array([[1,2,3],[4,5,6]]
arr2.ndim
2
arr2.shape
(2,3)
arr2.dtype
dtype('Int32')
```

Note that while Python's integer types have no limit on the number of digits, ndarray uses a fixed-length type that allows fast operations to be performed in order to speed up computation.

14.3.3 Accessing ndarray elements

The elements of an ndarray can be accessed using [] in the same way as a list. These ndarrays also start at an index of 0.

```
arr1 = np.array([1,2,3])
arr1[0]
1
arr1[1] = 1
arr1
array([1,1,3])
```

For multi-dimensional arrays you can also use [,] notation instead of [][].

```
arr2[0][0]
arr2[0,0]
```

14.3.4 Slicing

ndarrays can be sliced in the same manner as lists can.

```
arr1[2:]
array([3])
```

Multi-dimensional arrays use the following notation: [:,:]

```
arr2[0:2,0:2]
array([1,2][4,5])
```

Note The result of slicing an ndarray is not a “copy,” but a reference to a part of the original ndarray.

If you assign a scalar value to a slice, it will be assigned to all elements.

14.3.5 ndarray operations

ndarray data can be used to perform arithmetic operations, powers, comparisons, and so on. These operations are iterable over all elements in the ndarray.

Matrix products use the @ operator.

The value is applied to all the elements for scalar operations.

```
arr1 = np.array([1,2,3])
arr1*2
array([2,4,6])
arr1 + 1
array([2,3,4])
```

14.3.6 Extraction of elements satisfying the conditions

You can extract the elements that satisfy the conditions in the following way.

```
    arr1 = np.array([1,2,3,4,5])
    cond = arr1 > 2     # Generate an array of elements that satisfy the
conditions
    cond
    array([False, False,  True,  True,  True])
    arr1[cond]          # Slice based on the defined conditions
    array([3, 4, 5])
```

14.3.7 Matrix Calculations

Numpy makes it easy to perform mathematical matrix calculations on ndarrays.

- Matrix transposition (swapping rows and columns)
  We will use the T attribute of ndarray.
- Matrix product
  We will use the @ operator.
- Using the linalg (linear algebra) module: numpy.linalg (or np.linalg if you imported numpy with the np alias) defines the following matrix functions.
  - diag (diagonal elements),
  - trace (sum of diagonal elements),
  - det (determinant),

- eig (eigenvalue),
- inv (inverse),
- solve (to solve a linear equation)

14.3.8 Random numbers

Numpy allows you to generate random numbers in bulk.

- seed: sets the initial value for random number generation.
- rand: generates uniformly continuous random numbers.
- randn: generates random numbers that follow a standard normal distribution.
- randint: generates random numbers in a given range.

Here is an example of how to use it.

- np.random.rand(10)
  Generates 10 floating-point random numbers with values between 0 and 1
- np.random.randn(5,5)
  Generates random numbers that follow a standard normal distribution as a two-dimensional array of size (5,5)
- np.random.permutation([1,2,3,4,5])
  Generates a random permutation of the list [1,2,3,4,5]. range() and ndarray can be specified as arguments. For multi-dimensional arrays, only the first index is replaced.
- np.random.randint(2,size=10)
  Generates an integer random number in range(0,2) with a size of 10. Since we are able to provide lower and upper limits, The size of the array can be specified using “size=”

14.4 Matplotlib

14.4.1 backend: graph output environment

Matplotlib allows you to choose how to output the graph. In this section, we see how to use tkagg, which is a tkinter-based environment for IDLE.

- In Matplotlib, the environment that actually outputs the graph is called a backend. There are various backends available.
- tkagg is a backend that uses Tkinter to output graphs.
- In the IDLE environment, it is specified using the use() function after importing matplotlib.

```
import matplotlib
matplotlib.use('tkagg')
```

- The use() function must be specified before importing the matplotlib.pyplot plotting module.

- In IPython and Jupyter Notebook (which uses IPython), before using matplotlib, you can specify the following, depending on the backend you want to use.

```
%matplotlib notebook
%matplotlib tk
```

14.4.2 Text output in Japanese

- The standard matplotlib font does not support Japanese characters, so they will be displayed as boxes (□).
- The ttc font can now be used in matplotlib version 3.1 or later. If you are using this version or a newer one, you do not need to install additional fonts [1].
- There are several ways to specify fonts, but here we show how to specify them all at once in the program.
- If you have installed additional fonts, first delete the fontList.json in the .matplotlib folder in the user's folder if it is old.
- Overview of how to use it:

```
import matplotlib
# Set tkinter as the output destination before importing pyplot
matplotlib.use('tkagg')
import matplotlib.pyplot as plt
# Enable matplotlib to display Japanese characters.
# Yu Gothic can be used in matplotlib version 3.1 or later
matplotlib.rc('font', **{'family':'Yu Gothic'})
# For Mac User, try the following instead of the above line
# matplotlib.rc('font', **{'family':'Hiragino Maru Gothic Pro'})
# The following is an example plot
data = [1,2,3]
plt.plot(data)
plt.title('Title')
plt.show()
```

14.4.3 Setting the Title, Axis Labels, and Linestyle

- The function to set the title of the graph is title.

[1] The following example uses Yu Gothic, but Yu Mincho, MS Gothic, MS Mincho, etc. can also be used.

- The function to set the x-axis labels is xlabel.
- The function to set the y-axis labels is ylabel.
- The linestyle can be specified as an argument of the plot function.
  - By specifying the color, and linestyle, and marker as strings
  - By specifying the color, linestyle, linewidth, etc. as arguments.
- Example (the relevant part only)

```
plt.plot([1,2,3], 'k-')   # A black solid line
plt.plot([2,3,4], 'r--')  # A red dashed line
plt.plot([3,4,5], 'b--o') # A blue dashed line with circular markers (○)
plt.title('Title')
plt.xlabel('Horizontal Axis ')
plt.ylabel('Vertical Axis ')
plt.show()
```

14.4.4 Example

1) use_matplotlib_en.py

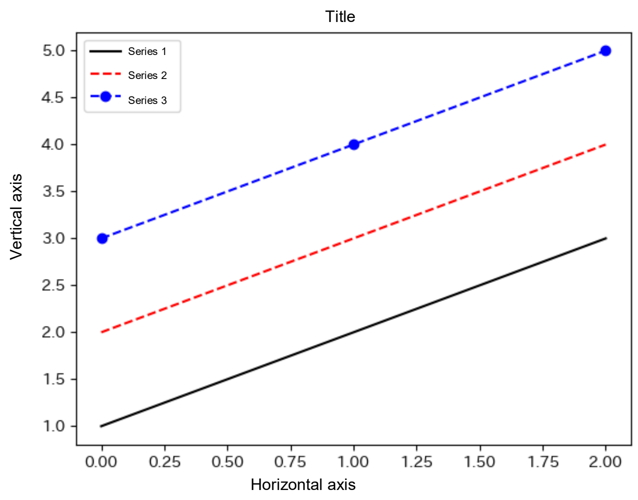


Figure 14-2 Example of Matplotlib usage

Program 14-1 use_matplotlib_en.py

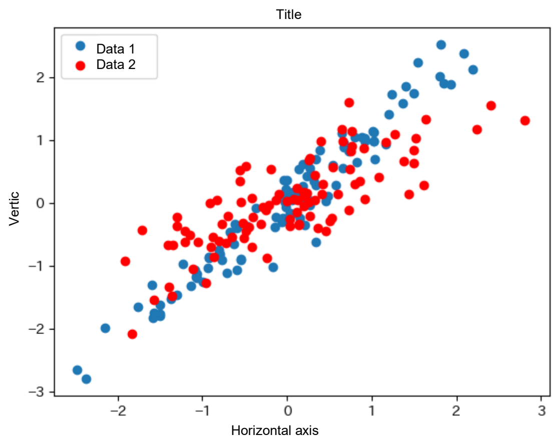
```
#
#␣The␣basic␣way␣to␣use␣matplotlib
#
import␣matplotlib
#
#␣set␣tkinter␣to␣use␣(as␣the␣output␣destination)␣before␣importing␣pyplot
#
matplotlib.use('tkagg')
import␣matplotlib.pyplot␣as␣plt
#
#␣Enable␣matplotlib␣to␣display␣Japanese␣characters.
#␣uncomment␣one␣apropriate␣for␣your␣environment
#
#␣For␣Windows
matplotlib.rc('font',␣**{'family':'Yu␣Gothic'})
#␣For␣Campus␣PC␣Terminal
#matplotlib.rc('font',␣**{'family':'IPAPGothic'})
#␣For␣macOS
#matplotlib.rc('font',␣**{'family':'Hiragino␣Maru␣Gothic␣Pro'})
#
#␣Draw␣a␣line␣graph␣with␣three␣lines
#
plt.plot([1,2,3],'k-',label='Series␣1')
plt.plot([2,3,4],'r--',label='Series␣2')
plt.plot([3,4,5],'b--o',label='Series␣3')
#
plt.title('Title')
plt.xlabel('Horizontal␣Axis␣')
plt.ylabel('Vertical␣Axis␣')
plt.legend()␣#␣Legend
plt.show()
```

2) Drawing a scatter plot

A scatter plot can be drawn by giving the x-axis data and y-axis data to the scatter function of pyplot.

- use_matplotlib_scatter_en.py

Next Chapter Table of Contents



•

Figure 14-3 Drawing a scatter plot

Program 14-2 use_matplotlib_scatter_en.py

```
#
#␣Draw␣a␣scatter␣plot␣with␣matplotlib
#
import␣matplotlib

matplotlib.use('tkagg')
import␣matplotlib.pyplot␣as␣plt
import␣numpy␣as␣np
#
#␣Enable␣matplotlib␣to␣display␣Japanese␣characters.
#␣uncomment␣one␣apropriate␣for␣your␣environment
#
#␣For␣Windows
matplotlib.rc('font',␣**{'family':'Yu␣Gothic'})
#␣For␣Campus␣PC␣Terminal
#matplotlib.rc('font',␣**{'family':'IPAPGothic'})
#␣For␣macOS
#matplotlib.rc('font',␣**{'family':'Hiragino␣Maru␣Gothic␣Pro'})
#
#␣Create␣random␣data
#
datax␣=␣np.random.randn(100)
datay␣=␣datax␣+␣np.random.randn(100)*0.3
#
#␣Draw␣a␣scatter␣plot.
#
plt.scatter(datax,datay,label='Data1')
#
#␣Create␣another␣set␣of␣data
```

```
#␣
datax␣=␣np.random.randn(100)
datay␣=␣0.6*datax␣+␣np.random.randn(100)*0.4
#
#␣Specify␣the␣color,␣then␣create␣a␣scatter␣plot
#
plt.scatter(datax,datay,color='red',label='Data2')
#
#␣Fill␣in␣the␣title,␣axis␣labels,␣and␣legend
#
plt.title('Title')
plt.xlabel('Horizontal Axis ')
plt.ylabel('Vertical Axis ')
plt.legend()
#
#␣Display
#
plt.show()
```

•

3) Drawing a Histogram

You can draw a histogram by feeding the data for it into the hist function of pyplot. The number of bars is automatically adjusted, but it can also be specified.

use_matplotlib_hist_en.py

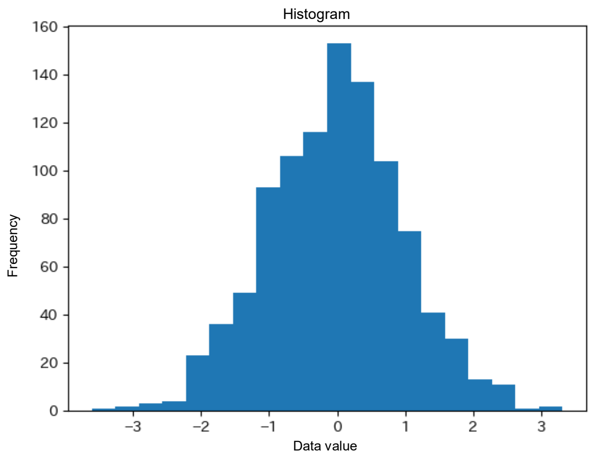


Figure 14-4 Drawing a Histogram

Program 14-3 use_matplotlib_hist_en.py

```
#
#␣Draw␣a␣histogram␣with␣matplotlib
#
```

```
import␣matplotlib
#
#␣Set␣tkinter␣as␣the␣output␣destination␣before␣importing␣pyplot
#
matplotlib.use('tkagg')
import␣matplotlib.pyplot␣as␣plt
import␣numpy␣as␣np
#
#␣Enable␣matplotlib␣to␣display␣Japanese␣characters.
#␣uncomment␣one␣apropriate␣for␣your␣environment
#
#␣For␣Windows
matplotlib.rc('font',␣**{'family':'Yu␣Gothic'})
#␣For␣Campus␣PC␣Terminal
#matplotlib.rc('font',␣**{'family':'IPAPGothic'})
#␣For␣macOS
#matplotlib.rc('font',␣**{'family':'Hiragino␣Maru␣Gothic␣Pro'})
#
#␣Create␣a␣histogram
#
data␣=␣np.random.randn(1000)
plt.hist(data,bins=20)
#
#␣Set␣the␣title␣and␣axis␣labels
#
plt.title('Histogram')
plt.xlabel('Value of Data')
plt.ylabel('Data Frequency')
#
#␣Display
#
plt.show()
```

4) Drawing Multiple Graphs

Matplotlib allows you to draw multiple graphs side by side in the following manner.

- Obtain a Figure object using pyplot's figure function.
- Add a subplot to the Figure object with pyplot's add_subplot function. Save the result to a variable.
- Adjust the spacing of the subplots with pyplot's subplots_adjust function.
- Draw each subplot with the plot, scatter, and hist functions.
- Add the title and axis labels with set_title, set_xlabel, and set_ylabel. Make sure the function names are correct.

Next Chapter Table of Contents

use_matplotlib_subplot_en.py

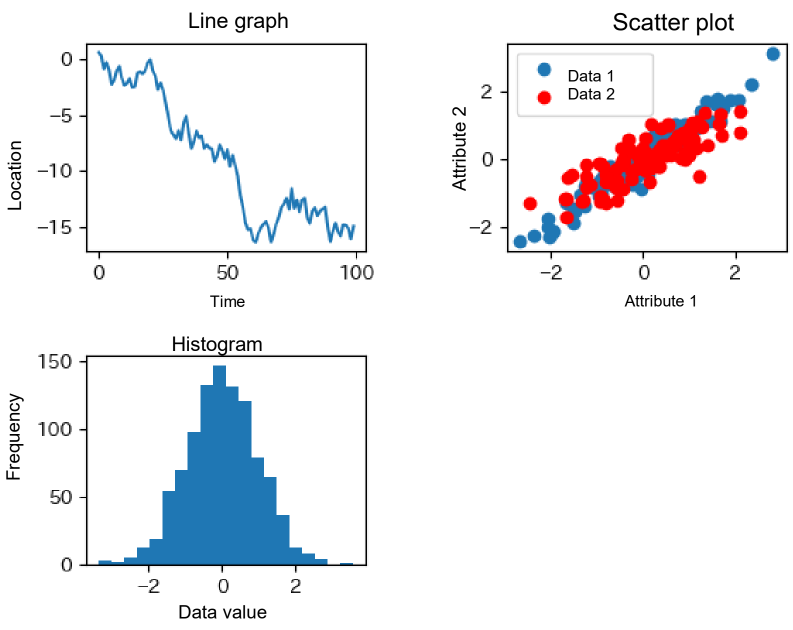


Figure 14-5 Drawing Multiple Graphs

Program 14-4 use_matplotlib_subplot_en.py

```
#
#␣Example␣using␣subplot
#
import␣matplotlib
matplotlib.use('tkagg')
import␣matplotlib.pyplot␣as␣plt
import␣numpy␣as␣np
#
#␣Enable␣matplotlib␣to␣display␣Japanese␣characters.
#␣uncomment␣one␣apropriate␣for␣your␣environment
#
#␣For␣Windows
matplotlib.rc('font',␣**{'family':'Yu␣Gothic'})
#␣For␣Campus␣PC␣Terminal
#matplotlib.rc('font',␣**{'family':'IPAPGothic'})
#␣For␣macOS
#matplotlib.rc('font',␣**{'family':'Hiragino␣Maru␣Gothic␣Pro'})
#
#␣Create␣3␣subplots,␣and␣adjust␣the␣spacing
#
fig␣=␣plt.figure()
ax1␣=␣fig.add_subplot(2,2,1)
ax2␣=␣fig.add_subplot(2,2,2)
```

```
ax3␣=␣fig.add_subplot(2,2,3)
plt.subplots_adjust(hspace=0.5,␣wspace=␣0.5)
#
#␣First,␣output␣a␣line␣graph
#␣
data␣=␣np.random.randn(100).cumsum()
ax1.plot(data)
ax1.set_title('Line Graph')
ax1.set_xlabel('Time')
ax1.set_ylabel('Place')
#
#␣Second,␣output␣a␣scatter␣plot
#
datax␣=␣np.random.randn(100)
datay␣=␣datax␣+␣np.random.randn(100)*0.3
ax2.scatter(datax,datay,label='Data1')

datax␣=␣np.random.randn(100)
datay␣=␣0.6*datax␣+␣np.random.randn(100)*0.4
ax2.scatter(datax,datay,color='red',label='Data2')

ax2.set_title('Scatter Plot')
ax2.set_xlabel('Attribute1')
ax2.set_ylabel('Attribute2')
ax2.legend()

#
#␣Third,␣output␣a␣histogram
#
data␣=␣np.random.randn(1000)
ax3.hist(data,bins=20)

ax3.set_title('Histogram')
ax3.set_xlabel('Value of Data')
ax3.set_ylabel('Data Frequency')

#
#␣Display␣the␣graphs
#␣
plt.show()
```

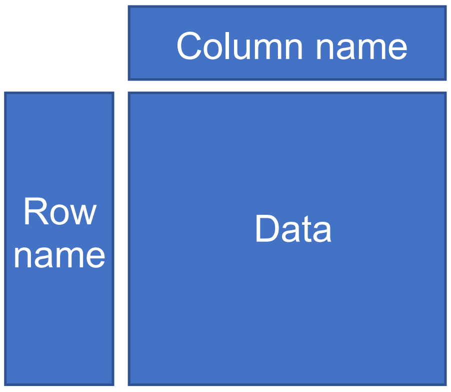
14.5 pandas

14.5.1 Dataframe

The following are Pandas-specific data formats.

- One-dimensional Series
- Two-dimensional DataFrame

A DataFrame has a row name (index) and a column name (column).



14.5.2 Create a DataFrame

1) Create from an array in numpy

```
import numpy as np
import pandas as pd
d = np.array([[1,2,3],[4,5,6],[7,8,9]])
df = pd.DataFrame(d,columns=['a' ,'b' ,'c'])
df
```

```
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

The column and row names can be found using:

```
df.columns
df.index
```

respectively.

Next Chapter Table of Contents

2) Create from a dictionary where the values are lists.

```
df = pd.DataFrame({'a': [1,4,7], 'b':[2,5,8], 'c':[3,6,9]})
df
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

A dictionary is a Python data type that consists of a set of key-value pairs.

```
dic = {'a':1, 'b':2, 'c':3}
```

You can search for values using the keys.

```
dic['a']
1
```

14.5.3 Importing csv files

- You can create a DataFrame by reading data from a spreadsheet saved as a csv file.

  ```
  df = pd.read_csv("file name")
  ```

- The first row is treated as the column name.
  - If you want to read all the data, specify the following options.

  ```
  header = None or names = [list of column names]
  ```

- To read a file containing Japanese characters on Windows, specify the Japanese character encoding.

  ```
  encoding = 'SHIFT-JIS'
  ```

- If you use Japanese character encoding for column names, an error will occur when specifying data in column names.
- sample.csv is show below.

ID	Japanese	English	Mathematics	Science	Social Studies
A	91	69	100	82	94
B	80	60	45	52	46
C	92	92	76	73	97
D	58	50	60	71	77
E	58	75	96	96	94
F	92	89	86	82	74
G	97	87	59	55	56

Follow the steps below to import it (use_read_csv_en.py)

1. Import needed modules

```
import numpy as np

import pandas as pd

import os
```

2. get folder path

```
folderpath␣=␣input(”Enter␣folder␣path:␣”)
```

3. Use os.chdir to change to the folder with the csv file.

```
os.chdir(folderpath)
```

4. Import the csv file

```
df = pd.read_csv(”sample.csv”)
```

Note: pd.read_csv does not seem to be able to handle Japanese file names correctly.

Program 14-5 use_read_csv_en.py

```
import␣numpy␣as␣np
import␣pandas␣as␣pd
import␣os
#
#␣Navigate␣to␣the␣folder␣with␣the␣data
#
#␣pandas␣does␣not␣handle␣Japanese␣file␣names␣well
#
folderpath␣=␣input(″Enter␣folder␣path:␣″)
os.chdir(folderpath)
df␣=␣pd.read_csv(″sample.csv″)
#
#␣Sum␣horizontally␣(axis␣=␣1)␣and␣create␣a␣column␣called␣Total
df['Total']␣=␣df.sum(axis=1)
#␣Display␣the␣DataFrame␣df
print(df)
#␣Display␣summary␣statistics␣for␣DataFrame␣df
print(df.describe())
```

14.5.4 Display summary statistics

The describe method can be used to display summary statistics.

14.5.5 Plotting Data in Pandas

Plotting in Pandas is done by calling the plot method in the DataFrame. (use_DadaFrame_plot.py)

```
df.plot()     # Line graph
df.plot.bar(stacked=True) # stacked bar graph
df.plot.scatter('Japanese','English') # Scatter plot with defined columns
df['Japanese'].plot.hist() # Histogram with defined columns
```

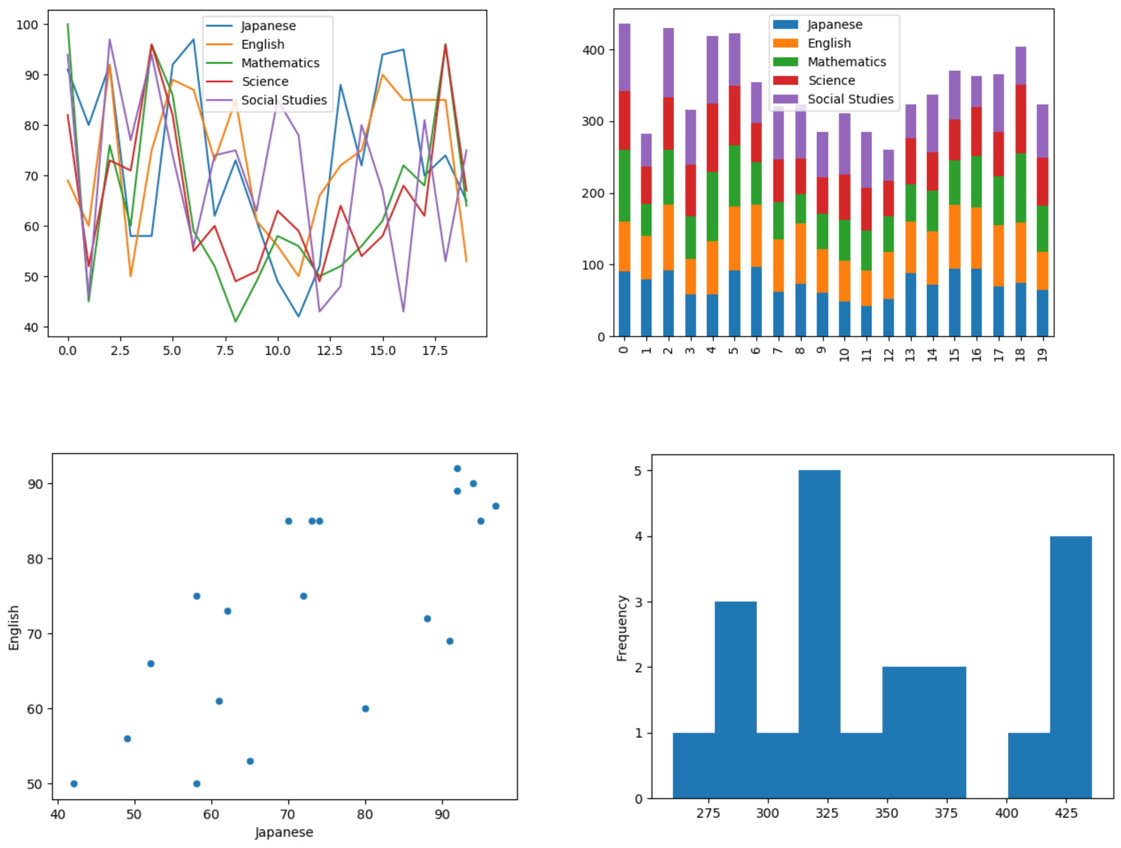


Figure 14-6 Graphing with pandas

Program 14-6 use_DadaFrame_plot_en.py

```
import␣numpy␣as␣np
import␣pandas␣as␣pd
import␣os
import␣matplotlib
matplotlib.use('tkagg')
import␣matplotlib.pyplot␣as␣plt
#
#␣Navigate␣to␣the␣folder␣with␣the␣data
#
#␣set␣adequate␣path␣for␣your␣environment
folderpath␣=␣input("Enter␣folder␣path:␣")
os.chdir(folderpath)
#
df␣=␣pd.read_csv("sample.csv")
#
#
#␣Line␣graph
```

Next Chapter Table of Contents

```
#

df.plot()
print("Close window to proceed")
plt.show()
#
# Stacked bar graph
#
df.plot.bar(stacked=True)
print("Close window to proceed ")
plt.show()
#
# Scatter plot
#
df.plot.scatter('Japanese','English')
print("Close window to proceed ")
plt.show()
#
# Sum horizontally (axis = 1) and create a column called Total
#
df['Total'] = df.sum(axis=1)

#
# Histogram
#
df['Total'].plot.hist()
print("Close window to proceed ")
plt.show()
```

14.6 Practice task

np_matplotlib.py is a program that uses Numpy and matplotlib to draw graphs of powers of 1 to 4.

Exercise 14-1 Modify it to draw a Fourier approximation of a saw wave.

In Numpy (np) you can use np.pi for pi and np.sin() for the sine function.

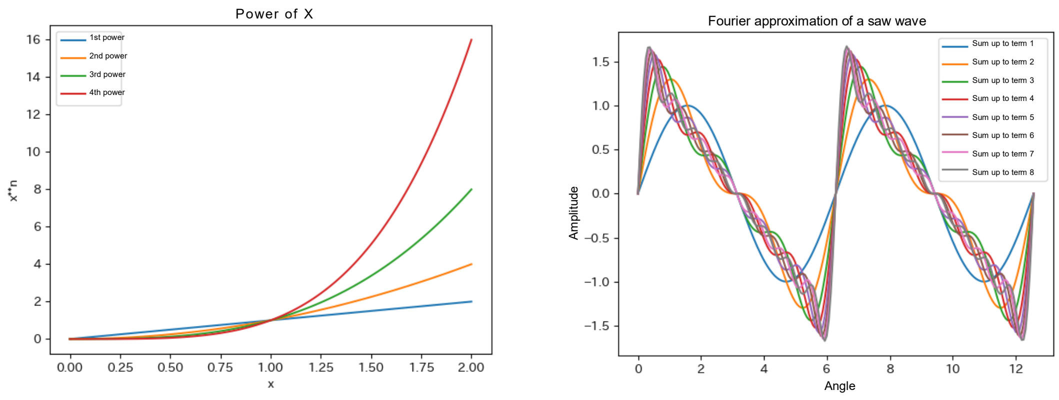


Figure 14-7 A power graph and an approximation of a sawtooth wave by summing trigonometric functions

Program 14-7 Drawing graphs of powers of 1 to 4 with Numpy and Matplotlib (use_matplotlib_power_function_en.py)

Row	Source code	Explanation
1	#␣Example␣of␣plotting␣data␣in␣Numpy	
2	import␣matplotlib	Preparing␣matplotlib
3	matplotlib.use(’tkagg’)	
4	import␣matplotlib.pyplot␣as␣plt	Preparing␣numpy
5	import␣numpy␣as␣np	Setting␣the␣font␣in␣matplotlib
6	#␣For␣Windows	
7	matplotlib.rc(’font’,␣**{’family’:’Yu␣Gothic’})	
8	#␣For␣Campus␣PC␣Terminal	
9	#matplotlib.rc(’font’,␣**{’family’:’IPAPGothic’})	
10	#␣For␣macOS	
11	#matplotlib.rc(’font’,␣**{’family’:’Hiragino␣Maru␣Gothic␣Pro’})	
12	#	
13	#␣Plot␣x␣to␣the␣1st␣to␣4th␣power	
14	#	
15	steps␣=␣100	
16	order␣=␣4	
17	maxx␣=␣2	
18	#	
19	#␣Create␣a␣matrix␣with␣a␣steps␣row␣and␣order␣column␣with␣element␣values␣set␣to␣0	
20	#	
21	datalist␣=␣np.zeros((steps,␣order))	
22	#	
23	#␣List␣for␣the␣legend	
24	#	

Next Chapter Table of Contents

```
legend_label=[]
#
#␣Create␣a␣value␣for␣x␣in␣linspace
#
x␣=␣np.linspace(0,maxx,steps)
#
#␣For␣each␣column,␣calculate␣everything␣at␣once
#␣
for␣j␣in␣range(1,order+1):
␣␣␣␣datalist[:,j-1]␣=␣x**j
␣␣␣␣legend_label.append('Power␣of'␣+␣str(j))
#
#␣Plot
#
plt.plot(x,␣datalist)
plt.title('Power␣function␣of␣x')
plt.xlabel('x')
plt.ylabel('x**n')
plt.legend(legend_label)
plt.show()
```

References

[18] Wes McKinney: Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython, O'Reilly Media, 2nd Ed. (2017)

The tutorial pages on the following sites are helpful.

[19] NumPy website
http://www.numpy.org/index.html

[20] Pandas website
https://pandas.pydata.org/

[21] Matplotlib website
https://matplotlib.org/

[22] TkAgg button controls, etc. (hard to find)
https://matplotlib.org/users/navigation_toolbar.html

15. Review, and Where to Go From Here

15.1 Learning goals of this chapter

1. This is the last part of the course. In this chapter, we will look back at what we have learned so far.
2. We learned how to use the IDLE integrated environment with Python exercises. We chose IDLE because it has limited functionality and is easy to learn, but we will also learn about what else is available here.
3. We will then think about how to apply what you have learned.

15.2 Reflection

Please reflect on your learning by comparing your skills before and after this class.

- What are you now able to do?
- How was it different from your expectations before the course?
- What kind of learning objectives will you set in the future?

15.3 *Shuhari* (three stages of mastery) - “Obey, Digress, Separate”

At the end of the semester, we asked students who had taken this class for their opinions. Several of them said, “I can read Python programs now, but I don't feel like I can write them myself.” In fact, “being able to read” is a big step forward, and if you can run the program while understanding it, it will be easy to modify it a little. As you gain more experience, you will gradually be able to write your own programs.

In Japanese martial arts, we use the term *Shuhari* (lit. “Obey, Digress, Separate”). In programming, if you read the programs written by your predecessors, execute them, play around with them by changing them slightly, and apply them, you will accumulate knowledge and eventually be able to do creative programming.

15.4 Python environments

In this class, we used IDLE as an integrated Python environment because of its simple structure and ease of use for beginners. IDLE is said to be a “throw-away” environment because of its simplicity. Python, on the other hand, has a variety of usage environments.

- Jupyter Notebook and Spider are included in Anaconda.
- These environments run in IPython, a more interactive execution environment than the Python

shell.

- A style of editors and command line execution suited to Python (python, IPython) is also used.

15.5 Adding modules

One of the main features of Python is that many people have developed a variety of libraries in the language. There is a lot of information on the web and elsewhere about specific application examples, but in order to use them you need to add live modules: in Anaconda you can use the conda command to add modules that are not covered by anaconda, or the pip command if you use Python as a distribution package. In Anaconda, use the conda command, and if you are using Python as a distribution package, use the pip command to add modules.

15.6 Topics not covered in this book

Python is a programming language with a wide range of applications, but each application requires knowledge of the relevant area of application. NumPy and pandas, which were introduced in this book, cannot be used without knowledge of numerical computation and statistical processing. For this reason, we have not covered some topics that require knowledge of related fields. The following are some examples. we hope you will study them according to your own interests.

- Network and web-related topics
- Multimedia topics such as image processing
- Topics related to databases
- Topics related to artificial intelligence such as machine learning

15.7 Gratitude and repayment - how to make use of what you have learned

There are many people who do carpentry at home. If you can do carpentry, you can solve problems in your home by making simple furniture and other DIY things. On the other hand, working with metal, which requires machines such as drilling machines, lathes, and milling machines, can be a bit more challenging. So, there are not many people who try to make DIY metal products[1].

If you are able to program computers, not just in Python, we believe that you will be able to **look at things in a way** that says, "**I should be able to do this with a computer**." If this is the case for you, we urge you to think about how you can contribute to society through computers and programming.

Computers, programming languages, and other software, including Python, are "other people's creations;" they are a gift from many engineers and programmers, so to speak. If

[1]When I was in junior high school, me and a friend of mine whose family owned an ironworks thought about making a bicycle together. What you think that you can do depends on the environment you are in.

you enjoy programming, be thankful for this and try to give back.

16. Useful notes on Python and IDLE

16.1 Useful notes on Python

- help() function:Allows you to read the description of a module or function supplied as an argument.
- globals(): Displays globally defined variables and other information.
- id(x): Shows the identity of object x. You can check if different variables point to the same object.
- type(x): Shows the type of object x. You can check what type of object is assigned to the variable.

16.2 Pay attention to file names

Do not create files with file names that are the same as modules to be imported.

Python searches for modules in a specific folder. The same folder as the file to be executed is the target of the module search. For example, if you are using the turtle module and you have a file named turtle.py, Python will incorrectly assume that this file is a module.

16.3 IDLE notes: Python shell hotkeys

- Ctrl-C: Stop a running program
- Ctrl-D: (when input in the terminal) End file
  - ➢ Note: In the shell interactive mode, this will exit the shell.
- TAB key:Smart indent
  - ➢ If pressed after a character, it will display auto-complete suggestions
- ALT-P: Go back in history (you can reuse lines you have already typed, etc. P stands for previous)
- ALT-N: Go forward in history (you can reuse lines you've already typed; N stands for next)
- Script execution
  - ➢ After executing a script created in the editor, the environment will be in the interactive mode. You can call functions and check global variables in the script.

16.4 IDLE notes: Editor

Hotkeys

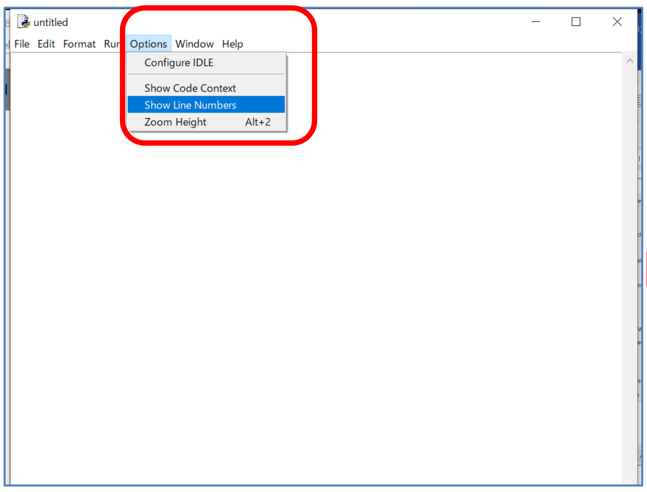
- Ctrl-]: Indent the selected range
- Ctrl-[: Unindent the indentation of the selected range
- ALT-3: Comment out the selected range
- ALT-4: Undo comment out for the selected range
- Ctrl-BS: Delete the word to the left
- Ctrl-Del: Delete the word to the right

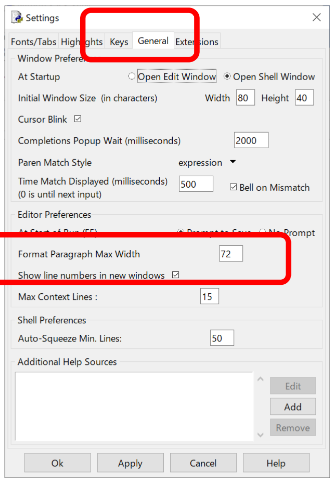
The following hotkey operations generally available in Windows are also useful.

- Ctrl-X: Cut
- Ctrl-C: Copy
- Ctril-V: Paste

16.4.1 Show Line Number

Since Python 3.8, the IDLE editor can show line numbers next to the source code, by selecting Show Line Number from either the Options menu, or by selecting it under the General tab in Configure IDLE menu.




Settings

Fonts/Tabs | Highlights | Keys | General | Extensions

Window Preferences

At Startup ○ Open Edit Window ◉ Open Shell Window

Initial Window Size (in characters) Width 80 Height 40

Cursor Blink ☑

Completions Popup Wait (milliseconds) 2000

Paren Match Style expression

Time Match Displayed (milliseconds) (0 is until next input) 500 ☑ Bell on Mismatch

Editor Preferences

At Start of Run (F5) ◉ Prompt to Save ○ No Prompt

Format Paragraph Max Width 72

Show line numbers in new windows ☑

Max Context Lines : 15

Shell Preferences

Auto-Squeeze Min. Lines: 50

Additional Help Sources

Also, command line arguments can now be passed at runtime by selecting the Run ... Customized menu. (Command line arguments are not explained in this text.)

17. How to Read Error Messages in IDLE/Python

As a program gets more complex, even a simple mistake in copying and executing the source code can result in a variety of errors. There are two ways that errors are displayed in IDLE:

- Syntax errors in the source code displayed by the IDLE editor
- Run-time errors displayed by the Python shell

Let's take a look at some of the most common errors and what they mean. **An error occurs when the computer that interprets and executes the program is unable to continue processing. In many cases, the error occurs in a different place than the actual error caused by the programmer**. You need to read error messages and think about what they mean.

17.1 Errors displayed by the IDLE editor

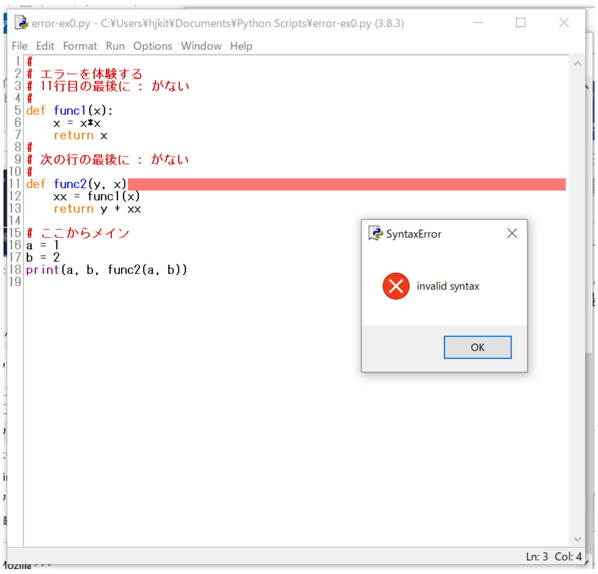
Syntax errors are checked before execution and displayed by the IDLE editor.

17.1.1 Example Program 1 - Missing Colon

Program 17-1 missing_colon_error.py

Row	Source code	Notes
1	`#`	
2	`#␣This␣code␣will␣give␣you␣some␣experience␣with␣errors`	
3	`#␣There␣is␣no␣colon␣(:)␣at␣the␣end␣of␣line␣11`	
4	`#`	
5	`def␣func1(x):`	
6	`␣␣␣␣x␣=␣x*x`	
7	`␣␣␣␣return␣x`	
8	`#`	
9	`#␣There␣is␣no␣colon␣(:)␣at␣the␣end␣of␣the␣next␣line`	
10	`#`	
11	`def␣func2(y,␣x)`	You need to use a colon (:) at the end here
12	`␣␣␣␣xx␣=␣func1(x)`	
13	`␣␣␣␣return␣y␣+␣xx`	
14		
15	`#␣Main␣from␣here`	
16	`a␣=␣1`	
17	`b␣=␣2`	
18	`print(a,␣b,␣func2(a,␣b))`	

In this example, the dialog "Invalid syntax" will be displayed and the offending part of the code will be shown in red, as shown in the following figure.


error-ex0.py - C:¥Users¥hjkit¥Documents¥Python Scripts¥error-ex0.py (3.8.3)
File Edit Format Run Options Window Help
SyntaxError
OK
Ln: 3 Col: 4

```
#
# エラーを体験する
# 11行目の最後に : がない
#
def func1(x):
    x = x*x
    return x
#
# 次の行の最後に : がない
#
def func2(y, x)
    xx = func1(x)
    return y + xx

# ここからメイン
a = 1
b = 2
print(a, b, func2(a, b))
```

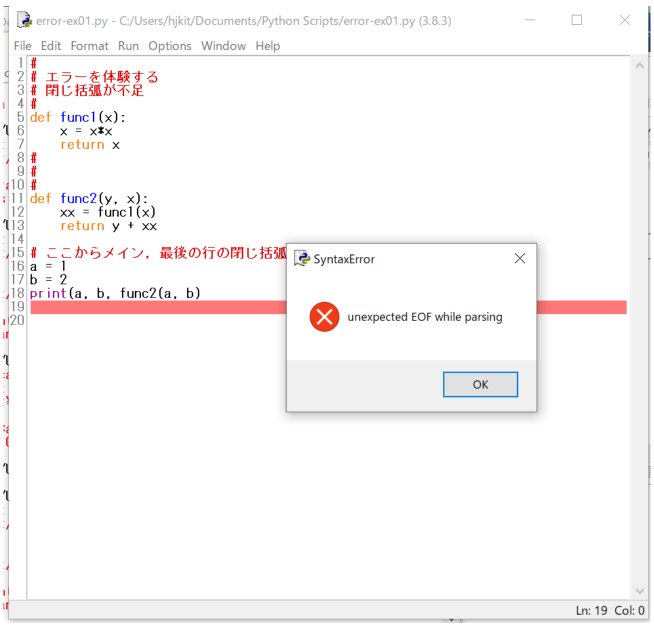
invalid syntax

17.1.2 Example Program 2 - Missing Parentheses

Program 17-2 missing_parentheses_error.py

Row	Source code	Notes
1	#	
2	#␣This␣code␣will␣give␣you␣some␣experience␣with␣errors	
3	#␣Missing␣a␣closing␣parenthesis	
4	#	
5	def␣func1(x):	
6	␣␣␣␣x␣=␣x*x	
7	␣␣␣␣return␣x	
8	#	
9	#␣	
10	#	
11	def␣func2(y,␣x):	
12	␣␣␣␣xx␣=␣func1(x)	
13	␣␣␣␣return␣y␣+␣xx	
14		
15	#␣The␣main␣part␣starts␣here;␣the␣last␣line␣is␣missing␣a␣closing␣parenthesis	
16	a␣=␣1	
17	b␣=␣2	
18	print(a,␣b,␣func2(a,␣b)	Missing a closing parenthesis

In this example, the dialog "unexpected EOF while parsing (EOF, end of file)" is shown, and the corresponding part is displayed in red, as shown in the following figure. This means that the source code has been terminated while looking for a pair of opening and closing parentheses; the warning is given after line 19, where the actual error is located.


error-ex01.py - C:/Users/hjkit/Documents/Python Scripts/error-ex01.py (3.8.3)

```
#
# エラーを体験する
# 閉じ括弧が不足
#
def func1(x):
    x = x*x
    return x
#
#
#
def func2(y, x):
    xx = func1(x)
    return y + xx

# ここからメイン, 最後の行の閉じ括弧
a = 1
b = 2
print(a, b, func2(a, b)
```

SyntaxError

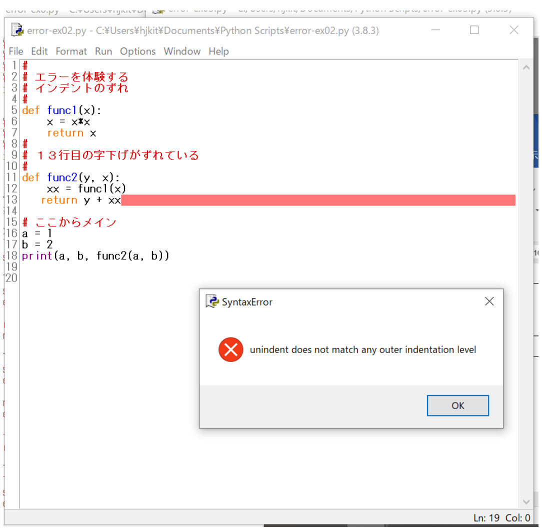
unexpected EOF while parsing

17.1.3 Example Program 3 - Indentation Offset (Insufficiency)

Program 17-3 insufficient_indentation_error.py

Row	Source code	Notes
1	#	
2	#␣This␣code␣will␣give␣you␣some␣experience␣with␣errors	
3	#␣Indentation␣offset	
4	#	
5	def␣func1(x):	
6	␣␣␣␣x␣=␣x*x	
7	␣␣␣␣return␣x	
8	#	
9	#␣The␣indent␣of␣line␣13␣is␣offset	
10	#	
11	def␣func2(y,␣x):	
12	␣␣␣␣xx␣=␣func1(x)	
13	␣␣␣return␣y␣+␣xx	This indent needs one more space
14		
15	#␣Main␣from␣here	
16	a␣=␣1	
17	b␣=␣2	
18	print(a,␣b,␣func2(a,␣b))	

In this example, the indentation of line 13 is missing one character, so the message "unindent does not match outer indentation level" is displayed. It warns that the indentation is insufficient, and that there is no matching level.


error-ex02.py - C:¥Users¥hjkit¥Documents¥Python Scripts¥error-ex02.py (3.8.3)
File Edit Format Run Options Window Help
Ln: 19 Col: 0
OK

```
#
# エラーを体験する
# インデントのずれ
#
def func1(x):
    x = x*x
    return x
#
# １３行目の字下げがずれている
#
def func2(y, x):
    xx = func1(x)
   return y + xx

# ここからメイン
a = 1
b = 2
print(a, b, func2(a, b))
```

SyntaxError

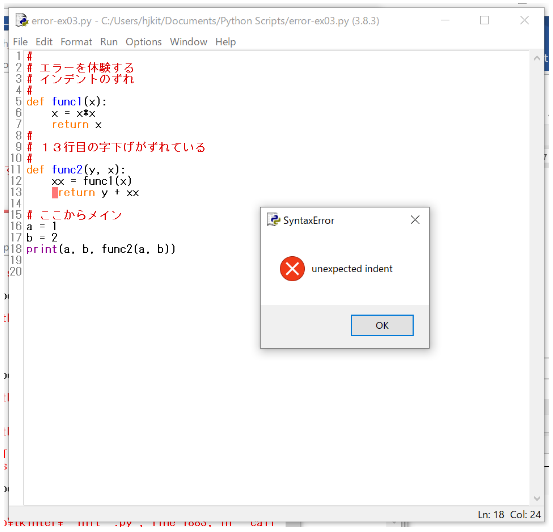
unindent does not match any outer indentation level

17.1.4 Example Program 4 - Indentation Offset (Excess)

Program 17-4 excess_indentation_error.py

Row	Source code	Notes
1	#	
2	#␣This␣code␣will␣give␣you␣some␣experience␣with␣errors	
3	#␣Indentation␣offset	
4	#	
5	def␣func1(x):	
6	␣␣␣␣x␣=␣x*x	
7	␣␣␣␣return␣x	
8	#	
9	#␣The␣indent␣of␣line␣13␣is␣offset	
10	#	
11	def␣func2(y,␣x):	
12	␣␣␣␣xx␣=␣func1(x)	
13	␣␣␣␣␣return␣y␣+␣xx	There is an extra space
14		here
15	#␣Main␣from␣here	
16	a␣=␣1	
17	b␣=␣2	
18	print(a,␣b,␣func2(a,␣b))	

This time the indentation has one character too many, so the error says “unexpected indent.” This can be interpreted to mean that the indentation is unexpected because there is no block that requires further indentation.


error-ex03.py - C:/Users/hjkit/Documents/Python Scripts/error-ex03.py (3.8.3)
File Edit Format Run Options Window Help
SyntaxError
OK
Ln: 18 Col: 24

```
#
# エラーを体験する
# インデントのずれ
#
def func1(x):
    x = x*x
    return x
#
# １３行目の字下げがずれている
#
def func2(y, x):
    xx = func1(x)
     return y + xx

# ここからメイン
a = 1
b = 2
print(a, b, func2(a, b))
```

```
unexpected indent
```

17.2 Errors displayed in Python shell when executing code

17.2.1 Example Program 5 - Referencing an Undefined Variable

Program 17-5 referencing_undefined_variable_error.py

Row	Source code	Notes
1	#	
2	#␣This␣code␣will␣give␣you␣some␣experience␣with␣errors	
3	#␣Line␣6␣refers␣to␣a␣variable␣that␣is␣not␣defined	
4	#	
5	def␣func1(x):	
6	␣␣␣␣x␣=␣xx␣#␣This␣variable␣does␣not␣exist	Refers to undefined variable xx
7	␣␣␣␣return␣x	
8		
9	def␣func2(y,␣x):	
10	␣␣␣␣xx␣=␣func1(x)	
11	␣␣␣␣return␣y␣+␣xx	
12		
13	#␣Main␣from␣here	
14	a␣=␣1	
15	b␣=␣2	
16	print(a,␣b,␣func2(a,␣b))	

In this example, the Python shell displays the following. Since the error is occurring in the function definition, the traceback will follow the call to the error location. On line 6, it says "NameError: name 'xx' is not defined," indicating that the variable called xx is not defined.

```
Traceback (most recent call last):
  File "M:\Documents\Python Scripts\error-ex1.py", line 16, in <module>
    print(a, b, func2(a, b))
  File "M:\Documents\Python Scripts\error-ex1.py", line 10, in func2
    xx = func1(x)
  File "M:\Documents\Python Scripts\error-ex1.py", line 6, in func1
    x = xx # This variable does not exist
NameError: name 'xx' is not defined
>>>
```

17.2.2 Example Program 6 - Wrong Argument Type

Program 17-6 wrong_argument_type_error.py

Row	Source code	Notes
1	#␣This␣code␣will␣give␣you␣some␣experience␣with␣errors	
2	#␣When␣calling␣math.sin(),␣the␣argument␣is␣a␣string	
3	#	
4	import␣math	
5	def␣func1(x):	
6	␣␣␣␣xx␣=␣math.sin(x)	
7	␣␣␣␣return␣xx	The value of x is the string "2".
8		
9	def␣func2(y,␣x):	
10	␣␣␣␣xx␣=␣func1(x)	
11	␣␣␣␣return␣y␣+␣xx	
12		
13	#␣Main␣from␣here	
14	a␣=␣1	
15	b␣=␣"2"	
16	print(a,␣b,␣func2(a,␣b))	set b a string "2" and pass to func2

In this example, the string "2" is passed as an argument to call math.sin(), and the message "TypeError: must be real number, not str" is displayed.

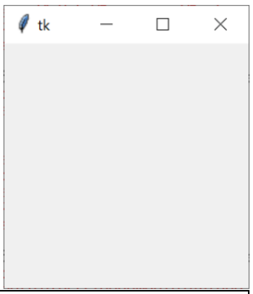
```
Traceback (most recent call last):
  File "M:\Documents\Python Scripts\error-ex2.py", line 17, in <module>
    print(a, b, func2(a, b))
  File "M:\Documents\Python Scripts\error-ex2.py", line 11, in func2
    xx = func1(x)
  File "M:\Documents\Python Scripts\error-ex2.py", line 7, in func1
    xx = math.sin(x)
TypeError: must be real number, not str
>>>
```

17.2.3 Example Program 7 - Incorrect Indentation in a Class

Program 17-7 incorrect_indatation_in_class_error.py

Row	Source code	Notes
1	`#`	
2	`#␣This␣code␣will␣give␣you␣some␣experience␣with␣errors`	
3	`#␣Indentation␣within␣a␣class`	
4	`#`	
5	`import␣math`	
6	`import␣tkinter␣as␣tk`	
7	`class␣MyFrame(tk.Frame):`	
8	`␣␣␣␣def␣__init__(self,␣master=None):`	
9	`␣␣␣␣␣␣␣␣super().__init__(master)`	
10	`␣␣␣␣␣␣␣␣self.b␣=␣tk.Button(self,␣text="Try!",command=self.do)`	
	`␣␣␣␣␣␣␣␣self.b.grid(row=0,␣column=0)`	
11	`#`	
12	`#␣The␣indent␣below␣is␣one␣level␣too␣deep`	
13	`#`	
14	`␣␣␣␣␣␣␣␣def␣do(self):`	
15	`␣␣␣␣␣␣␣␣␣␣␣␣self.b.configure(text="Did")`	This method will end up in the definition of the `__init__` method.
16		
17		
18	`root=␣tk.Tk()`	
19	`f␣=␣MyFrame(root)`	
20	`f.pack()`	
21	`tk.mainloop()`	

This example uses a tkinter class definition that extends the Frame class, but the indentation of the “do” method is one level too deep. This is why MyFrame is assumed to have no attribute called do when called as a button callback function. Also, the creation of the button widget b fails, so only the window of tkinter is displayed, as shown on the right.

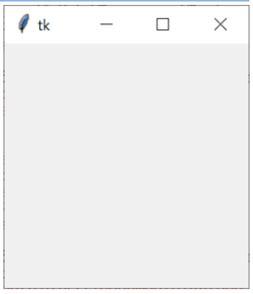


```
Traceback (most recent call last):
  File "M:/Documents/Python Scripts/error-ex3.py", line 20, in <module>
    f = MyFrame(root)
  File "M:/Documents/Python Scripts/error-ex3.py", line 10, in __init__
    self.b = tk.Button(self, text="Try!", command=self.do)
AttributeError: 'MyFrame' object has no attribute 'do'
>>>
```

17.2.4 Example Program 8 - Incorrect Optional Argument

Program 17-8 incorrect_optional_argument_error.py

Row	Source code	Notes
1	`#`	
2	`#␣This␣code␣will␣give␣you␣some␣experience␣with␣errors`	
3	`#␣Incorrect␣widget␣option`	
4	`#`	
5	`import␣math`	
6	`import␣tkinter␣as␣tk`	
7	`class␣MyFrame(tk.Frame):`	
8	`␣␣␣␣def␣__init__(self,␣master=None):`	
9	`␣␣␣␣␣␣␣␣super().__init__(master)`	
10	`␣␣␣␣␣␣␣␣self.b␣=␣tk.Button(self,␣text="Try!",␣commend=self.do)`	it says "commend"
	`␣␣␣␣␣␣␣␣self.b.grid(row=0,␣column=0)`	
11	`␣␣␣␣def␣do(self):`	
12	`␣␣␣␣␣␣␣␣self.b.configure(text="Did")`	
13		
14		
15	`root=␣tk.Tk()`	
16	`f␣=␣MyFrame(root)`	
17	`f.pack()`	
18	`tk.mainloop()`	



This example uses a class definition that extends the Frame class in tkinter, but the optional argument for the button widget in line 10 is spelled incorrectly. The creation of the button widget b will fail, so only the tkinter window will be displayed as shown in the figure on the right. It is difficult to see where the error occurs because it is in the tkinter module, but from the error code 'unknown option "-commend"' you can see that the error occurs when calling line 10, and that an option is wrongly specified.

```
Traceback (most recent call last):
  File "M:/Documents/Python Scripts/error-ex4.py", line 17, in <module>
    f = MyFrame(root)
  File "M:/Documents/Python Scripts/error-ex4.py", line 10, in __init__
    self.b = tk.Button(self, text="Try!", commend=self.do)
  File "M:\anaconda3\lib\tkinter\__init__.py", line 2645, in __init__
    Widget.__init__(self, master, 'button', cnf, kw)
  File "M:\anaconda3\lib\tkinter\__init__.py", line 2567, in __init__
    self.tk.call(
_tkinter.TclError: unknown option "-commend"
>>>
```