CS1302 Introduction to Computer Programming

Chung Chan

Dec 01, 2020

LECTURE NOTES

This is a jupyter book for the course *CS1302 Introduction to Computer Programming* at City University of Hong Kong. The course aims to introduce key concepts, techniques, and good practices of programming using a high-level programming language Python.

Course Intended Learning Outcomes (CILOs):

1. Explain the structure of a computer program.
2. Analyze, test and debug computer programs.
3. Apply proper programming techniques to solve a task.
4. Construct well-structured programs.

CHAPTER

ONE

INTRODUCTION TO COMPUTER PROGRAMMING

1.1 Computer

1.1.1 What is a computer?

Is computer a calculator that is bigger and more advanced?

If computer is a calculator, then, is abacus the first computer invented?

Is your smartphone a computer?What defines a computer?

- In addition to performing arithmetic calculations, a computer is designed in such a way that
- we can write different programs (in a process called **programming** or **software development**)
- for the computer to execute to perform different tasks.

1.1.2 What is the architecture of a computer?

A computer contains three main hardware components:

- Input device
- Processing unit
- Output device

Peripherals

Input and output devices connected to a computer are called *peripherals*.They allow users to interact with the computer in different ways.

Exercise Some examples of output devices are:

- Monitor
- Speaker

Can you give an awesome example in the following box?

- 3D printer available at CityU

Exercise Some examples of input devices are:

- Keyboard
- Mouse

Can you give an awesome example?

- 3D scanner available at CityU

Exercise Many devices are both input and output device. Can you give at least 3 examples?

- hard disk
- CD/DVD Rom (writable)
- touch screen

Central Processing Unit

The brain of a computer is its processor unit, or the **C**entral **P**rocesisng **U**nit (CPU).It is located on the *motherboard* and connects to different peripherals using different *connectors*.

Two important components in the CPU are:

- **A**rithmetic and **L**ogic **U**nit (**ALU**): Performs arithmetics like a calculator (but for binary numbers)
- **C**ontrol **U**nit (**CU**): Directs the operations of the processor in executing a program.

Let's run a CPU Simulator below from a GitHub project.

- Note that all values are zeros in the RAM (**R**andom **A**cess **M**emory) initially.
- Under Settings, click `Examples->Add two numbers`. Observe that the values in the RAM have changed.
- Click `Run` at the bottom right-hand corner.

```
%%html
<iframe src="https://tools.withcode.uk/cpu" width="800" height="800">
</iframe>
```

```
<IPython.core.display.HTML object>
```

1.2 Programming

1.2.1 What is programming?

Programming is the process of writing programs.But what is a program?

Exercise You have just seen a program written in machine language. Where is it?

The first six lines of binary sequences in the RAM. The last line `Ends` the program.

- The CPU is capable of carrying out
  - a set of instructions such as *addition*, *negation*, *Copy*, etc.
  - some numbers stored in the RAM.
- Both the instructions and the numbers are represented as binary sequences.
- E.g., in Intel-based CPU, the command for addition is like **00000011 00000100**

1.2.2 Why computer uses binary representation?

```
%%html
<iframe width="912" height="513" src="https://www.youtube.com/embed/Xpk67YzOn5w"␣
↪allowfullscreen></iframe>
```

```
<IPython.core.display.HTML object>
```

Exercise The first electronic computer, called Electronic Numerical Integrator and Computer (ENIAC), was programmed using binary circuitries, namely *switches* that can be either `On` or `Off`.

However, it did not represent values efficiently in binary. 10 binary digits (bits) was used to represent a decimal number 0 to 9.Indeed, how many decimals can be represented by 10 bits?

```
2 ** 10 # because there are that many binary sequences of length 10.
```

```
1024
```

As mentioned in the video, there are *International Standards* for representing characters:

- ASCII (American Standard Code for Information Interchange) maps English letters and some other symbols to 8-bits (8 binary digits, also called a byte). E.g., `A` is `01000001`.
- Unicode can also represent characters in different languages such as Chinese, Japanese…etc.

There are additional standards to represent numbers other than non-negative integers:

- 2's complement format for negative integers (e.g. -123)
- IEEE floating point format for floating point numbers such as $1.23x10^{-4}$.

Why define different standards?

- Different standards have different benefits. ASCII requires less storage for a character, but it represents less characters.
- Although digits are also represented in ASCII, the 2's complement format is designed for arithmetic operations.

1.3 Different generations of programming languages

Machine language is known as the **1st Generation Programming Language**.

Are we going to start with machine language? Start with learning 2's complement and the binary codes for different instructions?

No. Programmers do not write machine codes directly because it is too hard to think in binary representations.

Instead, programmers write human-readable **mnemonics** such as **ADD**, **SUB**…,called **Assembly language**, or the **2nd Generation Programming Language**.

Are you going to learn an assembly language?

Both machine language and assembly language are low-level language which

- are difficult to write for complicated tasks (requiring many lines of code), and
- are platform specific:
  - the sets of instructions and their binary codes can be different for different types of CPUs, and
  - different operating systems use different assembly languages/styles.

Anyone want to learn assembly languages, and write a program in many versions to support different platforms?

Probably for programmers who need to write fast or energy-efficient code such as

- a driver that controls a 3D graphics card, and
- a program that control a microprocessor with limited power supply.

But even in the above cases, there are often better alternatives. Play with the following microprocessor simulator:

- Click `CHOOSE A DEMO->LED`.
- Click `RUN SCRIPT` and observes the LED of the board.
- Run the demos `ASSEMBLY` and `MATH` respectively and compare their capabilities.

```
%%html
<iframe width="900", height="1000" src="https://micropython.org/unicorn/"></iframe>
```

```
<IPython.core.display.HTML object>
```

1.4 High-level Language

```
%%html
<iframe width="912" height="513" src="https://www.youtube.com/embed/QdVFvsCWXrA"␣
↪allowfullscreen></iframe>
```

```
<IPython.core.display.HTML object>
```

Programmer nowadays use human-readable language known as the **3rd generation language (3GL)** or **high-level language.**

- Examples includes: C, C++, Java, JavaScript, Basic, Python, PHP, …

1.4.1 What is a high-level language?

- A code written in high-level language gets converted automatically to a low-level machine code for the desired platform.
- Hence, it *abstracts* away details that can be handled by the computer (low-level code) itself.

For instance, a programmer needs not care where a value should be physically stored if the computer can find a free location automatically to store the value.

Different high-level languages can have different implementations of the conversion processes:

- **Compilation** means converting a program well before executing of the program. E.g., C++ and Java programs are compiled.
- **Interpretation** means converting a program on-the-fly during the execution of a program. E.g., JavaScript and Python programs are often interpreted.

Roughly speaking, compiled programs run faster but interpreted programs are more flexible and can be modified at run time.(The truth is indeed more complicated than required for this course.)

1.4.2 What programming language will you learn?

You will learn to program using **Python**. The course covers:

- Basic topics including *values*, *variables*, *conditional*, *iterations*, *functions*, *composite data types*.
- Advanced topics that touch on functional and object-oriented programming.
- Engineering topics such as *numerical methods*, *optimizations*, and *machine learning*.

See the course homepage for details.

Why Python?

```
%%html
<iframe width="912" height="513" src="https://www.youtube.com/embed/Y8Tko2YC5hA?
↪end=200" allowfullscreen></iframe>
```

```
<IPython.core.display.HTML object>
```

In summary:

- Python is expressive and can get things done with fewer lines of code as compared to other languages.
- Python is one of the most commonly used languages. It has an extensive set of libraries for Mathematics, graphics, AI, Machine Learning, etc.
- Python is Free and Open Source, so you get to see everything and use it without restrictions.
- Python is portable. The same code runs in different platforms without modifications.

How does a Python program look like?

```
# for step-by-step execution using mytutor
%reload_ext mytutor
```

```
%%mytutor -h 400
# The program here reads the cohort and reports which year the user is in
# Assumption: Input is integer no greater than 2020
import datetime  # load a library to tell the current year
cohort = input("In which year did you join CityU? ")
year = datetime.datetime.now().year - int(cohort) + 1
print("So you are a year", year, "student.")
```

```
<IPython.lib.display.IFrame at 0x7f7f2c675d68>
```

A Python program contains *statements* just like sentences in natural languages.E.g., `cohort = input("In which year did you join CityU? ")` is a statement that gives some value a name called `cohort`.

For the purpose of computations, a statement often contains *expressions* that evaluate to certain values.E.g., `input("In which year did you join CityU? ")` is an expression with the value equal to what you input to the prompt.That value is then given the name `cohort`.

Expressions can be composed of:

- *Functions* such as `input`, `now`, and `int`, etc., which are like math functions the return some values based on its arguments, if any.
- *Literals* such as the string `"In which year did you join CityU? "` and the integer `1`. They are values you type out literally.
- *Variables* such as `cohort` and `year`, which are meaningful names to values.

To help others understand the code, there are also *comments* that start with #.These are descriptions meant for human to read but not to be executed by the computer.

Exercise What do you think the next generation programmimng should be?

Perhaps programming using natural languages. Write programs that people enjoy reading, like literate programming.Indeed, Jupyter notebook is arguably a step towards this direction. See nbdev.

```
%%html
<iframe width="912" height="513" src="https://www.youtube.com/embed/bTkXg2LZIMQ"␣
↪allowfullscreen></iframe>
```

```
<IPython.core.display.HTML object>
```

CHAPTER

TWO

VALUES AND VARIABLES

2.1 Integers

How to enter an integer in a program?

```
15  # an integer in decimal
```

```
15
```

```
0b1111  # a binary number
```

```
15
```

```
0xF  # hexadecimal (base 16) with possible digits 0, 1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
```

```
15
```

Why all outputs are the same?

- What you have entered are *integer literals*, which are integers written out literally.
- All the literals have the same integer value in decimal.
- By default, if the last line of a code cell has a value, the jupyter notebook (*IPython*) will store and display the value as an output.

```
3  # not the output of this cell
4 + 5 + 6
```

```
15
```

- The last line above also has the same value, `15`.
- It is an *expression* (but not a literal) that *evaluates* to the integer value.

Exercise Enter an expression that evaluates to an integer value, as big as possible.(You may need to interrupt the kernel if the expression takes too long to evaluate.)

```
# There is no maximum for an integer for Python3.
# See https://docs.python.org/3.1/whatsnew/3.0.html#integers
11 ** 100000
```

2.2 Strings

How to enter a string in a program?

```
'\U0001f600: I am a string.'  # a sequence of characters delimited by single quotes.
```

```
'😀: I am a string.'
```

```
"\N{grinning face}: I am a string."  # delimited by double quotes.
```

```
'😀: I am a string.'
```

```
"""\N{grinning face}: I am a string."""  # delimited by triple single/double quotes.
```

```
'😀: I am a string.'
```

- \ is called the *escape symbol*.
- \U0001f600 and \N{grinning face} are *escape sequences*.
- These sequences represent the same grinning face emoji by its Unicode in hexadecimal and its name.

Why use different quotes?

```
print('I\'m line #1.\nI\'m line #2.')  # \n is a control code for line feed
print("I'm line #3.\nI'm line #4.")  # no need to escape single quote.
print('''I'm line #5.
I'm line #6.''')  # multi-line string
```

```
I'm line #1.
I'm line #2.
I'm line #3.
I'm line #4.
I'm line #5.
I'm line #6.
```

Note that:

- The escape sequence \n does not represent any symbol.
- It is a *control code* that creates a new line when printing the string.
- Another common control code is \t for tab.

Using double quotes, we need not escape the single quote in I'm.

Triple quotes delimit a multi-line string, so there is no need to use \n.(You can copy and paste a multi-line string from elsewhere.)

In programming, there are often many ways to do the same thing.The following is a one-line code (one-liner) that prints multiple lines of strings without using \n:

```
print("I'm line #1", "I'm line #2", "I'm line #3", sep='\n')  # one liner
```

```
I'm line #1
I'm line #2
I'm line #3
```

- `sep='\n'` is a *keyword argument* that specifies the separator of the list of strings.
- By default, `sep=' '`, a single space character.

In IPython, we can get the *docstring* (documentation) of a function conveniently using the symbol `?`.

```
?print
```

```
print?
```

Exercise Print a cool multi-line string below.

```
print('''
 (ง •̀_•́)ง
 ┗(●’◡’●)┓
 (..•˘_˘•..)
 (ง'̀-'́ 3ง)ง
''')
# See also https://github.com/glamp/bashplotlib
# Star Wars via Telnet http://asciimation.co.nz/
```

```
 (ง •̀_•́)ง
 ┗(●’◡’●)┓
 (..•˘_˘•..)
 (ง'̀-'́ 3ง)ง
```

2.3 Variables and Assignment

It is useful to store a value and retrieve it later.To do so, we assign the value to a variable:

```
x = 15
x  # output the value of x
```

```
15
```

Is assignment the same as equality?

No because:

- you cannot write `15 = x`, but
- you can write `x = x + 1`, which increases the value of `x` by `1`.

Exercise Try out the above code yourself.

```
x = x + 1
x
```

```
16
```

Let's see the effect of assignment step-by-step:

1. Run the following cell.
2. Click `Next >` to see the next step of the execution.

```
%%mytutor -h 200
x = 15
x = x + 1
```

```
<IPython.lib.display.IFrame at 0x7ff9204f8390>
```

The following *tuple assignment* syntax can assign multiple variables in one line.

```
%%mytutor -h 200
x, y, z = '15', '30', 15
```

```
<IPython.lib.display.IFrame at 0x7ff9204f86d8>
```

One can also use *chained assignment* to set different variables to the same value.

```
%%mytutor -h 250
x = y = z = 0
```

```
<IPython.lib.display.IFrame at 0x7ff9204f89b0>
```

Variables can be deleted using `del`. Accessing a variable before assignment raises a Name error.

```
del x, y
x, y
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-20-7a8db3f963f3> in <module>
----> 1 del x, y
      2 x, y

NameError: name 'y' is not defined
```

2.4 Identifiers

Identifiers such as variable names are case sensitive and follow certain rules.

What is the syntax for variable names?

1. Must start with a letter or _ (an underscore) followed by letters, digits, or _.
2. Must not be a keyword (identifier reserved by Python):

Exercise Evaluate the following cell and check if any of the rules above is violated.

```
from ipywidgets import interact
@interact
def identifier_syntax(assignment=['a-number = 15',
                      'a_number = 15',
                      '15 = 15',
                      '_15 = 15',
                      'del = 15',
                      'Del = 15',
                      'type = print',
```

(continues on next page)

(continued from previous page)

```
                    'print = type',
                    'input = print']):
    exec(assignment)
    print('Ok.')
```

```
interactive(children=(Dropdown(description='assignment', options=('a-number = 15', 'a_
↪number = 15', '15 = 15',…
```

1. `a-number = 15` violates Rule 1 because - is not allowed. - is interpreted as an operator.
2. `15 = 15` violates Rule 1 because `15` starts with a digit instead of letter or _.
3. `del = 15` violates Rule 2 because `del` is a keyword.

What can we learn from the above examples?

- `del` is a keyword and `Del` is not because identifiers are case sensitive.
- Function/method/type names `print`/`input`/`type` are not keywords and can be reassigned.This can useful if you want to modify the default implementations without changing their source code.

To help make code more readable, additional style guides such as PEP 8 are available:

- Function names should be lowercase, with words separated by underscores as necessary to improve readability.
- Variable names follow the same convention as function names.

2.5 User Input

How to let the user input a value at *runtime*,i.e., as the program executes?

We can use the method `input`:

- There is no need to delimit the input string by quotation marks.
- Simply press `enter` after typing a string.

```
print('Your name is', input('Please input your name: '))
```

- The `input` method prints its argument, if any, as a prompt.
- It takes user's input and *return* it as its value. `print` takes in that value and prints it.

Exercise Explain whether the following code prints `'My name is Python'`. Does `print` return a value?

```
print('My name is', print('Python'))
```

```
Python
My name is None
```

- Unlike `input`, the function `print` does not return the string it is trying to print. Printing a string is, therefore, different from returning a string.
- `print` actually returns a `None` object that gets printed as `None`.

2.6 Type Conversion

The following program tries to compute the sum of two numbers from user inputs:

```
num1 = input('Please input an integer: ')
num2 = input('Please input another integer: ')
print(num1, '+', num2, 'is equal to', num1 + num2)
```

Exercise There is a bug in the above code. Can you locate the error?

The two numbers are concatenated instead of added together.

`input` *returns* user input as a string.E.g., if the user enters `12`, the input is

- not treated as the integer twelve, but rather
- treated as a string containing two characters, one followed by two.

To see this, we can use `type` to return the data type of an expression.

```
num1 = input('Please input an integer: ')
print('Your input is', num1, 'with type', type(num1))
```

Exercise `type` applies to any expressions. Try it out below on `15`, `print`, `print()`, `input`, and even `type` itself and `type(type)`.

```
type(15), type(print), type(print()), type(input), type(type), type(type(type))
```

```

```

```
(int, builtin_function_or_method, NoneType, method, type, type)
```

So what happens when we add strings together?

```
'4' + '5' + '6'
```

```
'456'
```

How to fix the bug then?

We can convert a string to an integer using `int`.

```
int('4') + int('5') + int('6')
```

```
15
```

We can also convert an integer to a string using `str`.

```
str(4) + str(5) + str(6)
```

```
'456'
```

Exercise Fix the bug in the following cell.

```
num1 = input('Please input an integer: ')
num2 = input('Please input another integer: ')
# print(num1, '+', num2, 'is equal to', num1 + num2)  # fix this line below
```

(continues on next page)

(continued from previous page)

```
### BEGIN SOLUTION
print(num1, '+', num2, 'is equal to', int(num1) + int(num2))
### END SOLUTION
```

2.7 Error

In addition to writing code, a programmer spends significant time in *debugging* code that contains errors.

Can an error be automatically detected by the computer?

- You have just seen an example of *logical error*, which is due to an error in the logic.
- The ability to debug or even detect such error is, unfortunately, beyond Python's intelligence.

Other kinds of error may be detected automatically.As an example, note that we can omit + for string concatenation, but we cannot omit it for integer summation:

```
print('Skipping + for string concatenation')
'4' '5' '6'
```

```
Skipping + for string concatenation
```

```
'456'
```

```
print('Skipping + for integer summation')
4 5 6
```

```
  File "<ipython-input-32-234998ea6ba8>", line 2
    4 5 6
      ^
SyntaxError: invalid syntax
```

Python interpreter detects the bug and raises a *syntax* error.

Why Syntax error can be detected automatically?Why is the print statement before the error not executed?

- The Python interpreter can easily detect syntax error even before executing the code simply because
- the interpreter fails to interpret the code, i.e., translates the code to lower-level executable code.

The following code raises a different kind of error.

```
print("Evaluating '4' + '5' + 6")
'4' + '5' + 6  # summing string with integer
```

```
Evaluating '4' + '5' + 6
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-33-67f2880d2e89> in <module>
      1 print("Evaluating '4' + '5' + 6")
----> 2 '4' + '5' + 6  # summing string with integer

TypeError: can only concatenate str (not "int") to str
```

Why Python throws a TypeError when evaluating `'4' + '5' + 6`?

There is no default implementation of + operation on a value of type `str` and a value of type `int`.

- Unlike syntax error, the Python interpreter can only detect type error at runtime (when executing the code.)
- Hence, such error is called a *runtime error*.

Why is TypeError a runtime error?

The short answer is that Python is a strongly-and-dynamically-typed language:

- Strongly-typed: Python does not force a type conversion to avoid a type error.
- Dynamically-typed: Python allow data type to change at runtime.

The underlying details are more complicated than required for this course. It helps if you already know the following languages:

- JavaScript, which is a *weakly-typed* language that forces a type conversion to avoid a type error.
- C, which is a *statically-typed* language that does not allow data type to change at runtime.

```
%%javascript
alert('4' + '5' + 6)  // no error because 6 is converted to a str automatically
```

```
<IPython.core.display.Javascript object>
```

A weakly-typed language may seem more robust, but it can lead to more logical errors.To improve readability, typescript is a strongly-typed replacement of javascript.

Exercise Not all the strings can be converted into integers. Try breaking the following code by providing invalid inputs and record them in the subsequent cell. Explain whether the errors are runtime errors.

```
num1 = input('Please input an integer: ')
num2 = input('Please input another integer: ')
print(num1, '+', num2, 'is equal to', int(num1) + int(num2))
```

The possible invalid inputs are:

`4 + 5 + 6`, `15.0`, `fifteen`

It raises a value error, which is a runtime error detected during execution.

Note that the followings are okay

int('-1'), eval('4 + 5 + 6')

2.8 Floating Point Numbers

Not all numbers are integers. In Enginnering, we often need to use fractions.

How to enter fractions in a program?

```
x = -0.1 # decimal number
y = -1.0e-1 # scientific notation
z = -1/10 # fraction
x, y, z, type(x), type(y), type(z)
```

```
(-0.1, -0.1, -0.1, float, float, float)
```

What is the type `float`?

- `float` corresponds to the *floating point* representation.
- A `float` in stored exactly the way we write it in scientific notation:

$$\overbrace{-}^{\text{sign}}\underbrace{1.0}_{\text{mantissa}}e\overbrace{-1}^{\text{exponent}} = -1 \times 10^{-1}$$

- The truth is more complicated than required for the course.

Integers in mathematics may be regarded as a `float` instead of `int`:

```
type(1.0), type(1e2)
```

```
(float, float)
```

You can also convert an `int` or a `str` to a `float`.

```
float(1), float('1')
```

```
(1.0, 1.0)
```

Is it better to store an integer as `float`?

Python stores a floating point with finite precision (usually as a 64bit binary fraction):

```
import sys
sys.float_info
```

```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.
↪2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53,␣
↪epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

It cannot represent a number larger than the `max`:

```
sys.float_info.max * 2
```

```
inf
```

The precision also affects the check for equality.

```
(1.0 == 1.0 + sys.float_info.epsilon * 0.5, # returns true if equal
 1.0 == 1.0 + sys.float_info.epsilon * 0.6, sys.float_info.max + 1 == sys.float_info.
↪max)
```

```
(True, False, True)
```

Another issue with float is that it may keep more decimal places than desired.

```
1/3
```

```
0.3333333333333333
```

How to round a floating point number to the desired number of decimal places?

```
round(2.665,2), round(2.675,2)
```

```
(2.67, 2.67)
```

Why 2.675 rounds to 2.67 instead of 2.68?

- A `float` is actually represented in binary.
- A decimal fraction may not be represented exactly in binary.

The `round` function can also be applied to an integer.

```
round(150,-2), round(250,-2)
```

```
(200, 200)
```

Why 250 rounds to 200 instead of 300?

- Python 3 implements the default rounding method in IEEE 754.

2.9 String Formatting

Can we round a `float` or `int` for printing but not calculation?

This is possible with *format specifications*.

```
x = 10000/3
print('x ≈ {:.2f} (rounded to 2 decimal places)'.format(x))
x
```

```
x ≈ 3333.33 (rounded to 2 decimal places)
```

```
3333.3333333333335
```

- `{:.2f}` is a *format specification*
- that gets replaced by a string
- that represents the argument `x` of `format`
- as a decimal floating point number rounded to 2 decimal places.

Exercise Play with the following widget to learn the effect of different format specifications. In particular, print `10000/3` as `3,333.33`.

```
from ipywidgets import interact
@interact(x='10000/3',
          align={'None':'','<':'<','>':'>','=':'=','^':'^'},
          sign={'None':'','+':'+','-':'-','SPACE':' '},
          width=(0,20),
          grouping={'None':'','_':'_',',':','},
          precision=(0,20))
def print_float(x,sign,align,grouping,width=0,precision=2):
    format_spec = f"{{:{align}{sign}{'' if width==0 else width}{grouping}.{precision}
↪f}}"
    print("Format spec:",format_spec)
    print("x ≈",format_spec.format(eval(x)))
```

```
interactive(children=(Text(value='10000/3', description='x'), Dropdown(description=
↪'sign', options={'None': ''…
```

```
print('{:,.2f}'.format(10000/3))
```

```
3,333.33
```

String formatting is useful for different data types other than `float`.E.g., consider the following program that prints a time specified by some variables.

```
# Some specified time
hour = 12
minute = 34
second = 56

print("The time is " + str(hour) + ":" + str(minute) + ":" + str(second)+".")
```

```
The time is 12:34:56.
```

Imagine you have to show also the date in different formats.The code can become very hard to read/write because

- the message is a concatenation of multiple strings and
- the integer variables need to be converted to strings.

Omitting + leads to syntax error. Removing `str` as follows also does not give the desired format.

```
print("The time is ", hour, ":", minute, ":", second, ".")  # note the extra spaces
```

```
The time is  12 : 34 : 56 .
```

To make the code more readable, we can use the `format` function as follows.

```
message = "The time is {}:{}:{}."
print(message.format(hour,minute,second))
```

```
The time is 12:34:56.
```

- We can have multiple *place-holders* `{}` inside a string.
- We can then provide the contents (any type: numbers, strings..) using the `format` function, which
- substitutes the place-holders by the function arguments from left to right.

According to the string formatting syntax, we can change the order of substitution using

- indices *(0 is the first item)* or
- names inside the placeholder `{}`:

```
print("You should {0} {1} what I say instead of what I {0}.".format("do", "only"))
print("The surname of {first} {last} is {last}.".format(first="John", last="Doe"))
```

```
You should do only what I say instead of what I do.
The surname of John Doe is Doe.
```

You can even put variables inside the format specification directly and have a nested string formatting.

```
align, width = "^", 5
print(f"{{:*{align}{width}}}".format(x))  # note the syntax f"..."
```

```
3333.3333333333335
```

Exercise Play with the following widget to learn more about the formating specification.

1. What happens when `align` is none but `fill` is `*`?
2. What happens when the `expression` is a multi-line string?

```
from ipywidgets import interact
@interact(expression=r"'ABC'",
          fill='*',
          align={'None':'','<':'<','>':'>','=':'=','^':'^'},
          width=(0,20))
def print_objectt(expression,fill,align='^',width=10):
    format_spec = f"{{:{fill}{align}{'' if width==0 else width}}}"
    print("Format spec:",format_spec)
    print("Print:",format_spec.format(eval(expression)))
```

```
interactive(children=(Text(value="'ABC'", description='expression'), Text(value='*',␣
↪description='fill'), Drop…
```

1. It returns a ValueError because align must be specified when fill is.
2. The newline character is simply regarded a character. The formatting is not applied line-by-line. E.g., try 'ABC\nDEF'.

CHAPTER

THREE

EXPRESSIONS AND ARITHMETIC

3.1 Operators

The followings are common operators you can use to form an expression in Python:

Operator	Operation	Example
unary –	Negation	`-y`
`+`	Addition	`x + y`
–	Subtraction	`x - y`
`*`	Multiplication	`x*y`
`/`	Division	`x/y`

- `x` and `y` in the examples are called the *left and right operands* respectively.
- The first operator is a *unary operator*, which operates on just one operand.(+ can also be used as a unary operator, but that is not useful.)
- All other operators are *binary operators*, which operate on two operands.

Python also supports some more operators such as the followings:

Operator	Operation	Example
`//`	Integer division	`x//y`
`%`	Modulo	`x%y`
`**`	Exponentiation	`x**y`

```
# ipywidgets to demonstrate the operations of binary operators
from ipywidgets import interact
binary_operators = {'+':' + ','-':' - ','*':'*','/':'/','//':'//','%':'%','**':'**'}
@interact(operand1=r'10',
          operator=binary_operators,
          operand2=r'3')
def binary_operation(operand1,operator,operand2):
    expression = f"{operand1}{operator}{operand2}"
    value = eval(expression)
    print(f"""{'Expression:':>11} {expression}\n{'Value:':>11} {value}\n{'Type:':>11}
↪{type(value)}""")
```

```
interactive(children=(Text(value='10', description='operand1'), Dropdown(description=
↪'operator', options={'+':…
```

Exercise What is the difference between `/` and `//`?

- `/` is the usual division, and so `10/3` returns the floating-point number $3.\dot{3}$.
- `//` is integer division, and so `10//3` gives the integer quotient 3.

What does the modulo operator `%` do?

You can think of it as computing the remainder, but the truth is more complicated than required for the course.

Exercise What does `'abc' * 3` mean? What about `10 * 'a'`?

- The first expression means concatenating `'abc'` three times.
- The second means concatenating `'a'` ten times.

Exercise How can you change the default operands (`10` and `3`) for different operators so that the overall expression has type `float`.Do you need to change all the operands to `float`?

- `/` already returns a `float`.
- For all other operators, changing at least one of the operands to `float` will return a `float`.

3.2 Operator Precedence and Associativity

An expression can consist of a sequence of operations performed in a row such as `x + y*z`.

How to determine which operation should be performed first?

Like arithmetics, the order of operations is decided based on the following rules applied sequentially:

1. *grouping* by parentheses: inner grouping first
2. operator *precedence/priority*: higher precedence first
3. operator *associativity*:
    - left associativity: left operand first
    - right associativity: right operand first

What are the operator precedence and associativity?

The following table gives a concise summary:

Operators	Associativity
`**`	right
− (unary)	right
`*`,`/`,`//`,`%`	left
+,−	left

Exercise Play with the following widget to understand the precedence and associativity of different operators.In particular, explain whether the expression `-10 ** 2*3` gives $(-10)^{2\times 3} = 10^6 = 1000000$.

```
from ipywidgets import fixed
@interact(operator1={'None':'','unary -':'-'},
          operand1=fixed(r'10'),
          operator2=binary_operators,
          operand2=fixed(r'2'),
          operator3=binary_operators,
          operand3=fixed(r'3')
          )
def three_operators(operator1,operand1,operator2,operand2,operator3,operand3):
```

(continues on next page)

(continued from previous page)

```
    expression = f"{operator1}{operand1}{operator2}{operand2}{operator3}{operand3}"
    value = eval(expression)
    print(f"""{'Expression:':>11} {expression}\n{'Value:':>11} {value}\n{'Type:':>11}
↪{type(value)}""")
```

```
interactive(children=(Dropdown(description='operator1', options={'None': '', 'unary -
↪': '-'}, value=''), Dropd…
```

The expression evaluates to $(-(10^2)) \times 3 = -300$ instead because the exponentiation operator `**` has higher precedence than both the multiplication `*` and the negation operators `-`.

Exercise To avoid confusion in the order of operations, we should follow the style guide when writing expression.What is the proper way to write `-10 ** 2*3`?

```
print(-10**2 * 3)  # can use use code-prettify extension to fix incorrect styles
print((-10)**2 * 3)
```

```
-300
```

3.3 Augmented Assignment Operators

- For convenience, Python defines the augmented assignment operators such as +=, where
- `x += 1` means `x = x + 1`.

The following widgets demonstrate other augmented assignment operators.

```
from ipywidgets import interact, fixed
@interact(initial_value=fixed(r'10'),
          operator=['+=','-=','*=','/=','//=','%=','**='],
          operand=fixed(r'2'))
def binary_operation(initial_value,operator,operand):
    assignment = f"x = {initial_value}\nx {operator} {operand}"
    _locals = {}
    exec(assignment,None,_locals)
    print(f"""Assignments:\n{assignment:>10}\nx: {_locals['x']} ({type(_locals['x'])})
↪""")
```

```
interactive(children=(Dropdown(description='operator', options=('+=', '-=', '*=', '/=
↪', '//=', '%=', '**='), v…
```

Exercise Can we create an expression using (augmented) assignment operators? Try running the code to see the effect.

```
3*(x = 15)
```

```
  File "<ipython-input-62-566073d2d63b>", line 1
    3*(x = 15)
       ^
SyntaxError: invalid syntax
```

Assignment operators are used in assignment statements, which are not expressions because they cannot be evaluated.

CHAPTER

FOUR

CONDITIONAL EXECUTION

4.1 Motivation

Conditional execution means running different pieces of code based on different conditions. Why?

For instance, when trying to compute `a/b`, `b` may be `0` and division by `0` is invalid.

```
def multiply_or_divide(a, b):
    print('a:{}, b:{}, a*b:{}, a/b:{}'.format(a, b, a * b, a / b))


multiply_or_divide(1, 2)
multiply_or_divide(1, 0)  # multiplication is valid but not shown
```

```
a:1, b:2, a*b:2, a/b:0.5
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-c6f0ad2b2b09> in <module>
      4
      5 multiply_or_divide(1, 2)
----> 6 multiply_or_divide(1, 0)  # multiplication is valid but not shown

<ipython-input-2-c6f0ad2b2b09> in multiply_or_divide(a, b)
      1 def multiply_or_divide(a, b):
----> 2     print('a:{}, b:{}, a*b:{}, a/b:{}'.format(a, b, a * b, a / b))
      3
      4
      5 multiply_or_divide(1, 2)

ZeroDivisionError: division by zero
```

Can we skip only the division but not multiplication when `b` is `0`?

```
def multiply_or_divide(a, b):
    fix = a / b if b else 'undefined'
    print('a:{}, b:{}, a*b:{}, a/b:{}'.format(a, b, a * b, fix))


multiply_or_divide(1, 2)
multiply_or_divide(1, 0)  # multiplication is valid but not shown
```

```
a:1, b:2, a*b:2, a/b:0.5
a:1, b:0, a*b:0, a/b:undefined
```

The above solution involve:

- a *boolean expression* `fix` that checks whether a condition holds, and
- a *conditional construct* `... if ... else ...` that specify which code block should be executed under what condition.

4.2 Boolean expressions

4.2.1 Comparison Operators

How to compare different values?

Like the equality and inequality relationships in mathematics,Python also have binary *comparison/relational operators*:

Expression	True iff
`x == y`	$x = y$.
`x < y`	$x < y$.
`x <= y`	$x \leq y$.
`x > y`	$x > y$.
`x >= y`	$x \geq y$.
`x != y`	$x \neq y$.

Explore these operators using the widgets below:

```
# Comparisons
from ipywidgets import interact
comparison_operators = ['==','<','<=','>','>=','!=']
@interact(operand1='10',
          operator=comparison_operators,
          operand2='3')
def comparison(operand1,operator,operand2):
    expression = f"{operand1} {operator} {operand2}"
    value = eval(expression)
    print(f"""{'Expression:':>11} {expression}\n{'Value:':>11} {value}\n{'Type:':>11}
↪{type(value)}""")
```

```
interactive(children=(Text(value='10', description='operand1'), Dropdown(description=
↪'operator', options=('=='…
```

- These operators return either `True` or `False`, which are `keywords` of type *boolean*.
- The expressions are called *boolean expressions* or *predicates*, named after George Boole.
- N.b., the equality operator == consists of *two equal signs*, different from the assignment operator =.

What is the precedence of comparison operators?

All the comparison operators have the same precedence lower than that of + and −.

```
1 + 2 >= 3  # (1 + 2) >= 3
```

```
True
```

Python allows multiple comparison operations to be chained together:

```
2.0 == 2>1 #equivalent to (2.0 ==2) and (2>1)
```

```
True
```

What is the associativity?

Comparison operations are *non-associative*:

```
(2.0 == 2) > 1, 2.0 == (2 > 1)  # not the same as 2.0 == 2 > 1
```

```
(False, False)
```

Errorata in [Halterman17] due to a misunderstanding of non-associativity vs left-to-right evaluation order:

- Halterman17, p.69:

  > The relational operators are binary operators and are all ~left associative~ **non-associative**.

- Halterman17, p.50, Table 3.2:
  - = should be non-associative instead of right-associative.
  - The corresponding table in `Lecture2/Expressions and Arithmetic.ipynb` should also be corrected accordingly.

Exercise Explain why the following boolean expressions have different values.

```
1 <= 2 < 3 != 4, (1 <= 2) < (3 != 4)
```

```
(True, False)
```

The second expression is not a chained comparison:

- The expressions in the parentheses are evaluated to boolean values first to `True`, and so
- the overall expression `True < True` is evaluated to `False`.

Exercise The comparison operators can be applied to different data types, as illustrated below.Explain the meaning of the operators in each of the following expressions.

```
# Comparisons beyond numbers
@interact(expression=[
    '10 == 10.', '"A" == "A"', '"A" == "A "', '"A" != "a"',
    '"A" > "a"', '"aBcd" < "abd"', '"A" != 64', '"A" < 64'
])
def relational_expression(expression):
    print(eval(expression))
```

```
interactive(children=(Dropdown(description='expression', options=('10 == 10.', '"A"␣
↪== "A"', '"A" == "A "', '"…
```

1. Checks whether an integer is equal to a floating point number.
2. Checks whether two characters are the same.
3. Checks whether two strings are the same. Note the space character.

4. Checks whether a character is larger than the order character according to their unicodes.
5. Checks whether a string is lexicographically smaller than the other string.
6. Checks whether a character is not equal to an integer.
7. TypeError because there is no implementation that evaluates whether a string is smaller than an integer.

Is `!` the same as the `not` operator?

Errata There is an error in Halterman17, p.69 due to confusion with C language:

> ... `!(x >= 10)` and `!(10 <= x)` are ~equivalent~ **invalid**.

- We can write `1 != 2` as `not 1 == 2` but not `!(1 == 2)` because
- `!` is not a logical operator. It is used to call a system shell command in IPython.

```
!(1 == 2)
```

```
/bin/bash: 1: command not found
```

```
!ls  # a bash command that lists files in the current directory
```

```
'Conditional Execution.ipynb'                         Iteration.ipynb
'Conditional Execution.ipynb:Zone.Identifier'   Iteration.ipynb:Zone.Identifier
```

How to compare floating point numbers?

```
x = 10
y = (x**(1/3))**3
x == y
```

```
False
```

Why False? Shouldn't $(x^{\frac{1}{3}})^3 = x$?

- Floating point numbers have finite precisions and so
- we should instead check whether the numbers are close enough.

One method of comparing floating point numbers:

```
abs(x - y) <= 1e-9
```

```
True
```

`abs` is a function that returns the absolute value of its argument. Hence, the above translates to

$$|x - y| \leq \delta_{\text{abs}}$$

or equivalently

$$y - \delta_{\text{abs}} \leq x \leq y + \delta_{\text{abs}}$$

where δ_{abs} is called the *absolute tolerance*.

Is an absolute tolerance of `1e-9` good enough?

What if we want to compare `x = 1e10` instead of `10`?

```
x = 1e10
y = (x**(1/3))**3
abs(x - y) <= 1e-9
```

```
False
```

Floating point numbers "float" at different scales.A better way to use the `isclose` function from `math` module.

```
import math
math.isclose(x, y)
```

```
True
```

How does it work?

`math.isclose(x,y)` implements

$$|x-y| \leq \max\{\delta_{\text{rel}} \max\{|x|,|y|\}, \delta_{\text{abs}}\}$$

with the default

- *relative tolerance* δ_{rel} equal to `1e-9`, and
- absolute tolerance δ_{abs} equal to `0.0`.

Exercise Write the boolean expression implemented by `isclose`. You can use the function `max(a,b)` to find the maximum of `a` and `b`.

```
rel_tol, abs_tol = 1e-9, 0.0
x, y = 1e-100, 2e-100
### BEGIN SOLUTION
abs(x-y) <= max(rel_tol * max(abs(x), abs(y)), abs_tol)
### END SOLUTION
```

```
False
```

4.2.2 Boolean Operations

Since chained comparisons are non-associative. It follows a different evaluation rule than arithmetical operators.

E.g., `1 <= 2 < 3 != 4` is evaluated as follows:

```
1 <= 2 and 2 < 3 and 3 != 4
```

```
True
```

The above is called a *compound boolean expression*, which is formed using the *boolean/logical operator* `and`.

Why use boolean operators?

What if we want to check whether a number is either < 0 or ≥ 100?Can we achieve this only by chaining the comparison operators or applying the logical `and`?

```
# Check if a number is outside a range.
@interact(x='15')
def check_out_of_range(x):
```

(continues on next page)

(continued from previous page)

```
    x_ = float(x)
    is_out_of_range = x_<0 or x_>=100
    print('Out of range [0,100):', is_out_of_range)
```

```
interactive(children=(Text(value='15', description='x'), Output()), _dom_classes=(
↪'widget-interact',))
```

- `and` alone is not functionally complete, i.e., not enough to give all possible boolean functions.
- In addition to `and`, we can also use `or` and `not`.

x	y	x and y	x or y	not x
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

The above table is called a *truth table*. It enumerates all possible input and output combinations for each boolean operator.

How are chained logical operators evaluated?What are the precedence and associativity for the logical operators?

- All binary boolean operators are left associative.
- Precedence: `comparison operators` > `not` > `and` > `or`

Exercise Explain what the values of the following two compound boolean expressions are:

- Expression A: `True or False and True`
- Expression B: `True and False and True`
- Expression C: `True or True and False`
- Expression A evaluates to `True` because `and` has higher precedence and so the expression has the same value as `True or (False and True)`.
- Expression B evaluates to `False` because `and` is left associative and so the expression has the same value as `(True and False) and True`.
- Expression C evaluates to `True` because `and` has a higher precedence and so the expression has the same value as `True or (True and False)`. Note that `(True or True) and False` evaluates to something `False` instead, so precedence matters.

Instead of following the precedence and associativity, however, a compound boolean expression uses a short-circuit evaluation.

To understand this, we will use the following function to evaluate a boolean expression verbosely.

```
def verbose(id,boolean):
    '''Identify evaluated boolean expressions.'''
    print(id,'evaluated:',boolean)
    return boolean
```

```
verbose('A',verbose(1,True) or verbose(2,False) and verbose(3,True))  # True or␣
↪(False and True)
```

```
1 evaluated: True
A evaluated: True
```

```
True
```

Why expression 2 and 3 are not evaluated?

Because True or … must be True (Why?) so Python does not look further. From the documentation:

> The expression `x or y` first evaluates `x`; if `x` is true, its value is returned; otherwise, `y` is evaluated and the resulting value is returned.

Note that:

- Even though `or` has lower precedence than `and`, it is still evaluated first.
- The evaluation order for logical operators is left-to-right.

```
verbose('B',verbose(4,True) and verbose(5,False) and verbose(6,True))  # (True and
↪False) and True
```

```
4 evaluated: True
5 evaluated: False
B evaluated: False
```

```
False
```

Why expression 6 is not evaluated?

`True and False and ...` must be `False` so Python does not look further.

> The expression `x and y` first evaluates `x`; if `x` is false, its value is returned; otherwise, `y` is evaluated and the resulting value is returned.

Indeed, logical operators can even be applied to non-boolean operands. From the documentation:

> In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: `False`, None, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true.

Exercise How does the following code work?

```
print('You have entered', input() or 'nothing')
```

- The code replaces empty user input by the default string `nothing` because empty string is regarded as False in a boolean operation.
- If user input is non-empty, it is regarded as True in the boolean expression and returned immediately as the value of the boolean operation.

Is empty string equal to False?

```
print('Is empty string equal False?',''==False)
```

```
Is empty string equal False? False
```

- An empty string is regarded as False in a boolean operation but
- a *comparison operation is not a boolean operation*, even though it forms a boolean expression.

4.3 Conditional Constructs

Consider writing a program that sorts values in *ascending* order.A *sorting algorithm* refers to the procedure of sorting values in order.

4.3.1 If-Then Construct

How to sort two values?

Given two values are stored as `x` and `y`, we want to

- `print(x,y)` if `x <= y`, and
- `print(y,x)` if `y < x`.

Such a program flow is often represented by a flowchart like the following:

Python provides the `if` statement to implement the above *control flow* specified by the diamonds.

```
# Sort two values using if statement
def sort_two_values(x, y):
    if x <= y:
        print(x, y)
    if y < x: print(y, x)


@interact(x='1', y='0')
def sort_two_values_app(x, y):
    sort_two_values(eval(x), eval(y))
```

```
interactive(children=(Text(value='1', description='x'), Text(value='0', description='y
↪'), Output()), _dom_clas…
```

We can visualize the execution as follows:

```
%%mytutor -h 350
def sort_two_values(x, y):
    if x <= y:
        print(x, y)
    if y < x: print(y, x)

sort_two_values(1,0)
sort_two_values(1,2)
```

```
<IPython.lib.display.IFrame at 0x7f283814b358>
```

Python use indentation to indicate code blocks or *suite*:

- `print(x, y)` (Line 5) is indented to the right of `if x <= y:` (Line 4) to indicate it is the body of the if statement.
- For convenience, `if y < x: print(y, x)` (Line 6) is a one-liner for an `if` statement that only has one line in its block.
- Both `if` statements (Line 4-6) are indented to the right of `def sort_two_values(x,y):` (Line 3) to indicate that they are part of the body of the function `sort_two_values`.

How to indent?

- The style guide recommends using 4 spaces for each indentation.
- In IPython, you can simply type the `tab` key and IPython will likely enter the correct number of spaces for you.

What if you want to leave a block empty?

In programming, it is often useful to delay detailed implementations until we have written an overall skeleton.To leave a block empty, Python uses the keyword `pass`.

```
# write a code skeleton
def sort_two_values(x, y):
    pass
    # print the smaller value first followed by the larger one

sort_two_values(1,0)
sort_two_values(1,2)
```

Without `pass`, the code will fail to run, preventing you from checking other parts of the code.

```
# You can add more details to the skeleton step-by-step
def sort_two_values(x, y):
    if x <= y:
        pass
        # print x before y
    if y < x: pass  # print y before x

sort_two_values(1,0)
sort_two_values(1,2)
```

4.3.2 If-Then-Else Construct

The sorting algorithm is not efficient enough. Why not?Hint: `(x <= y) and not (y < x)` is a *tautology*, i.e., always true.

To improve the efficient, we should implement the following program flow.

This can be down by the `else` clause of the `if` statement.

```
%%mytutor -h 350
def sort_two_values(x, y):
    if x <= y:
        print(x, y)
    else:
        print(y,x)

sort_two_values(1,0)
sort_two_values(1,2)
```

```
<IPython.lib.display.IFrame at 0x7f282b7be6a0>
```

We can also use a *conditional expression* to shorten the code.

```
def sort_two_values(x, y):
    print(('{0} {1}' if x <= y else '{1} {0}').format(x, y))


@interact(x='1', y='0')
```

(continues on next page)

(continued from previous page)

```
def sort_two_values_app(x, y):
    sort_two_values(eval(x), eval(y))
```

```
interactive(children=(Text(value='1', description='x'), Text(value='0', description='y
↪'), Output()), _dom_clas…
```

Exercise Explain why the followings have syntax errors.

```
1 if True
```

```
  File "<ipython-input-30-1cbab0db8442>", line 1
    1 if True
            ^
SyntaxError: invalid syntax
```

```
x = 1 if True else x = 0
```

```
  File "<ipython-input-31-f68814adf81f>", line 1
    x = 1 if True else x = 0
        ^
SyntaxError: can't assign to conditional expression
```

A conditional expression must be an expression:

1. It must give a value under all cases. To enforce that, `else` keyword must be provided.
2. An assignment statement does not return any value and therefore cannot be used for the conditional expression.`x = 1 if True else 0` is valid because `x =` is not part of the conditional expression.

4.3.3 Nested Conditionals

Consider sorting three values instead of two. A feasible algorithm is as follows:

We can implement the flow using *nested conditional constructs*:

```
def sort_three_values(x, y, z):
    if x <= y <= z:
        print(x, y, z)
    else:
        if x <= z <= y:
            print(x, z, y)
        else:
            if y <= x <= z:
                print(y, x, z)
            else:
                if y <= z <= x:
                    print(y, z, x)
                else:
                    if z <= x <= y:
                        print(z, x, y)
                    else:
                        print(z, y, x)

def test_sort_three_values():
```

(continues on next page)

(continued from previous page)

```
    sort_three_values(0,1,2)
    sort_three_values(0,2,1)
    sort_three_values(1,0,2)
    sort_three_values(1,2,0)
    sort_three_values(2,0,1)
    sort_three_values(2,1,0)

test_sort_three_values()
```

```
0 1 2
0 1 2
0 1 2
0 1 2
0 1 2
0 1 2
```

Imagine what would happen if we have to sort many values.To avoid an excessively long line due to the indentation, Python provides the `elif` keyword that combines `else` and `if`.

```
def sort_three_values(x, y, z):
    if x <= y <= z:
        print(x, y, z)
    elif x <= z <= y:
        print(x, z, y)
    elif y <= x <= z:
        print(y, x, z)
    elif y <= z <= x:
        print(y, z, x)
    elif z <= x <= y:
        print(z, x, y)
    else:
        print(z, y, x)


test_sort_three_values()
```

```
0 1 2
0 1 2
0 1 2
0 1 2
0 1 2
0 1 2
```

Exercise The above sorting algorithm is inefficient because some conditions may be checked more than once.Improve the program to eliminate duplicate checks.*Hint:* Do not use chained comparison operators or compound boolean expressions.

```
def sort_three_values(x, y, z):
    if x <= y:
        if y <= z:
            print(x, y, z)
        elif x <= z:
            print(x, z, y)
        else:
            print(z, x, y)
    ### BEGIN SOLUTION
    elif z <= y:
```

(continues on next page)

(continued from previous page)

```
        print(z, y, x)
    elif z <= x:
        print(y, z, x)
    else:
        print(y, x, z)
    ### END SOLUTION

sort_three_values(10,17,14)
```

```
10 14 17
```

CHAPTER

FIVE

ITERATION

5.1 Motivation

Many tasks are repetitive:

- To print from 1 up to a user-specified number, which can be arbitrarily large.
- To compute the maximum of a sequence of numbers, which can be arbitrarily long.
- To repeatedly ask users for input until the input is within the right range.

How to write code to perform repetitive tasks?

E.g., can you complete the following code to print from 1 up to a user-specified number?

```
%%mytutor -h 300
num = int(input('>'))
if 1 < num: print(1)
if 2 < num: print(2)
if 3 < num: print(3)
# YOUR CODE HERE
```

```
<IPython.lib.display.IFrame at 0x7fed686d4f28>
```

code duplication is not good because:

- Duplicate code is hard to read/write/maintain.Imagine there is a small change needed to every duplicate code.
- The number of repetitions may not be known before runtime.

Instead, programmers write a *loop* which specifies a piece of code to be executed iteratively.

5.2 For Loop

5.2.1 Iterate over a sequence

How to print from 1 up to 4?

We can use a `for` statement as follows:

```
%%mytutor -h 300
for i in 1, 2, 3, 4:
    print(i)
```

```
<IPython.lib.display.IFrame at 0x7fed687726a0>
```

- `i` is automatically assigned to each element in the sequence `1, 2, 3, 4` one-by-one from left to right.
- After each assignment, the body `print(i)` is executed.

N.b., if `i` is defined before the for loop, its value will be overwritten.

The assignment is not restricted to integers and can also be a tuple assignment.

```
tuples = (0,'l'), (1,'o'), (2,'o'), (3,'p')
for i,c in tuples: print(i,c)  # one-liner
```

```
0 l
1 o
2 o
3 p
```

An even shorter code…

```
for i,c in enumerate('loop'): print(i,c)
```

```
0 l
1 o
2 o
3 p
```

5.2.2 Iterate over a range

How to print up to a user-specified number?

We can use `range`:

```
stop = int(input('>')) + 1
for i in range(stop):
    print(i)
```

Why add 1 to the user input number?

`range(stop)` generates a sequence of integers from `0` up to *but excluding* `stop`.

How to start from a number different from `0`?

```
for i in range(1,5): print(i)
```

```
1
2
3
4
```

What about a step size different from `1`?

```
for i in range(0,5,2): print(i)  # starting number must also be specified. Why?
```

```
0
2
4
```

Exercise How to count down from 4 to 0? Do it without addition or subtraction.

```
### BEGIN SOLUTION
for i in range(4,-1,-1): print(i)
### END SOLUTION
```

```
4
3
2
1
0
```

Exercise Print from `0` to a user-specified number but in steps of `0.5`.E.g., if the user inputs `2`, the program should print:

```
0.0
0.5
1.0
1.5
2.0
```

Note: `range` only accepts integer arguments.

```
num = int(input('>'))
### BEGIN SOLUTION
for i in range(0, 2 * num + 1, 1):
    print(i / 2)
### END SOLUTION
```

Exercise How to print the character `'*'` repeatedly for `m` rows and `n` columns?*Hint:* Use a *nested for loop*, i.e., write a for loop (called *inner loop*) inside the body of another for loop (*outer loop*).

```
@interact(m=(0, 10), n=(0, 10))
def draw_rectangle(m=5, n=5):
    ### BEGIN SOLUTION
    for i in range(m):
        for j in range(n):
            print('*', end='')
        print()
    ### END SOLUTION
```

```
interactive(children=(IntSlider(value=5, description='m', max=10), IntSlider(value=5,␣
↪description='n', max=10)…
```

5.2.3 Iterate over a string

What does the following do?

```
%%mytutor -h 300
for character in 'loop': print(character)
```

```
<IPython.lib.display.IFrame at 0x7fed686114e0>
```

A string is *iterable* because it can be regarded as a sequence of characters.

- The function `len` can return the length of a string.

- The indexing operator `[]` can return the character of a string at a specified location.

```
message = "loop"
print('length:', len(message))
print('characters:', message[0], message[1], message[2], message[3])
```

```
length: 4
characters: l o o p
```

We can also iterate over a string as follows although it is less elegant:

```
for i in range(len('loop')): print('loop'[i])
```

```
l
o
o
p
```

Exercise Print a string assigned to `message` in reverse.E.g., `'loop'` should be printed as `'pool'`.

```
@interact(message='loop')
def reverse_print(message):
    ### BEGIN SOLUTION
    for i in range(len(message)):
        print(message[-i - 1], end='')
    ### END SOLUTION
```

```
interactive(children=(Text(value='loop', description='message'), Output()), _dom_
↪classes=('widget-interact',))
```

5.3 While Loop

How to repeatedly ask the user to enter an input until the user input is not empty?

Python provides the `while` statement to loop until a specified condition is false.

```
while not input('Input something please:'): pass
```

As long as the condition after `while` is true, the body gets executed repeatedly. In the above example,

- if user press enter without inputting anything,
- `input` returns an empty string `''`, which is regarded as `False`, and so
- the looping condition `not input('...')` is `True`.

Is it possible to use a for loop instead of a while loop?

- Not without hacks because the for loop is a *definite loop* which has a definite number of iterations before the execution of the loop.
- `while` statement is useful for an *indefinite loop* where the number of iterations is unknown before the execution of the loop.

It is possible, however, to replace a for loop by a while loop.E.g., the following code prints from `0` to `4` using a while loop instead of a for loop.

```
i = 0
while i <= 4:
    print(i)
    i += 1
```

```
0
1
2
3
4
```

- A while loop may not be as elegant (short), c.f., `for i in range(5): print(i)`, but
- it can always be as efficient.

Should we just use while loop?

Consider using the following while loop to print from `0` to a user-specified value.

```
num = int(input('>'))
i = 0
while i!=num+1:
    print(i)
    i += 1
```

Exercise Is the above while loop doing the same thing as the for loop below?

```
for i in range(int(input('>')) + 1): print(i)
```

When user input negative integers smaller than or equal to -2,

- the while loop becomes an infinite loop, but
- the for loop terminates without printing any number.

We have to be careful not to create unintended *infinite loops*.The computer can't always detect whether there is an infinite loop. (Why not?)

5.4 Break/Continue/Else Constructs of a Loop

5.4.1 Breaking out of a loop

Is the following an infinite loop?

```
%%mytutor -h 300
while True:
    message = input('Input something please:')
    if message: break
print('You entered:', message)
```

```
<IPython.lib.display.IFrame at 0x7fed68668828>
```

The loop is terminated by the `break` statement when user input is non-empty.

Why is the `break` statement useful?

Recall the earlier `while` loop:

```
%%mytutor -h 300
while not input('Input something please:'): pass
```

```
<IPython.lib.display.IFrame at 0x7fed6861c1d0>
```

This while loop is not useful because it does not store the user input.

Is the `break` statement strictly necessary?

We can avoid `break` statement by using *flags*, which are boolean variables for flow control:

```
%%mytutor -h 350
has_no_input = True
while has_no_input:
    message = input('Input something please:')
    if message: has_no_input = False
print('You entered:', message)
```

```
<IPython.lib.display.IFrame at 0x7fed6861c400>
```

Using flags makes the program more readable, and we can use multiple flags for more complicated behavior.The variable names for flags are often `is_...`, `has_...`, etc.

5.4.2 Continue to Next Iteration

What does the following program do?Is it an infinite loop?

```
%%mytutor -h 300
while True:
    message = input('Input something please:')
    if not message: continue
    print('You entered:', message)
```

```
<IPython.lib.display.IFrame at 0x7fed6861c630>
```

- The program repeatedly ask the user for input.
- If the input is empty, the `continue` statement will skip to the next iteration.
- The loop can only be terminated by interrupting the kernel.
- Such an infinite loop can be useful. E.g., your computer clock continuously updates the current time.

Exercise Is the `continue` statement strictly necessary? Can you rewrite the above program without the `continue` statement?

```
%%mytutor -h 350
while True:
    message = input('Input something please:')
    ### BEGIN SOLUTION
    if message:
        print('You entered:', message)
    ### END SOLUTION
```

```
<IPython.lib.display.IFrame at 0x7fed6861c898>
```

5.4.3 Else construct for a loop

The following program

- checks whether the user input is a positive integer using `isdigit`, and if so,
- check if the positive integer is a composite number, i.e., a product of two smaller positive integers.

```
@interact(num='1')
def check_composite(num):
    if num.isdigit():
        num = int(num)
        for divisor in range(2,num):
            if num % divisor:
                continue
            else:
                print('It is composite.')
                break
        else:
            print('It is not composite.')
    else:
        print('Not a positive integer.')
```

```
interactive(children=(Text(value='1', description='num'), Output()), _dom_classes=(
↪'widget-interact',))
```

```
%%mytutor -h 500
def check_composite(num):
    if num.isdigit():
        num = int(num)
        for divisor in range(2,num):
            if num % divisor:
                continue
            else:
                print('It is composite.')
                break
        else:
            print('It is not composite.')
    else:
        print('Not a positive integer.')

check_composite('1')
check_composite('2')
check_composite('3')
check_composite('4')
```

```
<IPython.lib.display.IFrame at 0x7fed685af160>
```

In addition to using `continue` and `break` in an elegant way,the code also uses an else clause that is executed only when the loop terminates *normally* not by `break`.

Exercise There are three else claues in the earlier code. Which one is for the loop?

- The second else clause that `print('It is not composite.')`.
- The clause is called when there is no divisor found in the range from `2` to `num`.

Exercise Convert the for loop to a while loop.Can you improve the code to use fewer iterations?

```
@interact(num='1')
def check_composite(num):
    if num.isdigit():
        num = int(num)
        # for divisor in range(2,num):    # use while instead
        divisor = 2
        while divisor <= num**0.5:
            if num % divisor:
                divisor += 1
            else:
                print('It is composite.')
                break
        else:
            print('It is not composite.')
    else:
        print('Not a positive integer.')
```

```
interactive(children=(Text(value='1', description='num'), Output()), _dom_classes=(
↪'widget-interact',))
```

CHAPTER

SIX

USING FUNCTIONS

6.1 Motivation

How to reuse code so we can write less?

When we write a loop, the code is executed multiple times, once for each iteration.

This is a simple form of *code reuse* that

- gives your code an elegant *structure* that
- can be executed efficiently by a computer, and
- *interpreted* easily by a programmer.

How to repeat execution at different times, in different programs, and in slightly different ways?

6.2 Functions

How to calculate the logarithm?

There is no arithmetic operator for logarithm.Do we have to implement it ourselves?

We can use the function `log` from the `math` *module*:

```
from math import log
log(256, 2)  # log base 2 of 256
```

```
8.0
```

The above computes the base-2 logarithm, $\log_2(256)$. Like functions in mathematics, a computer function `log`

- is *called/invoked* with some input *arguments* `(256, 2)` following the function, and
- *returns* an output value computed from the input arguments.

```
# A function is callable while an integer is not
callable(log), callable(1)
```

```
(True, False)
```

Unlike mathematical functions:

- A computer function may require no arguments, but we still need to call it with `()`.

```
input()
```

- A computer function may have side effects and return `None`.

```
x = print()
print(x, 'of type', type(x))
```

```
None of type <class 'NoneType'>
```

An argument of a function call can be any expression.

```
print('1st input:', input(), '2nd input', input())
```

Note also that

- the argument can also be a function call like function composition in mathematics.
- Before a function call is executed, its arguments are evaluated first from left to right.

Why not implement logarithm yourself?

- The function from standard library is efficiently implemented and thoroughly tested/documented.
- Knowing what a function does is often insufficient for an efficient implementation.(See how to calculate logarithm as an example.)

Indeed, the `math` library does not implement `log` itself:

> **CPython implementation detail:** The `math` module consists mostly of thin *wrappers* around the platform C math library functions. - pydoc last paragraph

(See the source code wrapper for `log`.)

Exercise What is a function in programming?

- A function is a structure that allows a piece of code to be reused in a program.
- A function can adapt its computations to different situations using input arguments.

6.3 Import Functions from Modules

How to import functions?

We can use the `import` statement to import multiple functions into the program *global frame*.

```
%%mytutor -h 300
from math import log10, ceil
x = 1234
print('Number of digits of x:', ceil(log10(x)))
```

```
<IPython.lib.display.IFrame at 0x7f15c059a898>
```

The above import both the functions `log10` and `ceil` from `math` to compute the number $\lceil \log_{10}(x) \rceil$ of digits of a *strictly positive* integer x.

How to import all functions from a library?

```
%%mytutor -h 300
from math import *  # import all except names starting with an underscore
print('{:.2f}, {:.2f}, {:.2f}'.format(sin(pi / 6), cos(pi / 3), tan(pi / 4)))
```

```
<IPython.lib.display.IFrame at 0x7f15c059ad30>
```

The above uses the wildcard `*` to import (nearly) all the functions/variables provided in `math`.

What if different packages define the same function?

```
%%mytutor -h 300
print('{}'.format(pow(-1, 2)))
print('{:.2f}'.format(pow(-1, 1 / 2)))
from math import *
print('{}'.format(pow(-1, 2)))
print('{:.2f}'.format(pow(-1, 1 / 2)))
```

```
<IPython.lib.display.IFrame at 0x7f15c05a80f0>
```

- The function `pow` imported from `math` overwrites the built-in function `pow`.
- Unlike the built-in function, `pow` from `math` returns only floats but not integers nor complex numbers.
- We say that the import statement *polluted the namespace of the global frame* and caused a *name collision*.

How to avoid name collisions?

```
%%mytutor -h 250
import math
print('{:.2f}, {:.2f}'.format(math.pow(-1, 2), pow(-1, 1 / 2)))
```

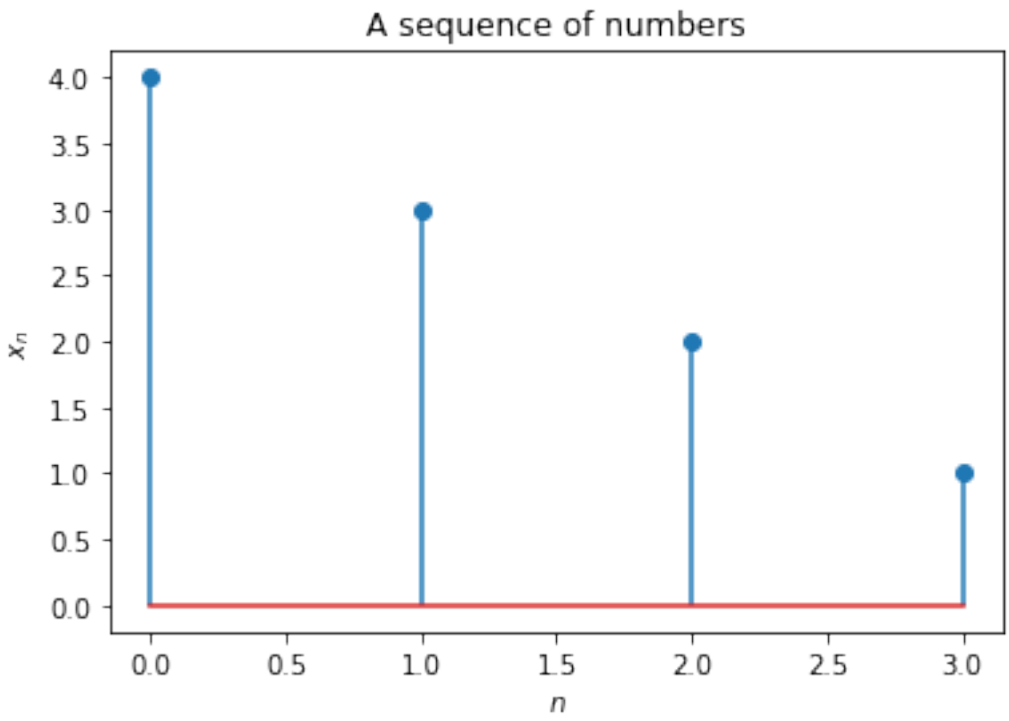
```
<IPython.lib.display.IFrame at 0x7f15c05a8400>
```

We can use the full name (*fully-qualified name*) `math.pow` prefixed with the module name (and possibly package names containing the module).

Can we shorten a name?

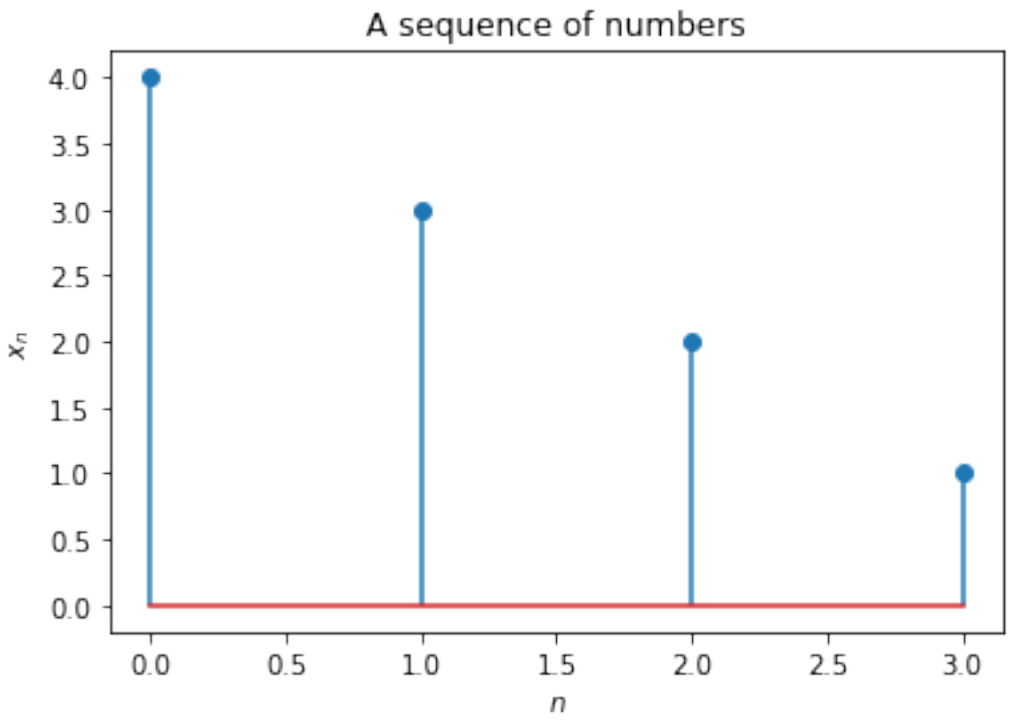
The name of a library can be very long and there can be a hierarchical structure as well.E.g., to plot a sequence using `pyplot` module from `matplotlib` package:

```
%matplotlib inline
import matplotlib.pyplot
matplotlib.pyplot.stem([4, 3, 2, 1])
matplotlib.pyplot.ylabel(r'$x_n$')
matplotlib.pyplot.xlabel(r'$n$')
matplotlib.pyplot.title('A sequence of numbers')
matplotlib.pyplot.show()
```



It is common to rename `matplotlib.pyplot` as `plt`:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.stem([4, 3, 2, 1])
plt.ylabel(r'$x_n$')
plt.xlabel(r'$n$')
plt.title('A sequence of numbers')
plt.show()
```



We can also rename a function as we import it to avoid name collision:

```
from math import pow as fpow
fpow(2, 2), pow(2, 2)
```

```
(4.0, 4)
```

Exercise What is wrong with the following code?

```
import math as m
for m in range(5): m.pow(m, 2)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-14-53146a1609f3> in <module>
      1 import math as m
----> 2 for m in range(5): m.pow(m, 2)

AttributeError: 'int' object has no attribute 'pow'
```

There is a name collision: `m` is assigned to an integer in the for loop and so it is no longer the module `math` when calling `m.pow`.

Exercise Use the `randint` function from `random` to simulate the rolling of a die, by printing a random integer from 1 to 6.

```
import random
print(random.randint(1, 6))
```

```
1
```

6.4 Built-in Functions

How to learn more about a function such as `randint`?

There is a built-in function `help` for showing the *docstring* (documentation string).

```
import random
help(random.randint)  # random must be imported before
```

```
Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

```
help(random)  # can also show the docstring of a module
```

```
help(help)
```

Does built-in functions belong to a module?

Indeed, every function must come from a module.

```
__builtin__.print('I am from the __builtin__ module.')
```

```
I am from the __builtin__ module.
```

`__builtin__` module is automatically loaded because it provides functions that are commonly use for all programs.

How to list everything in a module?

We can use the built-in function `dir` (*directory*).

```
dir(__builtin__)
```

We can also call `dir` without arguments.What does it print?

```
dir()
```

CHAPTER

SEVEN

WRITING FUNCTION

7.1 Function Definition

How to write a function?

A function is defined using the `def` keyword:

The following is a simple function that prints “Hello, World!”.

```
# Function definition
def say_hello():
    print('Hello, World!')
```

```
# Function invocation
say_hello()
```

```
Hello, World!
```

To make a function more powerful and solve different problems,we can

- use a return statement to return a value that
- depends on some input arguments.

```
def increment(x):
    return x + 1


increment(3)
```

```
4
```

We can also have multiple input arguments.

```
def length_of_hypotenuse(a, b):
    if a >= 0 and b >= 0:
        return (a**2 + b**2)**0.5
    else:
        print('Input arguments must be non-negative.')
```

```
length_of_hypotenuse(3, 4)
```

```
5.0
```

```
length_of_hypotenuse(-3, 4)
```

```
Input arguments must be non-negative.
```

7.2 Documentation

How to document a function?

```
# Author: John Doe
# Last modified: 2020-09-14
def increment(x):
    '''The function takes in a value x and returns the increment x + 1.

    It is a simple example that demonstrates the idea of
    - parameter passing,
    - return statement, and
    - function documentation.'''
    return x + 1  # + operation is used and may fail for 'str'
```

The `help` command shows the docstring we write

- at beginning of the function body
- delimited using triple single/double quotes.

```
help(increment)
```

```
Help on function increment in module __main__:

increment(x)
    The function takes in a value x and returns the increment x + 1.

    It is a simple example that demonstrates the idea of
    - parameter passing,
    - return statement, and
    - function documentation.
```

The docstring should contain the *usage guide*, i.e., information for new users to call the function properly.There is a Python style guide (PEP 257) for

- one-line docstrings and
- multi-line docstrings.

Why doesn't `help` show the comments that start with `#`?

```
# Author: John Doe
# Last modified: 2020-09-14
def increment(x):
    ...
    return x + 1  # + operation is used and may fail for 'str'
```

Those comments are not usage guide. They are intended for programmers who need to maintain/extend the function definition.

- Information about the author and modification date facilitate communications among programmers.

- Comments within the code help explain important and not-so-obvious implementation details.

How to let user know the data types of input arguments and return value?

We can annotate the function with *hints* of the types of the arguments and return value.

```
# Author: John Doe
# Last modified: 2020-09-14
def increment(x: float) -> float:
    '''The function takes in a value x and returns the increment x + 1.

    It is a simple example that demonstrates the idea of
    - parameter passing,
    - return statement, and
    - function documentation.'''
    return x + 1  # + operation is used and may fail for 'str'


help(increment)
```

```
Help on function increment in module __main__:

increment(x: float) -> float
    The function takes in a value x and returns the increment x + 1.

    It is a simple example that demonstrates the idea of
    - parameter passing,
    - return statement, and
    - function documentation.
```

The above annotations is not enforced by the Python interpreter.Nevertheless, such annotations make the code easier to understand and can be used by editor with type-checking tools.

```
def increment_user_input():
    return increment(input())  # does not raise error even though input returns str
```

```
increment_user_input()  # still lead to runtime error
```

7.3 Parameter Passing

Can we increment a variable instead of returning its increment?

```
def increment(x):
    x += 1
```

```
x = 3
increment(x)
print(x)  # 4?
```

```
3
```

Does the above code increment `x`?

```
%%mytutor -h 350
def increment(x):
    x += 1


x = 3
increment(x)
print(x)
```

```
<IPython.lib.display.IFrame at 0x7f2f4152d978>
```

- Step 3: The function `increment` is invoked with the argument evaluated to the value of `x`.
- Step 3-4: A local frame is created for variables local to `increment` during its execution.
  - The *formal parameter* `x` in `def increment(x):` becomes a local variable and
  - it is assigned the value `3` of the *actual parameter* given by the global variable `x`.
- Step 5-6: The local (but not the global) variable `x` is incremented.
- Step 6-7: The function call completes and the local frame is removed.

CHAPTER

EIGHT

OBJECTS

8.1 Object-Oriented Programming

Why object-oriented programming?

```
import jupyter_manim
from manimlib.imports import *
```

```
%%manim HelloWorld -l
class HelloWorld(Scene):
    def construct(self):
        self.play(Write(TextMobject('Hello, World!')))
```

```
<IPython.core.display.HTML object>
```

- `HelloWorld` is a specific `Scene` that is
- `constructed` by `playing` an animation that `Write`
- the `TextMobject` of the message `'Hello, World!'`.

Exercise Try changing

- Mobjects: `TextMobject('Hello, World!')` to `TexMobject(r'E=mc^2')` or `Circle()` or `Square()`.
- Animation objects: `Write` to `FadeIn` or `GrowFromCenter`.

See the documentation for other choices.

More complicated behavior can be achieved by using different objects.

```
%%html
<iframe width="912" height="513" src="https://www.youtube.com/embed/ENMyFGmq5OA"␣
↪frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media;␣
↪gyroscope; picture-in-picture" allowfullscreen></iframe>
```

```
<IPython.core.display.HTML object>
```

What is an object?

Almost everything is an `object` in Python.

```
isinstance?
isinstance(1, object), isinstance(1.0, object), isinstance('1', object)
```

```
(True, True, True)
```

A function is also a first-class object object.

```
isinstance(print, object), isinstance(''.isdigit, object)
```

```
(True, True)
```

A data type is also an object.

```
# chicken and egg relationship
isinstance(type, object), isinstance(object, type), isinstance(object, object)
```

```
(True, True, True)
```

Python is a *class-based* object-oriented programming language:

- Each object is an instance of a *class* (also called type in Python).
- An object is a collection of *members/attributes*, each of which is an object.

```
hasattr?
hasattr(str, 'isdigit')
```

```
True
```

Different objects of a class

- have the same set of attributes as that of the class, but
- the attribute values can be different.

```
dir?
dir(1)==dir(int), complex(1, 2).imag != complex(1, 1).imag
```

```
(True, True)
```

How to operate on an object?

- A class can define a function as an attribute for all its instances.
- Such a function is called a *method* or *member function*.

```
complex.conjugate(complex(1, 2)), type(complex.conjugate)
```

```
((1-2j), method_descriptor)
```

A method can be accessed by objects of the class:

```
complex(1, 2).conjugate(), type(complex(1, 2).conjugate)
```

```
((1-2j), builtin_function_or_method)
```

`complex(1,2).conjugate` is a *callable* object:

- Its attribute `__self__` is assigned to `complex(1,2)`.
- When called, it passes `__self__` as the first argument to `complex.conjugate`.

```
callable(complex(1,2).conjugate), complex(1,2).conjugate.__self__
```

```
(True, (1+2j))
```

8.2 File Objects

How to read a text file?

Consider reading a csv (comma separated value) file:

```
!more 'contact.csv'
```

```
name, email, phone
Amelia Hawkins,dugorre@lufu.cg,(414) 524-6465
Alta Perez,bos@fiur.sc,(385) 247-9001
Tai Ming Chan,tmchan@cityu.edu.hk,(634) 234-7294
Annie Zimmerman,okodag@saswuf.mn,(259) 862-1082
Eula Crawford,ve@rorohte.mx,(635) 827-9819
Clayton Atkins,vape@nig.eh,(762) 271-7090
Hallie Day,kozzazazi@ozakewje.am,(872) 949-5878
Lida Matthews,joobu@pabnesis.kg,(213) 486-8330
Amelia Pittman,nulif@uposzag.au,(800) 303-3234
```

To read the file by a Python program:

```
f = open('contact.csv')  # create a file object for reading
print(f.read())    # return the entire content
f.close()          # close the file
```

```
name, email, phone
Amelia Hawkins,dugorre@lufu.cg,(414) 524-6465
Alta Perez,bos@fiur.sc,(385) 247-9001
Tai Ming Chan,tmchan@cityu.edu.hk,(634) 234-7294
Annie Zimmerman,okodag@saswuf.mn,(259) 862-1082
Eula Crawford,ve@rorohte.mx,(635) 827-9819
Clayton Atkins,vape@nig.eh,(762) 271-7090
Hallie Day,kozzazazi@ozakewje.am,(872) 949-5878
Lida Matthews,joobu@pabnesis.kg,(213) 486-8330
Amelia Pittman,nulif@uposzag.au,(800) 303-3234
```

1. `open` is a function that creates a file object and assigns it to `f`.
2. Associated with the file object,

- `read` returns the entire content of the file as a string.
- `close` flushes and closes the file.

Why close a file?

If not, depending on the operating system,

- other programs may not be able to access the file, and
- changes may not be written to the file.

To ensure a file is closed properly, we can use the `with` statement:

```
with open('contact.csv') as f:
    print(f.read())
```

```
name, email, phone
Amelia Hawkins,dugorre@lufu.cg,(414) 524-6465
Alta Perez,bos@fiur.sc,(385) 247-9001
Tai Ming Chan,tmchan@cityu.edu.hk,(634) 234-7294
Annie Zimmerman,okodag@saswuf.mn,(259) 862-1082
Eula Crawford,ve@rorohte.mx,(635) 827-9819
Clayton Atkins,vape@nig.eh,(762) 271-7090
Hallie Day,kozzazazi@ozakewje.am,(872) 949-5878
Lida Matthews,joobu@pabnesis.kg,(213) 486-8330
Amelia Pittman,nulif@uposzag.au,(800) 303-3234
```

The `with` statement applies to any context manager that provides the methods

- `__enter__` for initialization, and
- `__exit__` for finalization.

```
with open('contact.csv') as f:
    print(f, hasattr(f, '__enter__'), hasattr(f, '__exit__'), sep='\n')
```

```
<_io.TextIOWrapper name='contact.csv' mode='r' encoding='UTF-8'>
True
True
```

- `f.__enter__` is called after the file object is successfully created and assigned to `f`, and
- `f.__exit__` is called at the end, which closes the file.
- `f.closed` indicates whether the file is closed.

```
f.closed
```

```
True
```

We can iterate a file object in a for loop,which implicitly call the method `__iter__` to read a file line by line.

```
with open('contact.csv') as f:
    for line in f:
        print(line, end='')

hasattr(f, '__iter__')
```

```
name, email, phone
Amelia Hawkins,dugorre@lufu.cg,(414) 524-6465
Alta Perez,bos@fiur.sc,(385) 247-9001
Tai Ming Chan,tmchan@cityu.edu.hk,(634) 234-7294
Annie Zimmerman,okodag@saswuf.mn,(259) 862-1082
Eula Crawford,ve@rorohte.mx,(635) 827-9819
Clayton Atkins,vape@nig.eh,(762) 271-7090
Hallie Day,kozzazazi@ozakewje.am,(872) 949-5878
Lida Matthews,joobu@pabnesis.kg,(213) 486-8330
Amelia Pittman,nulif@uposzag.au,(800) 303-3234
```

```
True
```

Exercise Print only the first 5 lines of the file `contact.csv`.

```
with open('contact.csv') as f:
    ### BEGIN SOLUTION
    for i, line in enumerate(f):
        print(line, end='')
        if i >= 5: break
    ### END SOLUTION
```

```
name, email, phone
Amelia Hawkins,dugorre@lufu.cg,(414) 524-6465
Alta Perez,bos@fiur.sc,(385) 247-9001
Tai Ming Chan,tmchan@cityu.edu.hk,(634) 234-7294
Annie Zimmerman,okodag@saswuf.mn,(259) 862-1082
Eula Crawford,ve@rorohte.mx,(635) 827-9819
```

How to write to a text file?

Consider backing up `contact.csv` to a new file:

```
destination = 'private/new_contact.csv'
```

The directory has to be created first if it does not exist:

```
import os
os.makedirs(os.path.dirname(destination), exist_ok=True)
```

```
os.makedirs?
!ls
```

```
Objects.ipynb  contact.csv  media  private
```

To write to the destination file:

```
with open('contact.csv') as source_file:
    with open(destination, 'w') as destination_file:
        destination_file.write(source_file.read())
```

```
destination_file.write?
!more {destination}
```

```
name, email, phone
Amelia Hawkins,dugorre@lufu.cg,(414) 524-6465
Alta Perez,bos@fiur.sc,(385) 247-9001
Tai Ming Chan,tmchan@cityu.edu.hk,(634) 234-7294
Annie Zimmerman,okodag@saswuf.mn,(259) 862-1082
Eula Crawford,ve@rorohte.mx,(635) 827-9819
Clayton Atkins,vape@nig.eh,(762) 271-7090
Hallie Day,kozzazazi@ozakewje.am,(872) 949-5878
Lida Matthews,joobu@pabnesis.kg,(213) 486-8330
Amelia Pittman,nulif@uposzag.au,(800) 303-3234
```

- The argument `'w'` to `open` sets the file object to write mode.
- The method `write` writes the input strings to the file.

Exercise We can also use `a` mode to *append* new content to a file.Complete the following code to append `new_data` to the file `destination`.

```
new_data = 'Effie, Douglas,galnec@naowdu.tc, (888) 311-9512'
with open(destination, 'a') as f:
    ### BEGIN SOLUTION
    f.write('\n')
    f.write(new_data)
    ### END SOLUTION
!more {destination}
```

```
name, email, phone
Amelia Hawkins,dugorre@lufu.cg,(414) 524-6465
Alta Perez,bos@fiur.sc,(385) 247-9001
Tai Ming Chan,tmchan@cityu.edu.hk,(634) 234-7294
Annie Zimmerman,okodag@saswuf.mn,(259) 862-1082
Eula Crawford,ve@rorohte.mx,(635) 827-9819
Clayton Atkins,vape@nig.eh,(762) 271-7090
Hallie Day,kozzazazi@ozakewje.am,(872) 949-5878
Lida Matthews,joobu@pabnesis.kg,(213) 486-8330
Amelia Pittman,nulif@uposzag.au,(800) 303-3234
Effie, Douglas,galnec@naowdu.tc, (888) 311-9512
```

How to delete a file?

Note that the file object does not provide any method to delete the file.Instead, we should use the function `remove` of the `os` module.

```
if os.path.exists(destination):
    os.remove(destination)
```

8.3 String Objects

How to search for a substring in a string?

A string object has the method `find` to search for a substring.E.g., to find the contact information of Tai Ming:

```
str.find?
with open('contact.csv') as f:
    for line in f:
        if line.find('Tai Ming') != -1:
            record = line
            print(record)
            break
```

```
Tai Ming Chan,tmchan@cityu.edu.hk,(634) 234-7294
```

How to split and join strings?

A string can be split according to a delimiter using the `split` method.

```
record.split(',')
```

```
['Tai Ming Chan', 'tmchan@cityu.edu.hk', '(634) 234-7294\n']
```

The list of substrings can be joined back together using the `join` methods.

```
print('\n'.join(record.split(',')))
```

```
Tai Ming Chan
tmchan@cityu.edu.hk
(634) 234-7294
```

Exercise Print only the phone number (last item) in `record`. Use the method `rstrip` or `strip` to remove unnecessary white spaces at the end.

```
str.rstrip?
### BEGIN SOLUTION
print(record.split(',')[-1].rstrip())
### END SOLUTION
```

```
(634) 234-7294
```

Exercise Print only the name (first item) in `record` but with

- surname printed first with all letters in upper case
- followed by a comma, a space, and
- the first name as it is in `record`.

E.g., `Tai Ming Chan` should be printed as `CHAN, Tai Ming`.

Hint: Use the methods `upper` and `rsplit` (with the parameter `maxsplit=1`).

```
str.rsplit?
### BEGIN SOLUTION
first, last = record.split(',')[0].rsplit(' ', maxsplit=1)
print('{}, {}'.format(last.upper(),first))
### END SOLUTION
```

```
CHAN, Tai Ming
```

8.4 Operator Overloading

8.4.1 What is overloading?

Recall that the addition operation + behaves differently for different types.

```
for x, y in (1, 1), ('1', '1'), (1, '1'):
    print(f'{x!r:^5} + {y!r:^5} = {x+y!r}')
```

```
  1   +   1   = 2
 '1'  +  '1'  = '11'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-32-297be71279db> in <module>
      1 for x, y in (1, 1), ('1', '1'), (1, '1'):
----> 2     print(f'{x!r:^5} + {y!r:^5} = {x+y!r}')
```

(continues on next page)

(continued from previous page)

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- Having an operator perform differently based on its argument types is called operator *overloading*.
- + is called a *generic* operator.
- We can also have function overloading to create generic functions.

8.4.2 How to dispatch on type?

The strategy of checking the type for the appropriate implementation is called *dispatching on type*.

A naive idea is to put all different implementations together with case-by-case checks of operand types.

```
def add_case_by_case(x, y):
    if isinstance(x, int) and isinstance(y, int):
        print('Do integer summation...')
    elif isinstance(x, str) and isinstance(y, str):
        print('Do string concatenation...')
    else:
        print('Return a TypeError...')
    return x + y  # replaced by internal implementations


for x, y in (1, 1), ('1', '1'), (1, '1'):
    print(f'{x!r:^10} + {y!r:^10} = {add_case_by_case(x,y)!r}')
```

```
Do integer summation...
    1      +     1      = 2
Do string concatenation...
   '1'     +    '1'     = '11'
Return a TypeError...
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-33-85c671bf17e3> in <module>
     10
     11 for x, y in (1, 1), ('1', '1'), (1, '1'):
---> 12     print(f'{x!r:^10} + {y!r:^10} = {add_case_by_case(x,y)!r}')

<ipython-input-33-85c671bf17e3> in add_case_by_case(x, y)
      6     else:
      7         print('Return a TypeError...')
----> 8     return x + y  # replaced by internal implementations
      9
     10

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

It can get quite messy with all possible types and combinations.

```
for x, y in ((1, 1.1), (1, complex(1, 2)), ((1, 2), (1, 2))):
    print(f'{x!r:^10} + {y!r:^10} = {x+y!r}')
```

```
   1      +    1.1     = 2.1
   1      +   (1+2j)   = (2+2j)
(1, 2)    +   (1, 2)   = (1, 2, 1, 2)
```

What about new data types?

```
from fractions import Fraction  # non-built-in type for fractions
for x, y in ((Fraction(1, 2), 1), (1, Fraction(1, 2))):
    print(f'{x} + {y} = {x+y}')
```

```
1/2 + 1 = 3/2
1 + 1/2 = 3/2
```

Weaknesses of the naive approach:

1. New data types require rewriting the addition operation.
2. A programmer may not know all other types and combinations to rewrite the code properly.

8.4.3 How to have data-directed programming?

The idea is to treat an implementation as a datum that can be returned by the operand types.

- `x + y` is a *syntactic sugar* that
- invokes the method `type(x).__add__(x,y)` of `type(x)` to do the addition.

```
for x, y in (Fraction(1, 2), 1), (1, Fraction(1, 2)):
    print(f'{x} + {y} = {type(x).__add__(x,y)}')  # instead of x + y
```

```
1/2 + 1 = 3/2
1 + 1/2 = NotImplemented
```

- The first case calls `Fraction.__add__`, which provides a way to add `int` to `Fraction`.
- The second case calls `int.__add__`, which cannot provide any way of adding `Fraction` to `int`. (Why not?)

Why return a `NotImplemented` object instead of raising an error/exception?

- This allows + to continue to handle the addition by
- dispatching on `Fraction` to call its reverse addition method `__radd__`.

```
%%mytutor -h 500
from fractions import Fraction
def add(x, y):
    '''Simulate the + operator.'''
    sum = x.__add__(y)
    if sum is NotImplemented:
        sum = y.__radd__(x)
    return sum


for x, y in (Fraction(1, 2), 1), (1, Fraction(1, 2)):
    print(f'{x} + {y} = {add(x,y)}')
```

```
<IPython.lib.display.IFrame at 0x7f5bc28de8d0>
```

The object-oriented programming techniques involved are formally called:

- *Polymorphism*: Different types can have different implementations of the `__add__` method.
- *Single dispatch*: The implementation is chosen based on one single type at a time.

Remarks:

- A method with starting and trailing double underscores in its name is called a *dunder method*.
- Dunder methods are not intended to be called directly. E.g., we normally use + instead of `__add__`.
- Other operators have their corresponding dunder methods that overloads the operator.

8.5 Object Aliasing

When are two objects identical?

- Two objects are the same if they occupy the same memory.
- The keyword `is` checks whether two objects are the same object.
- The function `id` returns a unique id number for each object.

```
%%mytutor -h 400
x, y = complex(1,2), complex(1,2)
z = x

for expr in 'id(x)', 'id(y)', 'id(z)', 'x == y == z', 'x is y', 'x is z':
    print(expr,eval(expr))
```

```
<IPython.lib.display.IFrame at 0x7f5bc19150b8>
```

As the box-pointer diagram shows:

- `x` is not `y` because they point to objects at different memory locations,even though the objects have the same type and value.
- `x` is `z` because the assignment `z = x` binds `z` to the same memory location `x` points to.`z` is said to be an *alias* (another name) of `x`.

Should we use `is` or `==`?

`is` is faster but:

```
1 is 1, 1 is 1., 1 == 1.
```

```
(True, False, True)
```

- `1 is 1.` returns false because `1` is `int` but `1.` is `float`.
- `==` calls the method `__eq__` of `float` which returns mathematical equivalence.

Can we use `is` for integer comparison?

```
x, y = 1234, 1234
1234 is 1234, x is y
```

```
(True, False)
```

No. The behavior of `is` is not entirely predictable.

When should we use `is`?

`is` can be used for built-in constants such as `None` and `NotImplemented`because there can only be one instance of each of them.

CHAPTER

NINE

MORE ON FUNCTIONS

9.1 Recursion

Consider computing the Fibonacci number of order n:

$$F_n := \begin{cases} F_{n-1} + F_{n-2} & n > 1 \quad \text{(recurrence)} \\ 1 & n = 1 \quad \text{(base case)} \\ 0 & n = 0 \quad \text{(base case).} \end{cases}$$

Fibonacci numbers have practical applications in generating pseudorandom numbers.

Can we define the function by calling the function itself?

```
%%mytutor -r -h 450
def fibonacci(n):
    if n > 1:
        return fibonacci(n - 1) + fibonacci(n - 2)  # recursion
    elif n == 1:
        return 1
    else:
        return 0

fibonacci(2)
```

```
1
```

```
<IPython.lib.display.IFrame at 0x7fd7e27fc898>
```

Recursion is a function that calls itself (*recurs*).

Exercise Write a function `gcd` that implements the Euclidean algorithm for the greatest common divisor:

$$\gcd(a, b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & \text{otherwise} \end{cases}$$

```
%%mytutor -r -h 550
def gcd(a, b):
    ### BEGIN SOLUTION
    return gcd(b, a % b) if b else a
    ### END SOLUTION


gcd(3 * 5, 5 * 7)
```

```
5
```

```
<IPython.lib.display.IFrame at 0x7fd7e27fca20>
```

Is recursion strictly necessary?

No. We can always convert a recursion to an iteration.E.g., the following computes the Fibonnacci number of order using a while loop instead.

```
%%mytutor -r -h 550
def fibonacci_iteration(n):
    if n > 1:
        _, F = 0, 1  # next two Fibonacci numbers
        while n > 1:
            _, F, n = F, F + _, n - 1
        return F
    elif n == 1:
        return 1
    else:
        return 0

fibonacci_iteration(3)
```

```
2
```

```
<IPython.lib.display.IFrame at 0x7fd7e807d5f8>
```

```
# more tests
for n in range(5):
    assert fibonacci(n) == fibonacci_iteration(n)
```

Exercise Implement `gcd_iteration` using a while loop instead of a recursion.

```
%%mytutor -r -h 550
def gcd_iteration(a, b):
    ### BEGIN SOLUTION
    while b:
        a, b = b, a % b
    return a
    ### END SOLUTION


gcd_iteration(3 * 5, 5 * 7)
```

```
5
```

```
<IPython.lib.display.IFrame at 0x7fd7e27fc2b0>
```

```
# test
for n in range(5):
    assert fibonacci(n) == fibonacci_iteration(n)
```

What is the benefit of recursion?

- Recursion is often shorter and easier to understand.

- It is also easier to write code by *wishful thinking* or *declarative programming*.

Is recusion more efficient than iteration?

Exercise Find the smallest values of `n` for`fibonacci(n)` and `fibonacci_iteration(n)` respectively to run for more than a second.

```
# Assign n
### BEGIN SOLUTION
n = 33
### END SOLUTION
fib_recursion = fibonacci(n)
```

```
# Assign n
### BEGIN SOLUTION
n = 300000
### END SOLUTION
fib_iteration = fibonacci_iteration(n)
```

To see why recursion is slow, we will modify `fibonacci` to print each function call as follows.

```
def fibonacci(n):
    '''Returns the Fibonacci number of order n.'''
    print('fibonacci({!r})'.format(n))
    return fibonacci(n - 1) + fibonacci(n - 2) if n > 1 else 1 if n == 1 else 0


fibonacci(5)
```

```
fibonacci(5)
fibonacci(4)
fibonacci(3)
fibonacci(2)
fibonacci(1)
fibonacci(0)
fibonacci(1)
fibonacci(2)
fibonacci(1)
fibonacci(0)
fibonacci(3)
fibonacci(2)
fibonacci(1)
fibonacci(0)
fibonacci(1)
```

```
5
```

`fibonacci(5)` calls `fibonacci(3)` and `fibonacci(4)`, which in turn call `fibonacci(2)` and `fibonacci(3)`. `fibonacci(3)` is called twice.

9.2 Global Variables

Consider the problem of generating a sequence of Fibonacci numbers.

```
for n in range(5):
    print(fibonacci_iteration(n))
```

```
0
1
1
2
3
```

Is the above loop efficient?

No. Each call to `fibonacci_iteration(n)` recomputes the last two Fibonacci numbers F_{n-1} and F_{n-2} for $n \geq 2$.

How to avoid redundant computations?

One way is to store the last two computed Fibonacci numbers.

```
%%mytutor -h 600
def next_fibonacci():
    '''Returns the next Fibonacci number.'''
    global _Fn, _Fn1, _n  # global declaration
    value = _Fn
    _Fn, _Fn1, _n = _Fn1, _Fn + _Fn1, _n + 1
    return value

def print_fibonacci_state():
    print('''States:
    _Fn  : Next Fibonacci number      = {}
    _Fn1 : Next next Fibonacci number = {}
    _n   : Next order                 = {}'''.format(_Fn,_Fn1,_n))

# global variables for next_fibonacci and print_fibonacci_state
_Fn, _Fn1, _n = 0, 1, 0

for n in range(5):
    print(next_fibonacci())
print_fibonacci_state()
```

```
<IPython.lib.display.IFrame at 0x7fd7e1f9f240>
```

Rules for *global/local variables*:

1. A local variable must be defined within a function.
2. An assignment defines a local variable except in a `global` statement.

Why `global` is NOT needed in `print_fibonacci_state`?

Without ambiguity, `_Fn, _Fn1, _n` in `print_fibonacci_state` are not local variables by Rule 1 because they are not defined within the function.

Why `global` is needed in `next_fibonacci`?

What happens otherwise:

```
def next_fibonacci():
    '''Returns the next Fibonacci number.'''
    # global _Fn, _Fn1, _n
    value = _Fn
    _Fn, _Fn1, _n = _Fn1, _Fn + _Fn1, _n + 1
    return value

next_fibonacci()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-13-44c8c496bfd8> in <module>
      6     return value
      7
----> 8 next_fibonacci()

<ipython-input-13-44c8c496bfd8> in next_fibonacci()
      2     '''Returns the next Fibonacci number.'''
      3     # global _Fn, _Fn1, _n
----> 4     value = _Fn
      5     _Fn, _Fn1, _n = _Fn1, _Fn + _Fn1, _n + 1
      6     return value

UnboundLocalError: local variable '_Fn' referenced before assignment
```

Why is there an `UnboundLocalError`?

- The assignment defines `_Fn` as a local variable by Rule 2.
- However, the assignment requires first evaluating `_Fn`, which is not yet defined.

Are global variables preferred over local ones?

Suppose for aesthetic reasons we remove the underscores in global variable names?

```
%%mytutor -h 600
def next_fibonacci():
    '''Returns the next Fibonacci number.'''
    global Fn, Fn1, n
    value = Fn
    Fn, Fn1, n = Fn1, Fn + Fn1, n + 1
    return value

def print_fibonacci_state():
    print('''States:
    Fn  : Next Fibonacci number      = {}
    Fn1 : Next next Fibonacci number = {}
    n   : Next order                 = {}'''.format(Fn,Fn1,n))

# global variables renamed without underscores
Fn, Fn1, n = 0, 1, 0

n = 0
while n < 5:
    print(next_fibonacci())
    n += 1
print_fibonacci_state()
```

```
<IPython.lib.display.IFrame at 0x7fd7e1f9fa20>
```

Exercise Why does the while loop prints only 3 instead of 5 Fibonacci numbers?

There is a name collision. `n` is also incremented by `next_fibonacci()`, and so the while loop is only executed 3 times in total.

With global variables

- codes are less predictable, more difficult to reuse/extend, and
- tests cannot be isolated, making debugging difficult.

Is it possible to store the function states without using global variables?

Yes. We can use nested functions and `nonlocal` variables.

```
def fibonacci_closure(Fn, Fn1):
    def next_fibonacci():
        '''Returns the next (generalized) Fibonacci number starting with
        Fn and Fn1 as the first two numbers.'''
        nonlocal Fn, Fn1, n  # declare nonlocal variables
        value = Fn
        Fn, Fn1, n = Fn1, Fn + Fn1, n + 1
        return value

    def print_fibonacci_state():
        print('''States:
        Next Fibonacci number      = {}
        Next next Fibonacci number = {}
        Next order                 = {}'''.format(Fn, Fn1, n))

    n = 0  # Fn and Fn1 specified in the function arguments
    return next_fibonacci, print_fibonacci_state


next_fibonacci, print_fibonacci_state = fibonacci_closure(0, 1)
n = 0
while n < 5:
    print(next_fibonacci())
    n += 1
print_fibonacci_state()
```

```
0
1
1
2
3
States:
        Next Fibonacci number      = 5
        Next next Fibonacci number = 8
        Next order                 = 5
```

The state variables `Fn, Fn1, n` are now *encapsulated*, and sothe functions returned by `fibonacci_closure` no longer depends on any global variables.

Another benefit of using nested functions is that we can also create different Fibonacci sequence with different base cases.

```
my_next_fibonacci, my_print_fibonacci_state = fibonacci_closure('cs', '1302')
for n in range(5):
    print(my_next_fibonacci())
my_print_fibonacci_state()
```

```
cs
1302
cs1302
1302cs1302
cs13021302cs1302
States:
        Next Fibonacci number      = 1302cs1302cs13021302cs1302
        Next next Fibonacci number = cs13021302cs13021302cs1302cs13021302cs1302
        Next order                 = 5
```

`next_fibonacci` and `print_fibonacci_state` are *local functions* of `fibonacci_closure`.

- They can access (*capture*) the other local variables of `fibonacci_closure` by forming the so-called *closures*.
- Similar to the use of `global` statement, a `non-local` statement is needed for assigning nonlocal variables.

Each local function has an attribute named `__closure__` that stores the captured local variables.

```
def print_closure(f):
    '''Print the closure of a function.'''
    print('closure of ', f.__name__)
    for cell in f.__closure__:
        print('    {} content: {!r}'.format(cell, cell.cell_contents))


print_closure(next_fibonacci)
print_closure(print_fibonacci_state)
```

```
closure of  next_fibonacci
    <cell at 0x7fd7e98a9978: int object at 0x55ee54d3a4a0> content: 5
    <cell at 0x7fd7e98a99d8: int object at 0x55ee54d3a500> content: 8
    <cell at 0x7fd7e98a9e88: int object at 0x55ee54d3a4a0> content: 5
closure of  print_fibonacci_state
    <cell at 0x7fd7e98a9978: int object at 0x55ee54d3a4a0> content: 5
    <cell at 0x7fd7e98a99d8: int object at 0x55ee54d3a500> content: 8
    <cell at 0x7fd7e98a9e88: int object at 0x55ee54d3a4a0> content: 5
```

9.3 Generator

Another way to generate a sequence of objects one-by-one is to write a *generator*.

```
fibonacci_generator = (fibonacci_iteration(n) for n in range(3))
fibonacci_generator
```

```
<generator object <genexpr> at 0x7fd7e1f08de0>
```

The above uses a *generator expression* to define `fibonacci_generator`.

How to obtain items from a generator?

We can use the `next` function.

```
while True:
    print(next(fibonacci_generator)) # raises StopIterationException eventually
```

```
0
1
1
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-19-e6f031ce2829> in <module>
      1 while True:
----> 2     print(next(fibonacci_generator)) # raises StopIterationException␣
 ↪eventually

StopIteration:
```

A generator object is *iterable*, i.e., it implements both `__iter__` and `__next__` methods that are automatically called in a `for` loop as well as the `next` function.

```
fibonacci_generator = (fibonacci_iteration(n) for n in range(5))
for fib in fibonacci_generator:  # StopIterationException handled by for loop
    print(fib)
```

```
0
1
1
2
3
```

Is `fibonacci_generator` efficient?

No again due to redundant computations.A better way to define the generator is to use the keyword `yield`:

```
%%mytutor -h 450
def fibonacci_sequence(Fn, Fn1, stop):
    '''Return a generator that generates Fibonacci numbers
    starting from Fn and Fn1 until stop (exclusive).'''
    while Fn < stop:
        yield Fn  # return Fn and pause execution
        Fn, Fn1 = Fn1, Fn1 + Fn


for fib in fibonacci_sequence(0, 1, 5):
    print(fib)
```

```
<IPython.lib.display.IFrame at 0x7fd7e27fc898>
```

1. `yield` causes the function to return a *generator* without executing the function body.
2. Calling `__next__` resumes the execution, which
    - pauses at the next `yield` expression, or
    - raises the `StopIterationException` at the end.

Exercise The yield expression `yield ...` is mistaken in [Halterman17] to be a statement. It is actually an expression because

- The value of a `yield` expression is `None` by default, but
- it can be set by the `generator.send` method.

Add the document string to the following function. In particular, explain the effect of calling the method `send` on the returned generator.

```
%%mytutor -r -h 500
def fibonacci_sequence(Fn, Fn1, stop):
    ### BEGIN SOLUTION
    '''Return a generator that generates Fibonacci numbers
    starting from Fn and Fn1 to stop (exclusive).
    generator.send(value) sets next number to value.'''
    ### END SOLUTION
    while Fn < stop:
        value = yield Fn
        if value is not None:
            Fn1 = value  # set next number to the value of yield expression
        Fn, Fn1 = Fn1, Fn1 + Fn
```

```
<IPython.lib.display.IFrame at 0x7fd7e807d438>
```

9.4 Optional Arguments

How to make function arguments optional?

```
def fibonacci_sequence(Fn=0, Fn1=1, stop=None):
    while stop is None or Fn < stop:
        value = yield Fn
        Fn, Fn1 = Fn1, Fn1 + Fn
```

```
for fib in fibonacci_sequence(0,1,5):
    print(fib)  # with all arguments specified
```

```
0
1
1
2
3
```

```
for fib in fibonacci_sequence(stop=5):
    print(fib)  # with default Fn=0, Fn1=1
```

```
0
1
1
2
3
```

`stop=5` is called a keyword argument. Unlike `positional arguments`, it specifies the name of the argument explicitly.

Exercise `stop` is an optional argument with the *default value* `None`. What is the behavior of the following code?

```
for fib in fibonacci_sequence(5):
    print(fib)
    if fib > 10:
        break  # Will this be executed?
```

```
5
1
6
7
13
```

With the default value of `None`, the while loop becomes an infinite loop. The generator will keep generating the next Fibonacci number without any bound on the order. In particular, `fibonacci_sequence(5)` creates an unstoppable (default) generator with base case `Fn=5` (specified) and `Fn1=1` (default).

Rules for specifying arguments:

1. Keyword arguments must be after all positional arguments.
2. Duplicate assignments to an argument are not allowed.

E.g., the following results in error:

```
fibonacci_sequence(stop=10, 1)
```

```
  File "<ipython-input-27-c4b4809b18c1>", line 1
    fibonacci_sequence(stop=10, 1)
                               ^
SyntaxError: positional argument follows keyword argument
```

```
fibonacci_sequence(1, Fn=1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-28-2db5e024912c> in <module>
----> 1 fibonacci_sequence(1, Fn=1)

TypeError: fibonacci_sequence() got multiple values for argument 'Fn'
```

The following shows that the behavior of `range` is different.

```
for count in range(1, 10, 2):
    print(count, end=' ')  # counts from 1 to 10 in steps of 2
print()
for count in range(1, 10):
    print(count, end=' ')  # default step=1
print()
for count in range(10):
    print(count, end=' ')  # default start=0, step=1
range(stop=10)  # fails
```

```
1 3 5 7 9
1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-29-f3395132058b> in <module>
      7 for count in range(10):
      8     print(count, end=' ')  # default start=0, step=1
----> 9 range(stop=10)  # fails

TypeError: range() takes no keyword arguments
```

`range` takes only positional arguments.However, the first positional argument has different intepretations (`start` or `stop`) depending on the number of arguments (2 or 1).

`range` is indeed NOT a generator.

```
print(type(range),type(range(10)))
```

```
<class 'type'> <class 'range'>
```

9.5 Variable number of arguments

We can simulate the behavior of range by having a variable number of arguments.

```
def print_arguments(*args, **kwargs):
    '''Take any number of arguments and prints them'''
    print('args ({}): {}'.format(type(args),args))
    print('kwargs ({}): {}'.format(type(kwargs),kwargs))

print_arguments(0, 10, 2, start=1, stop=2)
print("{k}".format(greeting="Hello",k=8),"*"  )
```

```
args (<class 'tuple'>): (0, 10, 2)
kwargs (<class 'dict'>): {'start': 1, 'stop': 2}
8 *
```

- `args` is a tuple of positional arguments.
- `kwargs` is a dictionary of keyword arguments.

`*` and `**` are *unpacking operators* for tuple/list and dictionary respectively:

```
args = (0, 10, 2)
kwargs = {'start': 1, 'stop': 2}
print_arguments(*args, **kwargs)
```

```
args (<class 'tuple'>): (0, 10, 2)
kwargs (<class 'dict'>): {'start': 1, 'stop': 2}
```

The following function converts all the arguments to a string.It will be useful later on.

```
def argument_string(*args, **kwargs):
    '''Return the string representation of the list of arguments.'''
    return '({})'.format(', '.join([
        *['{!r}'.format(v) for v in args],  # arguments
        *['{}={!r}'.format(k, v)
```

(continues on next page)

(continued from previous page)

```
            for k, v in kwargs.items()]  # keyword arguments
    ]))

argument_string(0, 10, 2, start=1, stop=2)
```

```
'(0, 10, 2, start=1, stop=2)'
```

Exercise Redefine `fibonacci_sequence` so that the positional arguments depend on the number of arguments:

```
def fibonacci_sequence(*args):
    '''Return a generator that generates Fibonacci numbers
    starting from Fn and Fn1 to stop (exclusive).
    generator.send(value) sets next number to value.

    fibonacci_sequence(stop)
    fibonacci_sequence(Fn,Fn1)
    fibonacci_sequence(Fn,Fn1,stop)
    '''
    Fn, Fn1, stop = 0, 1, None  # default values

    # handle different number of arguments
    if len(args) is 1:
        ### BEGIN SOLUTION
        stop = args[0]
        ### END SOLUTION
    elif len(args) is 2:
        Fn, Fn1 = args[0], args[1]
    elif len(args) > 2:
        Fn, Fn1, stop = args[0], args[1], args[2]

    while stop is None or Fn < stop:
        value = yield Fn
        if value is not None:
            Fn1 = value  # set next number to the value of yield expression
        Fn, Fn1 = Fn1, Fn1 + Fn
```

```
for fib in fibonacci_sequence(5): # default Fn=0, Fn=1
    print(fib)
```

```
0
1
1
2
3
```

```
for fib in fibonacci_sequence(1, 2): # default stop=None
    print(fib)
    if fib>5:
        break
```

```
1
2
3
5
8
```

```
args = (1, 2, 5)
for fib in fibonacci_sequence(*args): # default stop=None
    print(fib)
```

```
1
2
3
```

9.6 Decorator

What is function decoration?Why decorate a function?

```
def fibonacci(n):
    '''Returns the Fibonacci number of order n.'''
    global count, depth
    count += 1
    depth += 1
    print('{:>3}: {}fibonacci({!r})'.format(count, '|' * depth, n))

    value = fibonacci(n - 1) + fibonacci(n - 2) if n > 1 else 1 if n == 1 else 0

    depth -= 1
    if depth is -1:  # recursion done
        print('Done')
        count = 0  # reset count for subsequent recursions
    return value


count, depth = 0, -1
for n in range(6):
    print(fibonacci(n))
```

```
  1: fibonacci(0)
Done
0
  1: fibonacci(1)
Done
1
  1: fibonacci(2)
  2: |fibonacci(1)
  3: |fibonacci(0)
Done
1
  1: fibonacci(3)
  2: |fibonacci(2)
  3: ||fibonacci(1)
  4: ||fibonacci(0)
  5: |fibonacci(1)
Done
2
  1: fibonacci(4)
  2: |fibonacci(3)
  3: ||fibonacci(2)
  4: |||fibonacci(1)
```

(continues on next page)

(continued from previous page)

```
  5: |||fibonacci(0)
  6: ||fibonacci(1)
  7: |fibonacci(2)
  8: ||fibonacci(1)
  9: ||fibonacci(0)
Done
3
  1: fibonacci(5)
  2: |fibonacci(4)
  3: ||fibonacci(3)
  4: |||fibonacci(2)
  5: ||||fibonacci(1)
  6: ||||fibonacci(0)
  7: |||fibonacci(1)
  8: ||fibonacci(2)
  9: |||fibonacci(1)
 10: |||fibonacci(0)
 11: |fibonacci(3)
 12: ||fibonacci(2)
 13: |||fibonacci(1)
 14: |||fibonacci(0)
 15: ||fibonacci(1)
Done
5
```

The code decorates the `fibonacci` function by printing each recursive call and the depth of the call stack.The decoration is useful in showing the efficiency of the function, but it rewrites the function definition.

How to decorate a function without changing its code?

- What if the decorations are temporary and should be removed later?
- Go through the source codes of all decorated functions to remove the decorations?
- When updating a piece of code, switch back and forth between original and decorated codes?

What about defining a new function that calls and decorates the original function?

```
def fibonacci(n):
    '''Returns the Fibonacci number of order n.'''
    return fibonacci(n - 1) + fibonacci(n - 2) if n > 1 else 1 if n is 1 else 0

def fibonacci_decorated(n):
    '''Returns the Fibonacci number of order n.'''
    global count, depth
    count += 1
    depth += 1
    print('{:>3}: {}fibonacci({!r})'.format(count, '|' * depth, n))

    value = fibonacci(n)

    depth -= 1
    if depth is -1:  # recursion done
        print('Done')
        count = 0  # reset count for subsequent recursions
    return value
```

(continues on next page)

(continued from previous page)

```
count, depth = 0, -1
for n in range(6):
    print(fibonacci_decorated(n))
```

```
  1: fibonacci(0)
Done
0
  1: fibonacci(1)
Done
1
  1: fibonacci(2)
Done
1
  1: fibonacci(3)
Done
2
  1: fibonacci(4)
Done
3
  1: fibonacci(5)
Done
5
```

We want `fibonacci` to call `fibonacci_decorated` instead.What about renaming `fibonacci_decorated` to `fibonacci`?

```
fibonacci = fibonacci_decorated
count, depth = 0, -1
fibonacci_decorated(10)
```

(If you are faint-hearted, don't run the above code.)

We want `fibonacci_decorated` to call the original `fibonacci`.

The solution is to capture the original `fibonacci` in a closure:

```
import functools


def print_function_call(f):
    '''Return a decorator that prints function calls.'''
    @functools.wraps(f)  # give wrapper the identity of f and more
    def wrapper(*args, **kwargs):
        nonlocal count, depth
        count += 1
        depth += 1
        call = '{}{}'.format(f.__name__, argument_string(*args, **kwargs))
        print('{:>3}:{}{}'.format(count, '|' * depth, call))

        value = f(*args, **kwargs)  # wrapper calls f

        depth -= 1
        if depth is -1:
            print('Done')
            count = 0
        return value
```

(continues on next page)

(continued from previous page)

```
    count, depth = 0, -1
    return wrapper  # return the decorated function
```

`print_function_call` takes in `f` and returns `wrapper`, which captures and decorates `f`:

- `wrapper` expects the same set of arguments for `f`,
- returns the same value returned by `f` on the arguments, but
- can execute additional codes before and after calling `f` to print the function call.

By redefining `fibonacci` as the returned `wrapper`, the original `fibonacci` captured by `wrapper` calls `wrapper` as desired.

```
def fibonacci(n):
    return fibonacci(n - 1) + fibonacci(n - 2) if n > 1 else 1 if n is 1 else 0


fibonacci = print_function_call(
    fibonacci)  # so original fibonnacci calls wrapper
fibonacci(5)
```

```
  1:fibonacci(5)
  2:|fibonacci(4)
  3:||fibonacci(3)
  4:|||fibonacci(2)
  5:||||fibonacci(1)
  6:||||fibonacci(0)
  7:|||fibonacci(1)
  8:||fibonacci(2)
  9:|||fibonacci(1)
 10:|||fibonacci(0)
 11:|fibonacci(3)
 12:||fibonacci(2)
 13:|||fibonacci(1)
 14:|||fibonacci(0)
 15:||fibonacci(1)
Done
```

```
5
```

The redefinition does not change the original `fibonacci` captured by `wrapper`.

```
import inspect
for cell in fibonacci.__closure__:
    if callable(cell.cell_contents):
        print(inspect.getsource(cell.cell_contents))
```

```
def fibonacci(n):
    return fibonacci(n - 1) + fibonacci(n - 2) if n > 1 else 1 if n is 1 else 0
```

Python provides the syntatic sugar below to simplify the redefinition.

```
@print_function_call
def fibonacci(n):
    return fibonacci(n - 1) + fibonacci(n - 2) if n > 1 else 1 if n is 1 else 0
```

(continues on next page)

(continued from previous page)

```
fibonacci(5)
```

```
  1:fibonacci(5)
  2:|fibonacci(4)
  3:||fibonacci(3)
  4:|||fibonacci(2)
  5:||||fibonacci(1)
  6:||||fibonacci(0)
  7:|||fibonacci(1)
  8:||fibonacci(2)
  9:|||fibonacci(1)
 10:|||fibonacci(0)
 11:|fibonacci(3)
 12:||fibonacci(2)
 13:|||fibonacci(1)
 14:|||fibonacci(0)
 15:||fibonacci(1)
Done
```

```
5
```

There are many techniques used in the above decorator.

Why use a variable number of arguments in `wrapper`

To decorate any function with possibly different number of arguments.

Why decorate the wrapper with `@functools.wraps(f)`?

- Ensures some attributes (such as `__name__`) of the wrapper function is the same as those of `f`.
- Add useful attributes. E.g., `__wrapped__` stores the original function so we can undo the decoration.

```
fibonacci, fibonacci_decorated = fibonacci.__wrapped__, fibonacci  # recover
print('original fibonacci:')
print(fibonacci(5))

fibonacci = fibonacci_decorated  # decorate
print('decorated fibonacci:')
print(fibonacci(5))
```

```
original fibonacci:
5
decorated fibonacci:
  1:fibonacci(5)
  2:|fibonacci(4)
  3:||fibonacci(3)
  4:|||fibonacci(2)
  5:||||fibonacci(1)
  6:||||fibonacci(0)
  7:|||fibonacci(1)
  8:||fibonacci(2)
  9:|||fibonacci(1)
 10:|||fibonacci(0)
 11:|fibonacci(3)
 12:||fibonacci(2)
```

(continues on next page)

(continued from previous page)

```
 13:|||fibonacci(1)
 14:|||fibonacci(0)
 15:||fibonacci(1)
Done
5
```

How to use decorator to improve recursion?

We can also use a decorator to make recursion more efficient by caching the return values.`cache` is a dictionary where `cache[n]` stores the computed value of F_n to avoid redundant computations.

```
def caching(f):
    '''Return a decorator that caches a function with a single argument.'''
    @functools.wraps(f)
    def wrapper(n):
        if n not in cache:
            cache[n] = f(n)
        else:
            print('read from cache')
        return cache[n]

    cache = {}
    wrapper.clear_cache = lambda : cache.clear()  # add method to clear cache
    return wrapper


@print_function_call
@caching
def fibonacci(n):
    return fibonacci(n - 1) + fibonacci(n - 2) if n > 1 else 1 if n == 1 else 0
```

```
fibonacci(5)
fibonacci(5)
fibonacci.clear_cache()
fibonacci(5)
```

```
  1:fibonacci(5)
  2:|fibonacci(4)
  3:||fibonacci(3)
  4:|||fibonacci(2)
  5:||||fibonacci(1)
  6:||||fibonacci(0)
  7:|||fibonacci(1)
read from cache
  8:||fibonacci(2)
read from cache
  9:|fibonacci(3)
read from cache
Done
  1:fibonacci(5)
read from cache
Done
  1:fibonacci(5)
  2:|fibonacci(4)
  3:||fibonacci(3)
  4:|||fibonacci(2)
```

(continues on next page)

(continued from previous page)

```
  5:||||fibonacci(1)
  6:||||fibonacci(0)
  7:|||fibonacci(1)
read from cache
  8:||fibonacci(2)
read from cache
  9:|fibonacci(3)
read from cache
Done
```

```
5
```

A method `clear_cache` is added to the wrapper to clear the cache.`lambda <argument list> : <expression>`is called a *lambda* expression, which conveniently defines an *anonymous function*.

```
type(fibonacci.clear_cache), fibonacci.clear_cache.__name__
```

```
(function, '<lambda>')
```

9.7 Module

How to create a module?

To create a module, simply put the code in a python source file `<module name>.py` in

- the current directory, or
- a python *site-packages* directory in system path.

```
import sys
print(sys.path)
```

```
['/home/chungc/cs1302/CS1302ICP/Lecture6', '/home/chungc/anaconda3/lib/python37.zip',
↪'/home/chungc/anaconda3/lib/python3.7', '/home/chungc/anaconda3/lib/python3.7/lib-
↪dynload', '', '/home/chungc/anaconda3/lib/python3.7/site-packages', '/home/chungc/
↪anaconda3/lib/python3.7/site-packages/IPython/extensions', '/home/chungc/.ipython']
```

For example, to create a module for generating Fibonacci numbers:

```
%more fibonacci.py
```

```
import fibonacci as fib # as statement shortens name
help(fib)
```

```
Help on module fibonacci:

NAME
    fibonacci - Contain functions for generating fibonacci numbers.

FUNCTIONS
    fibonacci(n)
        Returns the Fibonacci number of order n.

```

(continues on next page)

(continued from previous page)

```
    fibonacci_iteration(n)
        Returns the Fibonacci number of order n but without recursion.

FILE
    /home/chungc/cs1302/CS1302ICP/Lecture6/fibonacci.py
```

```
print(fib.fibonacci(5))
print(fib.fibonacci_iteration(5))
```

```
5
5
```

CHAPTER

TEN

LISTS AND TUPLES

10.1 Motivation of composite data type

The following code calculates the average of five numbers:

```
def average_five_numbers(n1, n2, n3, n4, n5):
    return (n1 + n2 + n3 + n4 + n5) / 5


average_five_numbers(1, 2, 3, 4, 5)
```

```
3.0
```

What about using the above function to compute the average household income in Hong Kong.The labor size in Hong Kong in 2018 is close to 4 million.

- Should we create a variable to store the income of each individual?
- Should we recursively apply the function to groups of five numbers?

What we need is

- a *composite data type* that can keep a variable numbers of items, so that
- we can then define a function that takes an object of the *composite data type*,
- and returns the average of all items in the object.

How to store a sequence of items in Python?

`tuple` and `list` are two built-in classes for ordered collections of objects of possibly different types.

Indeed, we have already used tuples and lists before.

```
%%mytutor -h 300
a_list = '1 2 3'.split()
a_tuple = (lambda *args: args)(1,2,3)
a_list[0] = 0
a_tuple[0] = 0
```

```
<IPython.lib.display.IFrame at 0x7f5e4c877828>
```

What is the difference between tuple and list?

- List is *mutable* so programmers can change its items.
- Tuple is *immutable* like `int`, `float`, and `str`, so

- programmers can be certain the content stay unchanged, and
- Python can preallocate a fixed amount of memory to store its content.

10.2 Constructing sequences

How to create tuple/list?

Mathematicians often represent a set of items in two different ways:

1. Roster notation, which enumerates the elements in the sequence. E.g., ${0, 1, 4, 9, 16, 25, 36, 49, 64, 81}$
2. Set-builder notation, which describes the content using a rule for constructing the elements. ${x^2 | x \in \mathbb{N}, x < 10}$, namely the set of perfect squares less than 100.

Python also provides two corresponding ways to create a tuple/list:

1. Enclosure
2. Comprehension

How to create a tuple/list by enumerating its items?

To create a tuple, we enclose a comma separated sequence by parentheses:

```
%%mytutor -h 450
empty_tuple = ()
singleton_tuple = (0,)   # why not (0)?
heterogeneous_tuple = (singleton_tuple,
                       (1, 2.0),
                       print)
enclosed_starred_tuple = (*range(2),
                          *'23')
```

```
<IPython.lib.display.IFrame at 0x7f5e4c877ba8>
```

Note that:

- If the enclosed sequence has one term, there must be a comma after the term.
- The elements of a tuple can have different types.
- The unpacking operator `*` can unpack an iterable into a sequence in an enclosure.

To create a list, we use square brackets to enclose a comma separated sequence of objects.

```
%%mytutor -h 450
empty_list = []
singleton_list = [0]  # no need to write [0,]
heterogeneous_list = [singleton_list,
                      (1, 2.0),
                      print]
enclosed_starred_list = [*range(2),
                         *'23']
```

```
<IPython.lib.display.IFrame at 0x7f5e4c877588>
```

We can also create a tuple/list from other iterables using the constructors `tuple`/`list` as well as addition and multiplication similar to `str`.

```
%%mytutor -h 950
str2list = list('Hello')
str2tuple = tuple('Hello')
range2list = list(range(5))
range2tuple = tuple(range(5))
tuple2list = list((1, 2, 3))
list2tuple = tuple([1, 2, 3])
concatenated_tuple = (1,) + (2, 3)
concatenated_list = [1, 2] + [3]
duplicated_tuple = (1,) * 2
duplicated_list = 2 * [1]
```

```
<IPython.lib.display.IFrame at 0x7f5e37fd8128>
```

Exercise Explain the difference between following two expressions. Why a singleton tuple must have a comma after the item.

```
print((1+2)*2,
      (1+2,)*2, sep='\n')
```

```
6
(3, 3)
```

`(1+2)*2` evaluates to `6` but `(1+2,)*2` evaluates to `(3,3)`.

- The parentheses in `(1+2)` indicate the addition needs to be performed first, but
- the parentheses in `(1+2,)` creates a tuple.

Hence, singleton tuple must have a comma after the item to differentiate these two use cases.

How to use a rule to construct a tuple/list?

We can specify the rule using a comprehension,which we have used in a generator expression.E.g., the following is a python one-liner that returns a generator for prime numbers.

```
all?
prime_sequence = lambda stop: (x for x in range(2, stop)
                               if all(x % divisor for divisor in range(2, x)))
print(*prime_sequence(100))
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

There are two comprehensions used:

- In `all(x % divisor for divisor in range(2, x))`, the comprehension creates a generator of remainders to the function `all`, which returns true if all the remainders are `True` in boolean expression.
- In the return value `(x for x in range(2, stop) if ...)` of the anonymous function, the comprehension creates a generator of numbers from 2 to `stop-1` that satisfy the condition of the `if` clause.

Exercise Use comprehension to define a function `composite_sequence` that takes a non-negative integer `stop` and returns a generator of composite numbers strictly smaller than `stop`. Use `any` instead of `all` to check if a number is composite.

```
any?
### BEGIN SOLUTION
composite_sequence = lambda stop: (x for x in range(2, stop)
```

(continues on next page)

(continued from previous page)

```
                                if any(x % divisor == 0 for divisor in range(2, x)))
### END SOLUTION

print(*composite_sequence(100))
```

```
4 6 8 9 10 12 14 15 16 18 20 21 22 24 25 26 27 28 30 32 33 34 35 36 38 39 40 42 44 45␣
↪46 48 49 50 51 52 54 55 56 57 58 60 62 63 64 65 66 68 69 70 72 74 75 76 77 78 80 81␣
↪82 84 85 86 87 88 90 91 92 93 94 95 96 98 99
```

We can construct a list instead of a generator using comprehension:

```
print(list(x**2 for x in range(10)))  # Use the list constructor
print([x**2 for x in range(10)])      # Enclose comprehension by brackets
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can also use comprehension to construct a tuple:

```
print(tuple(x**2 for x in range(10))) # Use the tuple constructor
```

```
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
```

Exercise Explain the difference between the following expressions.

```
print((x**2 for x in range(10)),
      (*(x**2 for x in range(10)),), sep='\n')
```

```
<generator object <genexpr> at 0x7f5e37ff4408>
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
```

- The first is a generator expression, not a tuple.
- The second is a tuple constructed by enclosing the sequence from unpacking the generator.There must be a comma after the generator since there is only one enclosed term, even though that term generates multiple items.

Exercise Explain the difference between the following expressions.

```
print([x for x in range(10)],
      [(lambda arg: arg)(x for x in range(10))], sep='\n')
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[<generator object <genexpr> at 0x7f5e37ff44f8>]
```

- In the second expression, the comprehension provided as an argument to a function becomes a generator object,which is returned by the anonymous function and enclosed to form the singleton list.
- In the first expression, the comprehension is not converted to a generator.

With list comprehension, we can simulate a sequence of biased coin flips.

```
from random import random as rand
p = rand()  # unknown bias
coin_flips = ['H' if rand() <= p else 'T' for i in range(1000)]
print('Chance of head:', p)
print('Coin flips:',*coin_flips)
```

```
Chance of head: 0.12076730849725259
Coin flips: T T T T H T T T T T T T T T T T T T H T T T H T T T T T T T T T T T T T T␣
↪T T T T T T T T T T T T H T T T T T T T T T T T H H T T H T T T T T T T T T T T T T␣
↪H T T T H T T T T T H T H T T T T T H H T T T T T T H T T T T T T T H T T H T H T T␣
↪T T T T T T H T T T H T T T T T T T T T T T T H T T T T T T H H H T T T T T T T T T␣
↪T T H T T T H T T T T T T T T T T T T T T H T T T T T T T T T T T T T T T T T T T T␣
↪T T T T T H T T T T T T T T T H T T T H T T T H T H T T T T T T T T T T T T T T T T␣
↪T T T H T T T T T T T T H T T T T H T T T H T T T T T T T T T T T H T T T T T T T T␣
↪H T T T T T T T T T T T T T T T H T T T T T T H T T T T T T T T T T T T T H T T T T␣
↪T T T H T T T T T T H T T T T T T H T T T T T T T T T T H T T T T T T T T T T H T T␣
↪H T T T T H H T T T T T H T T T T T T T T T T T T T T T T H T T T H T T T T T T T T␣
↪T T T T T T T T T T T T T T T T T T H T T T H H T T H T T T T T T T T T T T T T T T␣
↪T T T H T T T H T T T T T T T T T T T T T T T T T T T T H T T T T H T T T H T T T T␣
↪T T T T T T T T T T T T T T T T H T T T T T T T T T T T T T T H T T T T T T T T T T␣
↪T T T T T T T T T T T T H T T T T T H T T T T T T T T T T H T T T T T T T T T T T T␣
↪T T T T T T T H T T T T T T H T T T H T T T T T T T H T T T T T T H T T T T T T T H␣
↪T T T T T T H T T T T T T T T T T H T T T T H T T T H T T T T H T T H H H H T T T T␣
↪T T H T T T T T T H T T T T T T T T H T T T T T T T T T T T T T T T T H T T T T T T␣
↪T T H T T T H T T T T T T H H T T T H T T T T H T T T T H T T T T T T T T T H T T T␣
↪T T T T T H T T T
```

```
T T T T T H T T T T T T T T T H H T T H T T T T T T T T T T T T T H T T T T T T T H T␣
↪H T T T H T T T T T H T T T T T T T T T T H T T H T T T T T T T T T T T T T T T T T␣
↪T T H T T T T H T H T T T T T T T T T H T T T H T T T T H T H T T T H T T H T T T T␣
↪T T T T T T T T T T T T H T T T T T T T H T T T T T T T T T T T T T T T T T T T T T␣
↪T T T T T T T T T T T H T T T T H T T T T T T T T T T T H T T T T T T T T T T T T T␣
↪T T T T T T T T T T T T T T T T T T T T H T T H T T T T T
```

We can then estimate the bias by the fraction of heads coming up.

```
def average(seq):
    return sum(seq)/len(seq)

head_indicators = [1 if outcome == 'H' else 0 for outcome in coin_flips]
fraction_of_heads = average(head_indicators)
print('Fraction of heads:', fraction_of_heads)
```

```
Fraction of heads: 0.12
```

Note that `sum` and `len` returns the sum and length of the sequence.

Exercise Define a function `variance` that takes in a sequence `seq` and returns the variance of the sequence.

```
def variance(seq):
    ### BEGIN SOLUTION
    return sum(i**2 for i in seq)/len(seq) - average(seq)**2
    ### END SOLUTION

delta = (variance(head_indicators)/len(head_indicators))**0.5
print('95% confidence interval: [{:.2f},{:.2f}]'.format(p-2*delta,p+2*delta))
```

```
95% confidence interval: [0.10,0.14]
```

10.3 Selecting items in a sequence

How to traverse a tuple/list?

Instead of calling the dunder method directly, we can use a for loop to iterate over all the items in order.

```
a = (*range(5),)
for item in a: print(item, end=' ')
```

```
0 1 2 3 4
```

To do it in reverse, we can use the `reversed` function.

```
reversed?
a = [*range(5)]
for item in reversed(a): print(item, end=' ')
```

```
4 3 2 1 0
```

We can also traverse multiple tuples/lists simultaneously by `zip`ping them.

```
zip?
a = (*range(5),)
b = reversed(a)
for item1, item2 in zip(a,b):
    print(item1,item2)
```

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

How to select an item in a sequence?

Sequence objects such as `str`/`tuple`/`list` implements the *getter method* `__getitem__` to return their items.

We can select an item by subscription

```
a[i]
```

where `a` is a list and `i` is an integer index.

A non-negative index indicates the distance from the beginning.

$$\boldsymbol{a} = (a_0, ..., a_{n-1})$$

```
a = (*range(10),)
print(a)
print('Length:', len(a))
print('First element:',a[0])
print('Second element:',a[1])
print('Last element:',a[len(a)-1])
print(a[len(a)]) # IndexError
```

```
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
Length: 10
First element: 0
Second element: 1
Last element: 9
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-20-8793aa5ed482> in <module>
      5 print('Second element:',a[1])
      6 print('Last element:',a[len(a)-1])
----> 7 print(a[len(a)]) # IndexError

IndexError: tuple index out of range
```

`a[i]` with `i >= len(a)` results in an `IndexError`.

A negative index represents a negative offset from an imaginary element one past the end of the sequence.

$$\begin{aligned}\boldsymbol{a} &= (a_0, ..., a_{n-1}) \\ &= (a_{-n}, ..., a_{-1})\end{aligned}$$

```
a = [*range(10)]
print(a)
print('Last element:',a[-1])
print('Second last element:',a[-2])
print('First element:',a[-len(a)])
print(a[-len(a)-1]) # IndexError
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Last element: 9
Second last element: 8
First element: 0
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-21-6f6376dfba21> in <module>
      4 print('Second last element:',a[-2])
      5 print('First element:',a[-len(a)])
----> 6 print(a[-len(a)-1]) # IndexError

IndexError: list index out of range
```

`a[i]` with `i < -len(a)` results in an `IndexError`.

How to select multiple items?

We can use a slicing to select a range of items:

```
a[start:stop]
a[start:stop:step]
```

where `a` is a list;

- `start` is an integer representing the index of the starting item in the selection;
- `stop` is an integer that is one larger than the index of the last item in the selection; and

- `step` is an integer that specifies the step/stride size through the list.

```
a = (*range(10),)
print(a[1:4])
print(a[1:4:2])
```

```
(1, 2, 3)
(1, 3)
```

The parameters take their default values if missing or equal to None.

```
a = [*range(10)]
print(a[:4])   # start defaults to 0
print(a[1:])   # stop defaults to len(a)
print(a[1:4:]) # step defaults to 1
```

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

They can take negative values.

```
print(a[-1:])
print(a[:-1])
print(a[::-1])
```

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

They can also take a mixture of negative and postive values.

```
print(a[-1:1])       # equal [a[-1], a[0]]?
print(a[1:-1])       # equal []?
print(a[1:-1:-1])    # equal [a[1], a[0]]?
print(a[-100:100])   # result in IndexError like subscription?
```

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

We can now implement a practical sorting algorithm called quicksort to sort a sequence.

```
import random


def quicksort(seq):
    '''Return a sorted list of items from seq.'''
    if len(seq) <= 1:
        return list(seq)
    i = random.randint(0, len(seq) - 1)
    pivot, others = seq[i], [*seq[:i], *seq[i + 1:]]
    left = quicksort([x for x in others if x < pivot])
    right = quicksort([x for x in others if x >= pivot])
    return [*left, pivot, *right]
```

(continues on next page)

(continued from previous page)

```
seq = [random.randint(0, 99) for i in range(10)]
print(seq, quicksort(seq), sep='\n')
```

```
[79, 39, 32, 63, 22, 22, 73, 93, 58, 76]
[22, 22, 32, 39, 58, 63, 73, 76, 79, 93]
```

The above recursion creates a sorted list as `[*left, pivot, *right]` where

- `pivot` is a randomly picked an item in `seq`,
- `left` is the sorted list of items smaller than `pivot`, and
- `right` is the sorted list of items no smaller than `pivot`.

The base case happens when `seq` contains at most one item, in which case `seq` is already sorted.

There is a built-in function `sorted` for sorting a sequence. It uses the Timsort algorithm.

```
sorted?
sorted(sorted(seq))
```

```
[22, 22, 32, 39, 58, 63, 73, 76, 79, 93]
```

10.4 Mutating a list

For list (but not tuple), subscription and slicing can also be used as the target of an assignment operation to mutate the list.

```
%%mytutor -h 300
b = [*range(10)]  # aliasing
b[::2] = b[:5]
b[0:1] = b[:5]
b[::2] = b[:5]  # fails
```

```
<IPython.lib.display.IFrame at 0x7f5e37ff6748>
```

Last assignment fails because `[::2]` with step size not equal to `1` is an *extended slice*, which can only be assigned to a list of equal size.

What is the difference between mutation and aliasing?

In the previous code:

- The first assignment `b = [*range(10)]` is aliasing, which gives the list the target name/identifier `b`.
- Other assignments such as `b[::2] = b[:5]` are mutations that calls `__setitem__` because the target `b[::2]` is not an identifier.

Exercise Explain the outcome of the following checks of equivalence?

```
%%mytutor -h 400
a = [10, 20, 30, 40]
b = a
print('a is b? {}'.format(a is b))
```

(continues on next page)

(continued from previous page)

```
print('{} == {}? {}'.format(a, b, a == b))
b[1:3] = b[2:0:-1]
print('{} == {}? {}'.format(a, b, a == b))
```

```
<IPython.lib.display.IFrame at 0x7f5e37ff6ac8>
```

- `a is b` and `a == b` returns `True` because the assignment `b = a` makes `b` an alias of the same object `a` points to.
- In particular, the operation`b[1:3] = b[2:0:-1]` affects the same list `a` points to.

Why mutate a list?

The following is another implementation of `composite_sequence` that takes advantage of the mutability of list.

```
def sieve_composite_sequence(stop):
    is_composite = [False] * stop  # initialization
    for factor in range(2,stop):
        if is_composite[factor]: continue
        for multiple in range(factor*2,stop,factor):
            is_composite[multiple] = True
    return (x for x in range(4,stop) if is_composite[x])

for x in sieve_composite_sequence(100): print(x, end=' ')
```

```
4 6 8 9 10 12 14 15 16 18 20 21 22 24 25 26 27 28 30 32 33 34 35 36 38 39 40 42 44 45␣
↪46 48 49 50 51 52 54 55 56 57 58 60 62 63 64 65 66 68 69 70 72 74 75 76 77 78 80 81␣
↪82 84 85 86 87 88 90 91 92 93 94 95 96 98 99
```

The algorithm

1. changes `is_composite[x]` from `False` to `True` if `x` is a multiple of a smaller number `factor`, and
2. returns a generator that generates composite numbers according to `is_composite`.

Exercise Is `sieve_composite_sequence` more efficient than your solution `composite_sequence`? Why?

```
for x in composite_sequence(10000): pass
```

```
for x in sieve_composite_sequence(1000000): pass
```

The line `if is_composite[factor]: continue` avoids the redundant computations of checking composite factors.

Exercise Note that the multiplication operation `*` is the most efficient way to initialize a 1D list with a specified size, but we should not use it to initialize a 2D list. Fix the following code so that `a` becomes `[[1, 0], [0, 1]]`.

```
%%mytutor -h 250
a = [[0] * 2] * 2
a[0][0] = a[1][1] = 1
print(a)
```

```
<IPython.lib.display.IFrame at 0x7f5e37fe92b0>
```

```
### BEGIN SOLUTION
a = [[0] * 2 for i in range(2)]
### END SOLUTION
```

(continues on next page)

(continued from previous page)

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

```
[[1, 0], [0, 1]]
```

10.5 Different methods to operate on a sequence

The following compares the lists of public attributes for `tuple` and `list`.

- We determine membership using the operator `in` or `not in`.
- Different from the keyword `in` in a for loop, operator `in` calls the method `__contains__`.

```
list_attributes = dir(list)
tuple_attributes = dir(tuple)

print(
    'Common attributes:', ', '.join([
        attr for attr in list_attributes
        if attr in tuple_attributes and attr[0] != '_'
    ]))

print(
    'Tuple-specific attributes:', ', '.join([
        attr for attr in tuple_attributes
        if attr not in list_attributes and attr[0] != '_'
    ]))

print(
    'List-specific attributes:', ', '.join([
        attr for attr in list_attributes
        if attr not in tuple_attributes and attr[0] != '_'
    ]))
```

```
Common attributes: count, index
Tuple-specific attributes:
List-specific attributes: append, clear, copy, extend, insert, pop, remove, reverse,␣
↪sort
```

- There are no public tuple-specific attributes, and
- all the list-specific attributes are methods that mutate the list, except `copy`.

The common attributes

- `count` method returns the number of occurrences of a value in a tuple/list, and
- `index` method returns the index of the first occurrence of a value in a tuple/list.

```
%%mytutor -h 300
a = (1,2,2,4,5)
print(a.index(2))
print(a.count(2))
```

```
<IPython.lib.display.IFrame at 0x7f5e37ff6f28>
```

`reverse` method reverses the list instead of returning a reversed list.

```
%%mytutor -h 300
a = [*range(10)]
print(reversed(a))
print(*reversed(a))
print(a.reverse())
```

```
<IPython.lib.display.IFrame at 0x7f5e37ff6a20>
```

- `copy` method returns a copy of a list.
- `tuple` does not have the `copy` method but it is easy to create a copy by slicing.

```
%%mytutor -h 400
a = [*range(10)]
b = tuple(a)
a_reversed = a.copy()
a_reversed.reverse()
b_reversed = b[::-1]
```

```
<IPython.lib.display.IFrame at 0x7f5e37ff6e48>
```

`sort` method sorts the list *in place* instead of returning a sorted list.

```
%%mytutor -h 300
import random
a = [random.randint(0,10) for i in range(10)]
print(sorted(a))
print(a.sort())
```

```
<IPython.lib.display.IFrame at 0x7f5e37fe90f0>
```

- `extend` method that extends a list instead of creating a new concatenated list.
- `append` method adds an object to the end of a list.
- `insert` method insert an object to a specified location.

```
%%mytutor -h 300
a = b = [*range(5)]
print(a + b)
print(a.extend(b))
print(a.append('stop'))
print(a.insert(0,'start'))
```

```
<IPython.lib.display.IFrame at 0x7f5e37fe9160>
```

- `pop` method deletes and return the last item of the list.
- `remove` method removes the first occurrence of a value in the list.
- `clear` method clears the entire list.

We can also use the function `del` to delete a selection of a list.

```
%%mytutor -h 300
a = [*range(10)]
del a[::2]
print(a.pop())
print(a.remove(5))
print(a.clear())
```

```
<IPython.lib.display.IFrame at 0x7f5e37fe9940>
```

CHAPTER

ELEVEN

DICTIONARIES AND SETS

11.1 Motivation for associative container

The following code simulates the outcomes from rolling a dice multiple times.

```
import random

dice_rolls = [random.randint(1,6) for i in range(10)]
print(*dice_rolls)
```

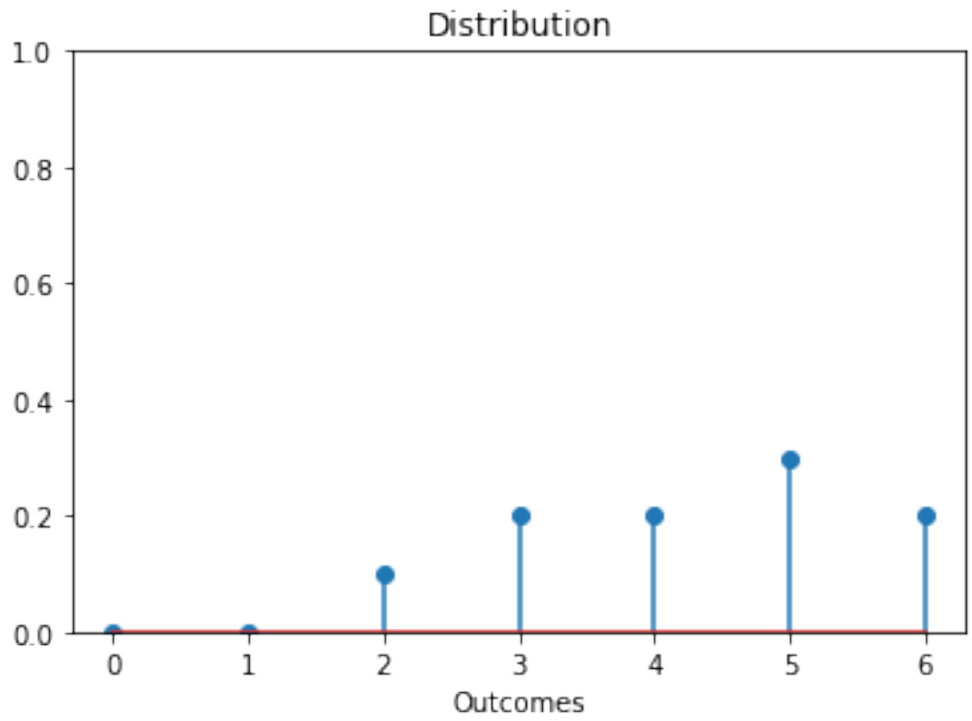
```
4 5 3 4 6 5 6 2 3 5
```

What is the distribution, i.e., fractional counts?

```
distribution = [dice_rolls.count(i) / len(dice_rolls) for i in range(7)]

import matplotlib.pyplot as plt
plt.stem(range(7), distribution, use_line_collection=True)
plt.xlabel('Outcomes')
plt.title('Distribution')
plt.ylim(0, 1)
```

```
(0.0, 1.0)
```



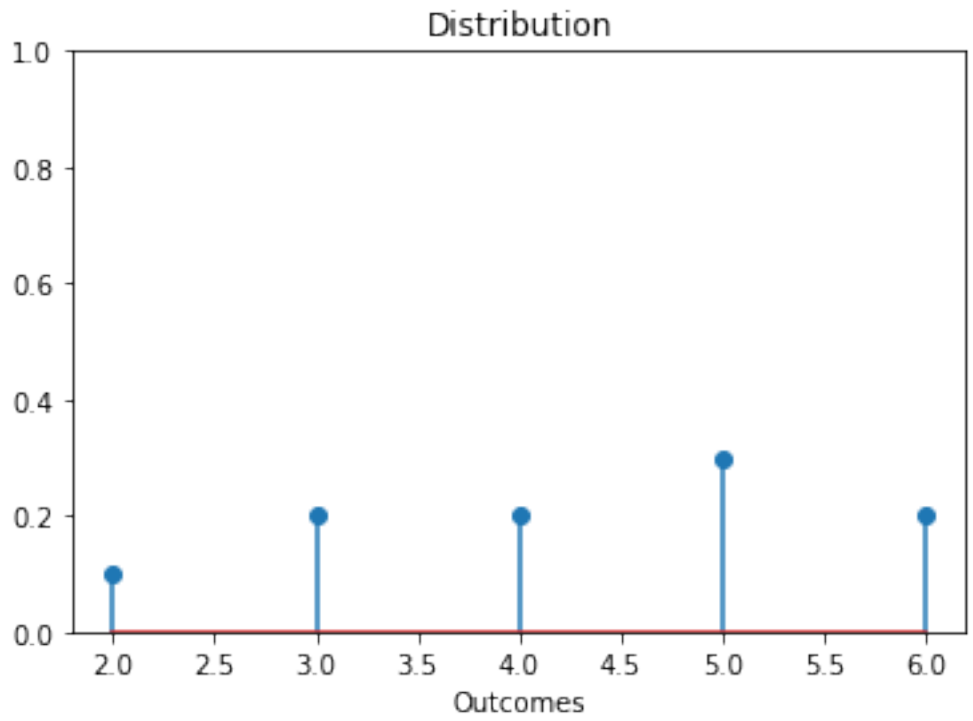
In the above code, `distribution[i]` stores the fractional count of outcome `i`.

However, `distribution[0]` is `0` because a dice does not have outcome `0`. Can we avoid such redundancy?

```
distinct_outcomes = [
    outcome for outcome in range(1, 7) if dice_rolls.count(outcome) > 0
]
distribution = [
    dice_rolls.count(distinct_outcomes[i]) / len(dice_rolls)
    for i in range(len(distinct_outcomes))
]

import matplotlib.pyplot as plt
plt.stem(distinct_outcomes, distribution, use_line_collection=True)
plt.xlabel('Outcomes')
plt.title('Distribution')
plt.ylim(0, 1)
```

```
(0.0, 1.0)
```



In the above code,

- `distinct_outcomes` stores the list of distinct outcomes, and
- `distribution[distinct_outcomes[i]]` stores the fractional count of the `i`-th distinct outcome.

What about finding the distribution of characters in an article?There are 1,112,064 unicode characters.

- How obtain the distribution efficiently without creating an entry for each unicode character?
- How to compute the set of distinct characters efficiently without iterating over the set of all unicode characters?
- Can we index `distribution` directly by the set of distinct characters?

What we need is a composite data type that

- can keep a set of *unique keys of different types* (such as the characters in our example), and
- associate to different keys possibly different *values of any types* such as (the fractional counts of the characters).

Such data structure is called an associative container.

How to use associative containers in Python?

There are two built-in classes for associative containers:

- `set` can store a set of unique keys of possibly different types.
- `dict`ionary can store a set of key-value pairs.

We have already used sets and dictionaries before.

```
%%mytutor -h 400
a = (lambda **kwargs: kwargs)(start=0, stop=5, step=1)
b = set([1,1,2,3,3,3])
assert len(a) == len(b)
```

```
<IPython.lib.display.IFrame at 0x7fa62d8f6c88>
```

Both `set` and `dict`

- implement `len` method that returns the number of keys, and
- are mutable, so we can mutate their keys and values.

11.2 Constructing associative containers

How to create set/dictionary?

Similar to tuple/list, we can use enclosure, constructors, and comprehension.

How to create a set/dict by enumerating its keys/values?

For `dict`, enclose a comma-separated sequence of `key : value` pairs by braces `{` and `}`.

```
%%mytutor -h 350
empty_dictionary = {}
a = {'a': 0, 'b': 1}
b = {**a, 'c': 0, 'd': 1}
```

```
<IPython.lib.display.IFrame at 0x7fa62d8aeba8>
```

For `set`, omit `: value`.

```
%%mytutor -h 300
a = {(1, 2.0), print, *range(2), *'23'}
empty_set = {*()}  # Why not use {}?
```

```
<IPython.lib.display.IFrame at 0x7fa62d8aee48>
```

We can also create a set/dictionary from other objects using their constructors `set`/`dict`.

```
%%mytutor -h 550
empty_set = set()
string2set = set('abc')
range2set = set(range(2))
list2set = set(['abc',range(2)])
set2set = set(list2set)
```

```
<IPython.lib.display.IFrame at 0x7fa62d8b20f0>
```

```
%%mytutor -h 650
empty_dict = dict()
enumerate2dict = dict(enumerate('abc'))
zip2dict = dict(zip('abc','123'))
kwargs2dict = dict(one=1,two=2)
dict2dict = dict(kwargs2dict)
```

```
<IPython.lib.display.IFrame at 0x7fa62d8b22e8>
```

Exercise `dict` also has a *class method* `fromkeys` to construct a dictionary with keys from iterable pointing to a default value. Create a dictionary using `fromkeys` with keys being the non-negative integers smaller than `100` and values being `0`.

Hint: Use `dict.fromkeys` since a class method is bound to the class rather than an object of the class.

```
dict.fromkeys?
### BEGIN SOLUTION
fromkeys_dict = dict.fromkeys(range(100),0)
### END SOLUTION

# test
assert all(fromkeys_dict[k] == 0 for k in fromkeys_dict)
```

How to use a rule to construct a set/dictionary?

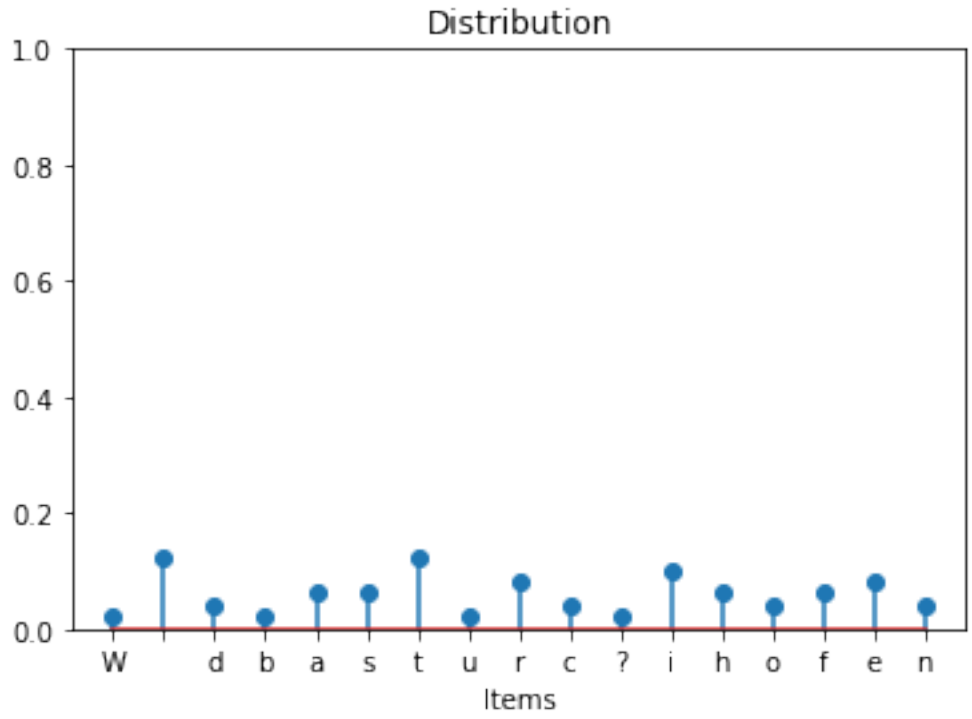
The following function uses a one-line dictionary comprehension to return the distribution of items in a sequence:

```
def distribute(seq):
    return {k : seq.count(k)/len(seq) for k in set(seq)}
```

```
import matplotlib.pyplot as plt

def plot_distribution(seq):
    dist = distribute(seq)
    plt.stem(dist.keys(),   # set-like view of the keys
             dist.values(), # view of the values
             use_line_collection=True)
    plt.xlabel('Items')
    plt.title('Distribution')
    plt.ylim(0, 1)

plot_distribution('What is the distribution of different characters?')
```



- The object methods `keys` and `values` provide a dynamic view of the keys.
- Unlike a copy, subsequent changes to the dictionary are also reflected in a previously returned view.

- `items` provides a set-like view of the key-value pairs.

```
%%mytutor -h 500
a = dict(enumerate('abc'))
views = a.keys(), a.values(), a.items()
a.pop(1)    # remove the key 1 and its associated value
a.popitem() # remove and return a key-value pair
a.clear()   # clear the dictionary
```

```
<IPython.lib.display.IFrame at 0x7fa62d839198>
```

`set` has `pop` and `clear` but not `popitem`. However, `set.pop` behaves like `dict.popitem` instead of `dict.pop`. (Why?)

```
%%mytutor -h 250
a = set('abc')
a.pop()     # remove and return an element
a.clear()   # clear the set
```

```
<IPython.lib.display.IFrame at 0x7fa62d7f0438>
```

Exercise Use one-line comprehension to return a set of composite numbers smaller than `stop`.

Hint: You do not need to worry about duplicate elements for `set`.

```
def composite_set(stop):
    ### BEGIN SOLUTION
    return {x for factor in range(2,stop) for x in range(factor*2,stop,factor)}
    ### END SOLUTION

print(*sorted(composite_set(100)))
```

```
4 6 8 9 10 12 14 15 16 18 20 21 22 24 25 26 27 28 30 32 33 34 35 36 38 39 40 42 44 45␣
↪46 48 49 50 51 52 54 55 56 57 58 60 62 63 64 65 66 68 69 70 72 74 75 76 77 78 80 81␣
↪82 84 85 86 87 88 90 91 92 93 94 95 96 98 99
```

11.3 Hashability

For `set` and `dict`,

- identical keys are merged to the same entry even though
- values associated with different keys can be the same.

```
%%mytutor -h 350
a = {0: 'a', 0.0: 'b', 2: 'b'}
b = {0j, 0, 0.0, '', False}
assert 0 == 0.0 == 0j == False != ''
```

```
<IPython.lib.display.IFrame at 0x7fa62d7f07b8>
```

This is implemented efficiently by *hashing*. A key must be a hashable object which:

- has a hash value (returned by `__hash__` method) that never changes during its lifetime, and

- can be compared (using `__eq__` method) to other objects.*Hashable objects which compare equal must have the same hash value.*

```
import collections

for i in 0, 0.0, 0j, '', False, (), [], {}, set(), frozenset():
    if isinstance(i, collections.abc.Hashable):
        print('{} is hashable. E.g., hash({!r}) == {}'.format(type(i),i,hash(i)))
    else:
        print('{} is NOT hashable.'.format(type(i)))
```

```
<class 'int'> is hashable. E.g., hash(0) == 0
<class 'float'> is hashable. E.g., hash(0.0) == 0
<class 'complex'> is hashable. E.g., hash(0j) == 0
<class 'str'> is hashable. E.g., hash('') == 0
<class 'bool'> is hashable. E.g., hash(False) == 0
<class 'tuple'> is hashable. E.g., hash(()) == 3527539
<class 'list'> is NOT hashable.
<class 'dict'> is NOT hashable.
<class 'set'> is NOT hashable.
<class 'frozenset'> is hashable. E.g., hash(frozenset()) == 133146708735736
```

Why the key should be hashable?What is the use of a hash value?

Associative containers are implemented as *hash tables* for efficient lookup of key values.

```
%%html
<iframe width="912" height="513" src="https://www.youtube.com/embed/LPzN8jgbnvA"␣
↪frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media;␣
↪gyroscope; picture-in-picture" allowfullscreen></iframe>
```

```
<IPython.core.display.HTML object>
```

Most mutable objects are not hashable. Why?Mutating a key makes it a different key, which is hard to track.

`set` has an immutable counterpart called `frozenset`, but `dict` does not have any immutable counterpart. Why?While elements of a set must be hashable and therefore mostly immutable, dictionary values may be of mutable types.

Python also uses dictionary for its global/local frames.Indeed, hash collisions can slow down the lookup process.

Exercise Why equal objects must have the same hash but different objects may have the same hash? An example is given below:

```
assert hash(0) == hash(0.0) == hash(0j) == hash(False) == hash('') and False != ''
```

1. To avoid duplicate keys occupying different entries in a hash table.
2. Hash collision can be detected by == and handled by collision resolution techniques. To keep the hash table small, hash collision is unavoidable.

11.4 Accessing keys/values

How to traverse a set/dictionary?

Set and dictionaries are iterable.The for loop iterates over the keys.

```
a = set('abcde')
b = dict(enumerate('abcde'))
print(*(element for element in a))
print(*((key,b[key]) for key in b))
a[0]  # TypeError
```

```
b c a e d
(0, 'a') (1, 'b') (2, 'c') (3, 'd') (4, 'e')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-6702302c4a5e> in <module>
      3 print(*(element for element in a))
      4 print(*((key,b[key]) for key in b))
----> 5 a[0]  # TypeError

TypeError: 'set' object is not subscriptable
```

- For the dictionary `b`, we used subscription `b[key]` to access the value associated with `key`.
- Unlike dictionary, set does not implement `__getitem__` and is therefore not subscriptable.

Unlike tuple/list, `b[-1]` does not refer to the value of the last entry. (Dictionary is not ordered.)

```
b[-1]  # KeyError
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-21-784624678be8> in <module>
----> 1 b[-1]  # KeyError

KeyError: -1
```

The above raises a key error because `-1` is not a key in the dictionary `b`.

Dictionary implements the `__setitem__` method so we can enter a key value pair to a dictionary using the assignment operator.

```
b[-1] = 'f'
b[-1]
```

```
'f'
```

To delete a key, we can use the function `del`.

```
del b[-1]
b[-1]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

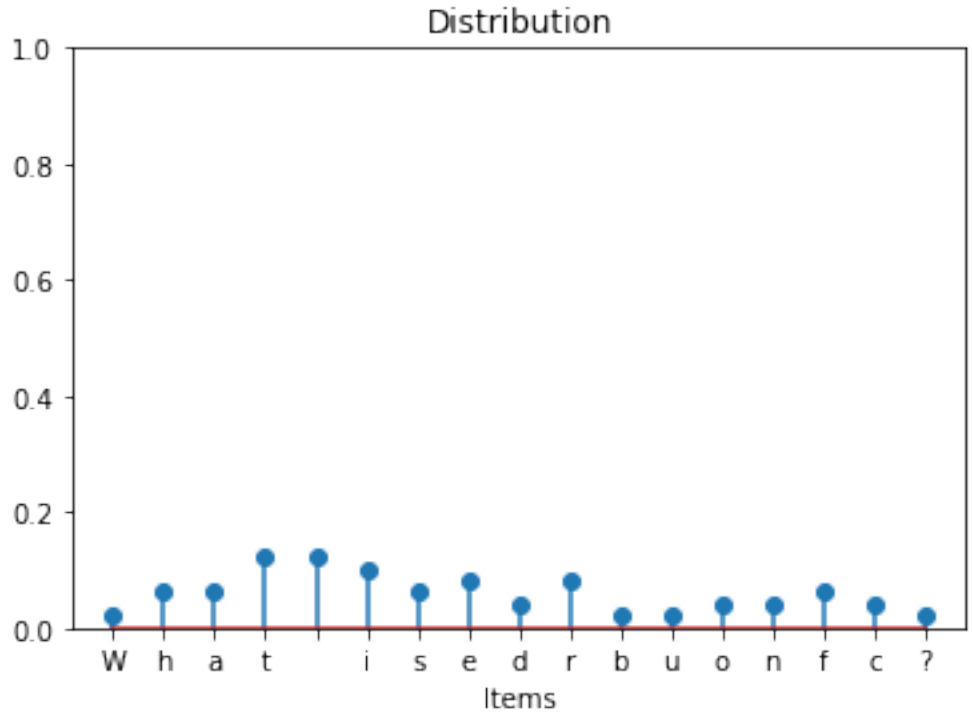
```
<ipython-input-23-a57f788767eb> in <module>
      1 del b[-1]
----> 2 b[-1]

KeyError: -1
```

To avoid key error, we can check if a key is in a dictionary efficiently (due to hashing) using the `in` operator.The following is a different implementation of `distribute`.

```
def distribute(seq):
    dist = {}
    for i in seq:
        dist[i] = (dist[i] if i in dist else 0) + 1/len(seq)
    return dist

plot_distribution('What is the distribution of different characters?')
```



Exercise Unlike the previous implementation using one-line dictionary comprehension, the above alternative implementation uses multiple lines of code to build the dictionary incrementally starting from an empty dictionary.

```
def distribute(seq):
    return {k : seq.count(k)/len(seq) for k in set(seq)}
```

Explain whether the alternative is more efficient.

It is more efficient because

- the alternative implementation traverses `seq` once with near constant time lookup of the key, but
- the list comprehension can traverse `seq` a multiple times linear in `len(seq)`, since every call to `seq.count` has to traverse `seq` once.

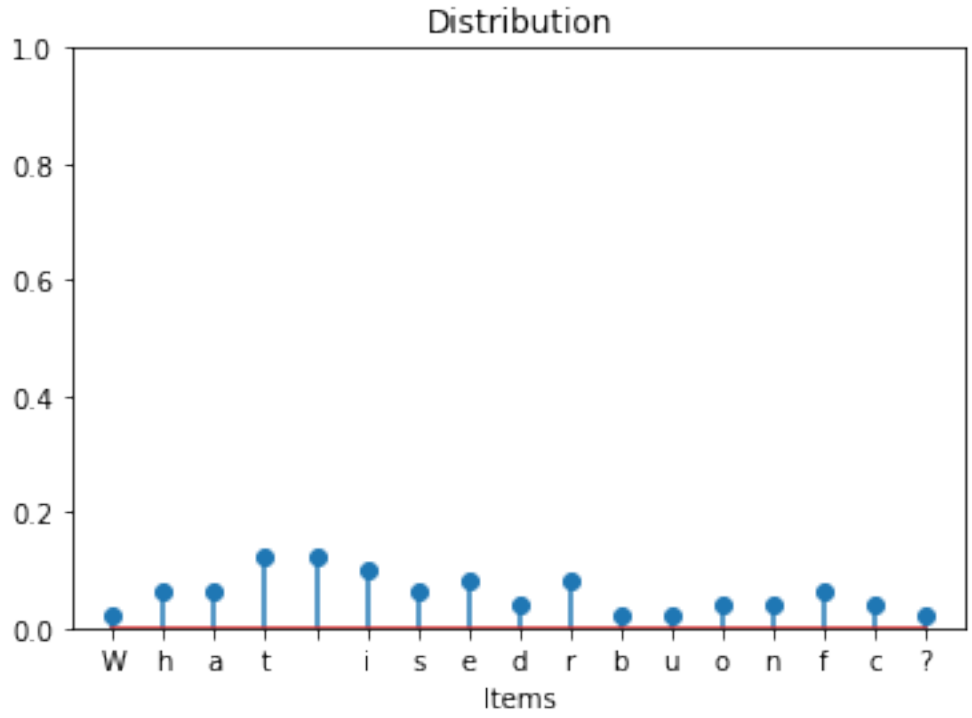
Shorter code needs not be more efficient.

Exercise `dict` also has a getter method `get` that conveniently returns a default value if the key does not exist. Rewrite the alternative implementation of `distribute` to use `get` instead of `in`.

```
dict.get?
```

```
def distribute(seq):
    dist = {}
    for i in seq:
        ### BEGIN SOLUTION
        dist[i] = dist.get(i,0) + 1/len(seq)
        ### END SOLUTION
    return dist

plot_distribution('What is the distribution of different characters?')
```



How to traverse in ascending order of the keys?

We can apply the function `sorted` to a set/dictionary to return a sorted list of the keys.

```
%%mytutor -h 600
a = set(reversed('abcde'))
b = dict(reversed([*enumerate('abcde')]))
sorted_elements = sorted(a)
sorted_keys = sorted(b)
```

```
<IPython.lib.display.IFrame at 0x7fa62d7eb7f0>
```

Exercise Re-implement `plot_distribution` to plot the distribution in ascending order of the keys.

```
def plot_distribution(seq):
    dist = distribute(seq)
```

(continues on next page)

(continued from previous page)

```
    # pyplot.stem(dist.keys(), dist.values(), use_line_collection=True)
    ### BEGIN SOLUTION
    dist_list = sorted(dist.items(), key = lambda p: p[0])
    pyplot.stem([p[0] for p in dist_list], [p[1] for p in dist_list], use_line_
↪collection=True)
    ### END SOLUTION
    pyplot.xlabel('Items')
    pyplot.title('Distribution')
    pyplot.ylim(0, 1)

plot_distribution('What is the distribution of different characters?')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-27-aa27f8ab3ded> in <module>
     10     pyplot.ylim(0, 1)
     11
---> 12 plot_distribution('What is the distribution of different characters?')

<ipython-input-27-aa27f8ab3ded> in plot_distribution(seq)
      4     ### BEGIN SOLUTION
      5     dist_list = sorted(dist.items(), key = lambda p: p[0])
----> 6     pyplot.stem([p[0] for p in dist_list], [p[1] for p in dist_list], use_
↪line_collection=True)
      7     ### END SOLUTION
      8     pyplot.xlabel('Items')

NameError: name 'pyplot' is not defined
```

How to add an element to a set and remove an element from it?

Instead of subscription, `set` has the `add`/`discard`/`remove` methods for adding/removing elements.

```
%%mytutor -h 400
a = set('abc')
a.add('d')
a.discard('a')
a.remove('b')
a.clear()
a.discard('a')  # no error
a.remove('b')   # KeyError
```

```
<IPython.lib.display.IFrame at 0x7fa62d7eb5c0>
```

11.5 Other operators and methods

Unlike `str`/`tuple`/`list`, `set` and `dict` do not implement addition `+` and multiplication `*`:

```
any(hasattr(container, attr) for attr in ('__add__', '__mult__')
    for container in (dict, set, frozenset))
```

```
False
```

Exercise Use the unpacking operators `*` and `**` to concatenate two sets/dictionaries below into a new set/dictionary.

```
set1 = set('abc')
set2 = set('cde')
### BEGIN SOLUTION
concatenated_set = {*set1,*set2}
### END SOLUTION
concatenated_set
```

```
{'a', 'b', 'c', 'd', 'e'}
```

```
dict1 = dict(enumerate('abc'))
dict2 = dict(enumerate('def',start=2))
### BEGIN SOLUTION
concatenated_dict = {**dict1,**dict2}
### END SOLUTION
concatenated_dict
```

```
{0: 'a', 1: 'b', 2: 'd', 3: 'e', 4: 'f'}
```

`set` overloads many other operators:

```
%%mytutor -h 550
a, b = {1,2}, {2,3}

union = a | b
assert all(i in union for i in a) and all(i in union for i in b)

intersection = a & b
assert all(i in a and i in b for i in intersection)

assert intersection <= a <= union  # subset
assert union > b > intersection    # proper superset
assert len(a) + len(b) == len(intersection) + len(union)

symmetric_difference = a ^ b
assert all((i in a or i in b) and not (i in a and i in b)
           for i in symmetric_difference)
assert symmetric_difference == union - intersection
assert set.isdisjoint(intersection, symmetric_difference)
assert len(union) == len(intersection) + len(symmetric_difference)
```

```
<IPython.lib.display.IFrame at 0x7fa62d66cac8>
```

The following uses `&` and `-` to compare the sets of public attributes for `set` and `dict`:

```
set_attributes = {attr for attr in dir(set) if attr[0] != '_'}
dict_attributes = {attr for attr in dir(dict) if attr[0] != '_'}
print('Common attributes:', ', '.join(set_attributes & dict_attributes))
print('dict-specific attributes:', ', '.join(dict_attributes - set_attributes))
print('set-specific attributes:', ', '.join(set_attributes - dict_attributes))
```

```
Common attributes: copy, update, pop, clear
dict-specific attributes: items, get, popitem, values, fromkeys, setdefault, keys
set-specific attributes: issubset, add, remove, symmetric_difference, difference,␣
↪intersection, discard, issuperset, union, isdisjoint, intersection_update,␣
↪difference_update, symmetric_difference_update
```

For `set`, the intersection operation `&` can also be performed by

- the class method `intersection` which returns the intersection of its arguments, and
- the object method `intersection_update` which mutates a set object by intersecting the set with the arguments.

```
%%mytutor -h 300
a = {0,1,2}
b = {1,2,3}
c = set.intersection(a,b,{2,3,4})
a.intersection_update(b,c)
```

```
<IPython.lib.display.IFrame at 0x7fa62d66c748>
```

- All other set-specific methods have an associated operator except `isdisjoint` as shown below.
- The object method for `union` is `update` not `union_update`.

class method	object method	operator
`union`	`update`	`|`
`intersection`	`intersection_update`	`&`
`symmetric_difference`	`symmetric_difference_update`	`^`
`issubset`		`<=`
`issuperset`		`>=`
`isdisjoint`		

`dict` also has an `update` method that can update a dictionary using dictionary, iterables and keyword arguments:

```
%%mytutor -h 300
a = {}
a.update(enumerate('a'),b=2)
b = a.copy()
a.update(b,c=3)
```

```
<IPython.lib.display.IFrame at 0x7fa62d66cac8>
```

Exercise For `dict`, there is also a method called `setdefault`. Use it to define a function `group_by_type` that

- takes a sequence `seq` of objects and
- returns a dictionary `d` such that `d[repr(t)]` returns the list of objects in `seq` of type `t`

If there is no objects of type `t`, raise a key error.

```
def group_by_type(seq):
    group = {}
    for i in seq:
        ### BEGIN SOLUTION
        group.setdefault(repr(type(i)),[]).append(i)
        ### END SOLUTION
    return group

group_by_type([*range(3),
               *'abc',
               *[i/2 for i in range(3)],
               *[(i,) for i in range(3)],
```

(continues on next page)

(continued from previous page)

```
                *[[i] for i in range(3)],
                *[{i} for i in range(3)],
                *[{i:i} for i in range(3)],
                print,hash,
                int,str,float,set,dict,
                (i for i in range(10)),
                enumerate('abc'),
                range(3),
                zip(),
                set.add,
                dict.copy])
```

```
{"<class 'int'>": [0, 1, 2],
 "<class 'str'>": ['a', 'b', 'c'],
 "<class 'float'>": [0.0, 0.5, 1.0],
 "<class 'tuple'>": [(0,), (1,), (2,)],
 "<class 'list'>": [[0], [1], [2]],
 "<class 'set'>": [{0}, {1}, {2}],
 "<class 'dict'>": [{0: 0}, {1: 1}, {2: 2}],
 "<class 'builtin_function_or_method'>": [<function print>,
  <function hash(obj, /)>],
 "<class 'type'>": [int, str, float, set, dict],
 "<class 'generator'>": [<generator object <genexpr> at 0x7fa62d7688b8>],
 "<class 'enumerate'>": [<enumerate at 0x7fa62d672948>],
 "<class 'range'>": [range(0, 3)],
 "<class 'zip'>": [<zip at 0x7fa62d66eb08>],
 "<class 'method_descriptor'>": [<method 'add' of 'set' objects>,
  <method 'copy' of 'dict' objects>]}
```

CHAPTER

TWELVE

MONTE CARLO SIMULATION AND LINEAR ALGEBRA

12.1 Monte Carlo simulation

What is Monte Carlo simulation?

> The name Monte Carlo refers to the Monte Carlo Casino in Monaco where Ulam's uncle would borrow money from relatives to gamble.

It would be nice to simulate the casino, so Ulam's uncle did not need to borrow money to go.Actually…,

- Monte Carlo is the code name of the secret project for creating the hydrogen bomb.
- Ulam worked with John von Neumann to program the first electronic computer ENIAC to simulate a computational model of a thermonuclear reaction.

(See also The Beginning of the Monte Carlo Method for a more detailed account.)

How to compute the value of π?

An application of Monte Carlo simulation is in approximating π using the Buffon's needle.There is a program written in javascript to do this.

The javascript program a bit long to digest, so we will use an alternative simulation that is easier to understand/program.

If we uniformly randomly pick a point in a square. What is the chance it is in the inscribed circle, i.e., the biggest circle inside the square?

The chance is the area of the circle divided by the area of the square. Suppose the square has length ℓ, then the chance is

$$\frac{\pi(\ell/2)^2}{(\ell)^2} = \frac{\pi}{4}$$

independent of the length ℓ.

Exercise Complete the following function to return an approximation of π as follows:

1. Simulate the random process of picking a point from a square repeatedly $\texttt{n}$ times bygenerating the x and y coordinates uniformly randomly from a unit interval $[0, 1)$.
2. Compute the fraction of times the point is in the first quadrant of the inscribed circle as shown in the figure below.
3. Return 4 times the fraction as the approximation.

```
import random, math

def approximate_pi(n):
    ### BEGIN SOLUTION
     return 4*len([True for i in range(n)
```

(continues on next page)

(continued from previous page)

```
                    if random.random()**2 + random.random()**2 < 1])/n
    ### END SOLUTION
print(f'Approximate: {approximate_pi(int(1e7))}\nGround truth: {math.pi}')
```

```
Approximate: 3.1414888
Ground truth: 3.141592653589793
```

How accurate is the approximation?

The following uses a powerful library `numpy` for computing to return a 95%-confidence interval.

```
import numpy as np

def np_approximate_pi(n):
    in_circle = (np.random.random((n,2))**2).sum(axis=-1) < 1
    mean = 4 * in_circle.mean()
    std = 4 * in_circle.std() / n**0.5
    return np.array([mean - 2*std, mean + 2*std])

interval = np_approximate_pi(int(1e7))
print(f'''95%-confidence interval: {interval}
Estimate: {interval.mean():.4f} ± {(interval[1]-interval[0])/2:.4f}
Ground truth: {math.pi}''')
```

```
95%-confidence interval: [3.14097918 3.14305602]
Estimate: 3.1420 ± 0.0010
Ground truth: 3.141592653589793
```

Note that the computation done using `numpy` is over 5 times faster despite the additional computation of the standard deviation.

There are faster methods to approximate π such as the Chudnovsky_algorithm, but Monte-Carlo method is still useful in more complicated situations.E.g., see the Monte Carlo simulation of a real-life situation in playing basketball:

> "When down by three and left with only 30 seconds is it better to attempt a hard 3-point shot or an easy 2-point shot and get another possession?" –LeBron James

12.2 Linear Algebra

How to solve a linear equation?

Given the following linear equation in variable x with real-valued coefficient a and b,

$$ax = b,$$

what is the value of x that satisfies the equation?

Exercise Complete the following function to return either the unique solution of x or `None` if a unique solution does not exist.

```
def solve_linear_equation(a,b):
    ### BEGIN SOLUTION
    return b/a if a != 0 else None
    ### END SOLUTION
```

(continues on next page)

(continued from previous page)

```
import ipywidgets as widgets
@widgets.interact(a=(0,5,1),b=(0,5,1))
def linear_equation_solver(a=2, b=3):
    print(f'''linear equation: {a}x = {b}
       solution: x = {solve_linear_equation(a,b)}''')
```

```
interactive(children=(IntSlider(value=2, description='a', max=5), IntSlider(value=3,␣
↪description='b', max=5), …
```

How to solve multiple linear equations?

In the general case, we have a system of m linear equations and n variables:

$$\begin{aligned} a_{00}x_0 + a_{01}x_1 + \cdots + a_{0(n-1)}x_{n-1} &= b_0 \\ a_{10}x_0 + a_{11}x_1 + \cdots + a_{1(n-1)}x_{n-1} &= b_1 \\ \vdots \qquad\qquad &= \vdots \\ a_{(m-1)0}x_0 + a_{(m-1)1}x_1 + \cdots + a_{(m-1)(n-1)}x_{n-1} &= b_{m-1} \end{aligned}$$

where

- x_j for $j \in \{0, \ldots, n-1\}$ are the variables, and
- a_{ij} and b_j for $i \in \{0, \ldots, m-1\}$ and $j \in \{0, \ldots, n-1\}$ are the coefficients.

A fundamental problem in linear algebra is to compute the unique solution to the system if it exists.

We will consider the simpler 2-by-2 system with 2 variables and 2 equations:

$$\begin{aligned} a_{00}x_0 + a_{01}x_1 &= b_0 \\ a_{10}x_0 + a_{11}x_1 &= b_1. \end{aligned}$$

To get an idea of the solution, suppose

$$a_{00} = a_{11} = 1, a_{01} = a_{10} = 0.$$

The system of equations become

$$\begin{aligned} x_0 \qquad &= b_0 \\ x_1 &= b_1, \end{aligned}$$

which gives the solution directly.

What about $a_{00} = a_{11} = 2$ instead?

$$\begin{aligned} 2x_0 \qquad &= b_0 \\ 2x_1 &= b_1, \end{aligned}$$

To obtain the solution, we simply divide both equations by 2:

$$\begin{aligned} x_0 \qquad &= \frac{b_0}{2} \\ x_1 &= \frac{b_1}{2}. \end{aligned}$$

What if $a_{01} = 2$ instead?

$$\begin{aligned} 2x_0 + 2x_1 &= b_0 \\ 2x_1 &= b_1 \end{aligned}$$

The second equation gives the solution of x_1, and we can use the solution in the first equation to solve for x_0. More precisely:

- Subtract the second equation from the first one:

$$\begin{aligned} 2x_0 &= b_0 - b_1 \\ 2x_1 &= b_1 \end{aligned}$$

- Divide both equation by 2:

$$\begin{aligned} x_0 &= \frac{b_0 - b_1}{2} \\ x_1 &= \frac{b_1}{2} \end{aligned}$$

The above operations are called *row operations* in linear algebra: each row is an equation.A system of linear equations can be solved by the linear operations of

1. multiplying an equation by a constant, and
2. subtracting one equation from another.

How to write a program to solve a general 2-by-2 system? We will use the `numpy` library.

12.2.1 Creating `numpy` arrays

How to store the coefficients?

In linear algebra, a system of equations such as

$$\begin{aligned} a_{00}x_0 + a_{01}x_1 &= b_0 \\ a_{10}x_0 + a_{11}x_1 &= b_1 \end{aligned}$$

is written concisely in *matrix* form as $\mathbf{Ax} = \mathbf{b}$:

$$\overbrace{\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}}^{\mathbf{A}} \overbrace{\begin{bmatrix} x_0 \\ x_1 \end{bmatrix}}^{\mathbf{x}} = \overbrace{\begin{bmatrix} b_0 \\ b_1 \end{bmatrix}}^{\mathbf{b}},$$

where $\mathbf{Ax}$ is the *matrix multiplication*

$$\mathbf{Ax} = \begin{bmatrix} a_{00}x_0 + a_{01}x_1 \\ a_{10}x_0 + a_{11}x_1 \end{bmatrix}.$$

We say that $\mathbf{A}$ is a *matrix* and its dimension/shape is 2-by-2:

- The first dimension/axis has size 2. We also say that the matrix has 2 rows.
- The second dimension/axis has size 2. We also say that the matrix has 2 columns. $\mathbf{x}$ and $\mathbf{b}$ are called column vectors, which are matrices with one column.

Consider the example $$2x_0 + 2x_1 = 1 \\ 2x_1 = 1, \quad or in matrix form with \quad \mathbf{A} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 0 & 2 \end{bmatrix} \\ \mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Instead of using `list` to store the matrix, we will use a `numpy` array.

```
A = np.array([[2.,2],[0,2]])
b = np.array([1.,1])
A, b
```

```
(array([[2., 2.],
        [0., 2.]]),
 array([1., 1.]))
```

Compared to `list`, `numpy` array is often more efficient and has more useful attributes.

```
array_attributes = set(attr for attr in dir(np.array([])) if attr[0]!='_')
list_attributes = set(attr for attr in dir(list) if attr[0]!='_')
print('\nCommon attributes:\n',*(array_attributes & list_attributes))
print('\nArray-specific attributes:\n', *(array_attributes - list_attributes))
print('\nList-specific attributes:\n',*(list_attributes - array_attributes))
```

```
Common attributes:
 copy sort

Array-specific attributes:
 compress nbytes tobytes var shape ravel choose squeeze round all astype newbyteorder␣
↪conjugate tolist imag flags cumprod flat argmax searchsorted mean data view␣
↪setflags cumsum swapaxes ctypes put take argmin std diagonal ndim any dumps␣
↪tostring item conj real dtype dump tofile reshape max argpartition dot argsort␣
↪strides flatten sum nonzero itemset clip size partition base ptp itemsize min prod␣
↪trace byteswap getfield repeat fill T transpose resize setfield

List-specific attributes:
 reverse extend append remove pop count index insert clear
```

The following attributes give the dimension/shape, number of dimensions, size, and datatype.

```
for array in A, b:
    print(f'''{array}
    shape: {array.shape}
    ndim: {array.ndim}
    size: {array.size}
    dtype: {array.dtype}
    ''')
```

```
[[2. 2.]
 [0. 2.]]
    shape: (2, 2)
    ndim: 2
    size: 4
    dtype: float64

[1. 1.]
    shape: (2,)
    ndim: 1
    size: 2
    dtype: float64
```

Note that the function `len` only returns the size of the first dimension:

```
assert A.shape[0] == len(A)
len(A)
```

```
2
```

Unlike `list`, every `numpy` array has a data type. For efficient computation/storage, numpy implements different data types with different storage sizes:

- integer: `int8`, `int16`, `int32`, `int64`, `uint8`, ...
- float: `float16`, `float32`, `float64`, ...
- complex: `complex64`, `complex128`, ...
- boolean: `bool8`
- Unicode: `string`
- Object: `object`

E.g., `int64` is the 64-bit integer. Unlike `int`, `int64` has a range.

```
np.int64?
print(f'range: {np.int64(-2**63)} to {np.int64(2**63-1)}')
np.int64(2**63)    # overflow error
```

```
range: -9223372036854775808 to 9223372036854775807
```

```
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
<ipython-input-9-b5ecbde7f9e9> in <module>
      1 get_ipython().run_line_magic('pinfo', 'np.int64')
      2 print(f'range: {np.int64(-2**63)} to {np.int64(2**63-1)}')
----> 3 np.int64(2**63)    # overflow error

OverflowError: Python int too large to convert to C long
```

We can use the `astype` method to convert the data type:

```
A_int64 = A.astype(int)        # converts to int64 by default
A_float32 = A.astype(np.float32)  # converts to float32
for array in A_int64, A_float32:
    print(array, array.dtype)
```

```
[[2 2]
 [0 2]] int64
[[2. 2.]
 [0. 2.]] float32
```

We have to be careful about assigning items of different types to an array.

```
A_int64[0,0] = 1
print(A_int64)
A_int64[0,0] = 0.5
print(A_int64)                 # intended assignment fails
np.array([int(1), float(1)])   # will be all floating point numbers
```

```
[[1 2]
 [0 2]]
[[0 2]
 [0 2]]
```

```
array([1., 1.])
```

Exercise Create a heterogeneous numpy array to store both integer and strings:

```
[0, 1, 2, 'a', 'b', 'c']
```

Hint: There is an numpy data type called `object`.

```
np.object?
### BEGIN SOLUTION
heterogeneous_np_array = np.array([*range(3),*'abc'],dtype=object)
### END SOLUTION
heterogeneous_np_array
```

```
array([0, 1, 2, 'a', 'b', 'c'], dtype=object)
```

Be careful when creating arrays of `tuple`/`list`:

```
for array in (np.array([(1,2),[3,4,5]],dtype=object),
              np.array([(1,2),[3,4]],dtype=object)):
    print(array, '\nshape:', array.shape, 'length:', len(array), 'size:', array.size)
```

```
[(1, 2) list([3, 4, 5])]
shape: (2,) length: 2 size: 2
[[1 2]
 [3 4]]
shape: (2, 2) length: 2 size: 4
```

`numpy` provides many functions to create an array:

```
np.zeros?
np.zeros(0), np.zeros(1), np.zeros((2,3,4))  # Dimension can be higher than 2
```

```
(array([], dtype=float64),
 array([0.]),
 array([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]]))
```

```
np.ones?
np.ones(0, dtype=int), np.ones((2,3,4), dtype=int)  # initialize values to int 1
```

```
(array([], dtype=int64),
 array([[[1, 1, 1, 1],
         [1, 1, 1, 1],
         [1, 1, 1, 1]],

        [[1, 1, 1, 1],
         [1, 1, 1, 1],
         [1, 1, 1, 1]]]))
```

```
np.eye?
np.eye(0), np.eye(1), np.eye(2), np.eye(3)  # identity matrices
```

```
(array([], shape=(0, 0), dtype=float64),
 array([[1.]]),
 array([[1., 0.],
        [0., 1.]]),
 array([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]]))
```

```
np.diag?
np.diag(range(1)), np.diag(range(2)), np.diag(np.ones(3),k=1)  # diagonal matrices
```

```
(array([[0]]),
 array([[0, 0],
        [0, 1]]),
 array([[0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.],
        [0., 0., 0., 0.]]))
```

```
np.empty?
np.empty(0), np.empty((2,3,4), dtype=int)  # create array faster without␣
↪initialization
```

```
(array([], dtype=float64),
 array([[[ 94852514792576,               0, 140173244071472,
          140173241675144],
         [140173170760144, 140173170760200, 140173231958064,
          140173171104432],
         [140173171105048, 140173171105104, 140173171104824,
          140173214579376]],

        [[140173171109360, 140173170697200, 140173231740104,
          140173170697264],
         [140173242276584, 140173171115248, 140173231954368,
          140173171111232],
         [140173171109680, 140173170696752, 140173170696944,
          140173234814792]]]))
```

`numpy` also provides functions to build an array using rules.

```
np.arange?
np.arange(5), np.arange(4,5), np.arange(4.5,5.5,0.5)  # like range but allow non-
↪integer parameters
```

```
(array([0, 1, 2, 3, 4]), array([4]), array([4.5, 5. ]))
```

```
np.linspace?
np.linspace(4,5), np.linspace(4,5,11), np.linspace(4,5,11)  # can specify number of␣
↪points instead of step
```

```
(array([4.        , 4.02040816, 4.04081633, 4.06122449, 4.08163265,
        4.10204082, 4.12244898, 4.14285714, 4.16326531, 4.18367347,
        4.20408163, 4.2244898 , 4.24489796, 4.26530612, 4.28571429,
        4.30612245, 4.32653061, 4.34693878, 4.36734694, 4.3877551 ,
        4.40816327, 4.42857143, 4.44897959, 4.46938776, 4.48979592,
```

(continues on next page)

(continued from previous page)

```
        4.51020408, 4.53061224, 4.55102041, 4.57142857, 4.59183673,
        4.6122449 , 4.63265306, 4.65306122, 4.67346939, 4.69387755,
        4.71428571, 4.73469388, 4.75510204, 4.7755102 , 4.79591837,
        4.81632653, 4.83673469, 4.85714286, 4.87755102, 4.89795918,
        4.91836735, 4.93877551, 4.95918367, 4.97959184, 5.        ]),
 array([4. , 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5. ]),
 array([4. , 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5. ]))
```

```
np.fromfunction?
np.fromfunction(lambda i, j: i * j, (3,4))  # can initialize using a function
```

```
array([[0., 0., 0., 0.],
       [0., 1., 2., 3.],
       [0., 2., 4., 6.]])
```

We can also reshape an array using the `reshape` method/function:

```
array = np.arange(2*3*4)
array.reshape?
(array,
 array.reshape(2,3,4),      # last axis index changes fastest
 array.reshape(2,3,-1),     # size of last axis calculated automatically
 array.reshape((2,3,4), order='F')) # first axis index changes fastest
```

```
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23]),
 array([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

        [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]]]),
 array([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

        [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]]]),
 array([[[ 0,  6, 12, 18],
         [ 2,  8, 14, 20],
         [ 4, 10, 16, 22]],

        [[ 1,  7, 13, 19],
         [ 3,  9, 15, 21],
         [ 5, 11, 17, 23]]]))
```

`flatten` is a special case of reshaping an array to one dimension.(Indeed, `flatten` returns a copy of the array but `reshape` returns a dynamic view whenever possible.)

```
array = np.arange(2*3*4).reshape(2,3,4)
array, array.flatten(), array.reshape(-1), array.flatten(order='F')
```

```
(array([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

        [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]]]),
 array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23]),
 array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23]),
 array([ 0, 12,  4, 16,  8, 20,  1, 13,  5, 17,  9, 21,  2, 14,  6, 18, 10,
        22,  3, 15,  7, 19, 11, 23]))
```

Exercise Correct the following function to print every element of an array line-by-line.

```
def print_array_entries_line_by_line(array):
    for i in array:
        print(i)
```

```
def print_array_entries_line_by_line(array):
    ### BEGIN SOLUTION
    for i in array.flatten():
        print(i)
    ### END SOLUTION

print_array_entries_line_by_line(np.arange(2*3*4).reshape(2,3,4))
```

12.2.2 Operating on `numpy` arrays

How to verify the solution of a system of linear equations?

Before solving the system of linear equations, let us try to verify a solution to the equations:

$$\begin{aligned} 2x_0 + 2x_1 &= 1 \\ 2x_1 &= 1 \end{aligned}$$

`numpy` provides the function `matmul` and the operator `@` for matrix multiplication.

```
print(np.matmul(A,np.array([0,0])) == b)
print(A @ np.array([0,0.5]) == b)
```

```
[False False]
[ True  True]
```

Note that the comparison on `numpy` arrays returns a boolean array instead of a boolean value, unlike the comparison operations on lists.

To check whether all items are true, we use the `all` method.

```
print((np.matmul(A,np.array([0,0])) == b).all())
print((A @ np.array([0,0.5]) == b).all())
```

```
False
True
```

How to concatenate arrays?

We will operate on an augmented matrix of the coefficients:

$$\begin{aligned}\mathbf{C} &= \begin{bmatrix}\mathbf{A} & \mathbf{b}\end{bmatrix} \\ &= \begin{bmatrix}a_{00} & a_{01} & b_0 \\ a_{10} & a_{11} & b_1\end{bmatrix}\end{aligned}$$

`numpy` provides functions to create block matrices:

```
np.block?
C = np.block([A,b.reshape(-1,1)])  # reshape to ensure same ndim
C
```

```
array([[2., 2., 1.],
       [0., 2., 1.]])
```

To stack an array along different axes:

```
array = np.arange(1*2*3).reshape(1,2,3)
for concat_array in [array,
                     np.hstack((array,array)),   # stack along the first axis
                     np.vstack((array,array)),                    # second axis
                     np.concatenate((array,array), axis=-1),      # last axis
                     np.stack((array,array), axis=0)]:            # new axis
    print(concat_array, '\nshape:', concat_array.shape)
```

```
[[[0 1 2]
  [3 4 5]]]
shape: (1, 2, 3)
[[[0 1 2]
  [3 4 5]
  [0 1 2]
  [3 4 5]]]
shape: (1, 4, 3)
[[[0 1 2]
  [3 4 5]]

 [[0 1 2]
  [3 4 5]]]
shape: (2, 2, 3)
[[[0 1 2 0 1 2]
  [3 4 5 3 4 5]]]
shape: (1, 2, 6)
[[[[0 1 2]
   [3 4 5]]]


 [[[0 1 2]
   [3 4 5]]]]
shape: (2, 1, 2, 3)
```

How to perform arithmetic operations on a `numpy` array?

To divide all the coefficients by 2, we can simply write:

```
D = C / 2
D
```

```
array([[1. , 1. , 0.5],
       [0. , 1. , 0.5]])
```

Note that the above does not work for `list`.

```
C.tolist() / 2 # deep convert to list
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-30-7f30faff9f2c> in <module>
----> 1 C.tolist() / 2 # deep convert to list

TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

Arithmetic operations on `numpy` arrays apply if the arrays have compatible dimensions. Two dimensions are compatible when

- they are equal, except for
- components equal to 1.

`numpy` uses broadcasting rules to stretch the axis of size 1 up to match the corresponding axis in other arrays.`C / 2` is a example where the second operand 2 is broadcasted to a 2-by-2 matrix before the elementwise division. Another example is as follows.

```
three_by_one = np.arange(3).reshape(3,1)
one_by_four = np.arange(4).reshape(1,4)
print(f'''
{three_by_one}
*
{one_by_four}
==
{three_by_one * one_by_four}
''')
```

```
[[0]
 [1]
 [2]]
*
[[0 1 2 3]]
==
[[0 0 0 0]
 [0 1 2 3]
 [0 2 4 6]]
```

Next, to subtract the second row of the coefficients from the first row:

```
D[0,:] = D[0,:] - D[1,:]
D
```

```
array([[1. , 0. , 0. ],
       [0. , 1. , 0.5]])
```

Notice the use of commas to index different dimensions instead of using multiple brackets:

```
assert (D[0][:] == D[0,:]).all()
```

Using this indexing technique, it is easy extract the last column as the solution to the system of linear equations:

```
x = D[:,-1]
x
```

```
array([0. , 0.5])
```

This gives the desired solution $x_0 = 0$ and $x_1 = 0.5$ for

$$2x_0 + 2x_1 = 1$$
$$2x_1 = 1$$

`numpy` provides many convenient ways to index an array.

```
B = np.arange(2*3).reshape(2,3)
B, B[(0,1),(2,0)]  # selecting the corners using integer array
```

```
(array([[0, 1, 2],
        [3, 4, 5]]),
 array([2, 3]))
```

```
B = np.arange(2*3*4).reshape(2,3,4)
B, B[0], B[0,(1,2)], B[0,(1,2),(2,3)], B[:,(1,2),(2,3)]  # pay attention to the last␣
↪two cases
```

```
(array([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

        [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]]]),
 array([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]]),
 array([[ 4,  5,  6,  7],
        [ 8,  9, 10, 11]]),
 array([ 6, 11]),
 array([[ 6, 11],
        [18, 23]]))
```

```
assert (B[...,-1] == B[:,:,-1]).all()
B[...,-1]  # ... expands to selecting all elements of all previous dimensions
```

```
array([[ 3,  7, 11],
       [15, 19, 23]])
```

```
B[B>5]  # indexing using boolean array
```

```
array([ 6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
       23])
```

Finally, the following function solves a system of 2 linear equations with 2 variables.

```
def solve_2_by_2_system(A,b):
    '''Returns the unique solution of the linear system, if exists,
```

(continues on next page)

(continued from previous page)

```
    else returns None.'''
    C = np.hstack((A,b.reshape(-1,1)))
    if C[0,0] == 0: C = C[(1,0),:]
    if C[0,0] == 0: return None
    C[0,:] = C[0,:] / C[0,0]
    C[1,:] = C[1,:] - C[0,:] * C[1,0]
    if C[1,1] == 0: return None
    C[1,:] = C[1,:] / C[1,1]
    C[0,:] = C[0,:] - C[1,:] * C[0,1]
    return C[:,-1]
```

```
# tests
for A in (np.eye(2),
          np.ones((2,2)),
          np.stack((np.ones(2),np.zeros(2))),
          np.stack((np.ones(2),np.zeros(2)),axis=1)):
    print(f'A={A}\nb={b}\nx={solve_2_by_2_system(A,b)}\n')
```

```
A=[[1. 0.]
 [0. 1.]]
b=[1. 1.]
x=[1. 1.]

A=[[1. 1.]
 [1. 1.]]
b=[1. 1.]
x=None

A=[[1. 1.]
 [0. 0.]]
b=[1. 1.]
x=None

A=[[1. 0.]
 [1. 0.]]
b=[1. 1.]
x=None
```

12.2.3 Universal functions

Why does the first line of code below return two arrays but the second code return only one array? Shouldn't the first line of code return the following?

```
array([[(0,1), (0,2), (0,3)],
       [(1,1), (1,2), (1,3)]])
```

```
print(np.fromfunction(lambda i,j:(i,j), (2,3), dtype=int))
print(np.fromfunction(lambda i,j:(i*j), (2,3), dtype=int))
```

```
(array([[0, 0, 0],
       [1, 1, 1]]), array([[0, 1, 2],
       [0, 1, 2]]))
[[0 0 0]
 [0 1 2]]
```

From the documentation, `fromfunction` applies the given function to the two arrays as arguments.

- The first line of code returns a tuple of the arrays.
- The second line of code multiplies the two arrays to give one array, according to how multiplication works for numpy arrays.

Indeed, `numpy` implements universal/vectorized functions/operators that take arrays as arguments and perform operations with appropriate broadcasting rules. The following is an example that uses the universal function `np.sin`:

```
import matplotlib.pyplot as plt

@widgets.interact(a=(0,5,1),b=(-1,1,0.1))
def plot_sin(a=1,b=0):
    x = np.linspace(0,2*math.pi)
    plt.plot(x,np.sin(a*x+b*math.pi))  # np.sin, *, + are universal functions
    plt.title(r'$\sin(ax+b\pi)$')
    plt.xlabel(r'$x$ (radian)')
```

```
interactive(children=(IntSlider(value=1, description='a', max=5), FloatSlider(value=0.
↪0, description='b', max=…
```

In addition to making the code shorter, universal functions are both efficient and flexible. (Recall the Monte Carlo simulation to approximate π.)

Exercise Explain how the Monte Carlo simulation work using universal functions:

```
def np_approximate_pi(n):
    in_circle = (np.random.random((n,2))**2).sum(axis=-1) < 1
    mean = 4 * in_circle.mean()
    std = 4 * in_circle.std() / n**0.5
    return np.array([mean - 2*std, mean + 2*std])
```

- `random.random` generates a numpy array for n points in the unit square randomly.
- `sum` sums up the element along the last axis to give the squared distance.
- `<` returns the boolean array indicating whether each point is in the first quadrant of the inscribed circle.
- `mean` and `std` returns the mean and standard deviation of the boolean array with True and False interpreted as 1 and 0 respectively.

CHAPTER

THIRTEEN

REVIEW QUESTIONS

13.1 Dictionaries and Sets

Exercise (Concatenate two dictionaries with precedence) Define a function `concat_two_dicts` that accepts two arguments of type `dict` such that `concat_two_dicts(a, b)` will return a new dictionary containing all the items in `a` and the items in `b` that have different keys than those in `a`. The input dictionaries should not be mutated.

```
def concat_two_dicts(a, b):
    ### BEGIN SOLUTION
    return {**b, **a}
    ### END SOLUTION
```

```
#tests
a={'x':10, 'z':30}; b={'y':20, 'z':40}
a_copy = a.copy(); b_copy = b.copy()
assert concat_two_dicts(a, b) == {'x': 10, 'z': 30, 'y': 20}
assert concat_two_dicts(b, a) == {'x': 10, 'z': 40, 'y': 20}
assert a == a_copy and b == b_copy
### BEGIN HIDDEN TESTS
a={'x':10, 'z':30}; b={'y':20}
a_copy = a.copy(); b_copy = b.copy()
assert concat_two_dicts(a, b) == {'x': 10, 'z': 30, 'y': 20}
assert concat_two_dicts(b, a) == {'x': 10, 'z': 30, 'y': 20}
assert a == a_copy and b == b_copy
### END HIDDEN TESTS
```

- `{**dict1,**dict2}` creates a new dictionary by unpacking the dictionaries `dict1` and `dict2`.
- By default, `dict2` overwrites `dict1` if they have identical keys.

Exercise (Count characters) Define a function `count_characters` which

- accepts a string and counts the numbers of each character in the string, and
- returns a dictionary that stores the results.

```
def count_characters(string):
    ### BEGIN SOLUTION
    counts = {}
    for char in string:
        counts[char] = counts.get(char, 0) + 1
    return counts
    ### END SOLUTION
```

```
# tests
assert count_characters('abcbabc') == {'a': 2, 'b': 3, 'c': 2}
assert count_characters('aababcccabc') == {'a': 4, 'b': 3, 'c': 4}
### BEGIN HIDDEN TESTS
assert count_characters('abcdefgabc') == {'a': 2, 'b': 2, 'c': 2, 'd': 1, 'e': 1, 'f
↪': 1, 'g': 1}
assert count_characters('ab43cb324abc') == {'2': 1, '3': 2, '4': 2, 'a': 2, 'b': 3, 'c
↪': 2}
### END HIDDEN TESTS
```

- Create an empty dictionary `counts`.
- Use a `for` loop to iterate over each character of `string` to count their numbers of occurrences.
- The `get` method of `dict` can initialize the count of a new character before incrementing it.

Exercise (Count non-Fibonacci numbers) Define a function `count_non_fibs` that

- accepts a container as an argument, and
- returns the number of items in the container that are not fibonacci numbers.

```
def count_non_fibs(container):
    ### BEGIN SOLUTION
    def fib_sequence_inclusive(stop):
        Fn, Fn1 = 0, 1
        while Fn <= stop:
            yield Fn
            Fn, Fn1 = Fn1, Fn + Fn1

    non_fibs = set(container)
    non_fibs.difference_update(fib_sequence_inclusive(max(container)))
    return len(non_fibs)
    ### END SOLUTION
```

```
# tests
assert count_non_fibs([0, 1, 2, 3, 5, 8]) == 0
assert count_non_fibs({13, 144, 99, 76, 1000}) == 3
### BEGIN HIDDEN TESTS
assert count_non_fibs({5, 8, 13, 21, 34, 100}) == 1
assert count_non_fibs({0.1, 0}) == 1
### END HIDDEN TESTS
```

- Create a set of Fibonacci numbers up to the maximum of the items in the container.
- Use `difference_update` method of `set` to create a set of items in the container but not in the set of Fibonacci numbers.

Exercise (Calculate total salaries) Suppose `salary_dict` contains information about the name, salary, and working time about employees in a company. An example of `salary_dict` is as follows:

```
salary_dict = {
     'emp1': {'name': 'John', 'salary': 15000, 'working_time': 20},
     'emp2': {'name': 'Tom', 'salary': 16000, 'working_time': 13},
     'emp3': {'name': 'Jack', 'salary': 15500, 'working_time': 15},
}
```

Define a function `calculate_total` that accepts `salary_dict` as an argument, and returns a `dict` that uses the same keys in `salary_dict` but the total salaries as their values. The total salary of an employee is obtained

by multiplying his/her salary and his/her working_time. E.g.,, for the `salary_dict` example above, `calculate_total(salary_dict)` should return

```
{'emp1': 300000, 'emp2': 208000, 'emp3': 232500}.
```

where the total salary of `emp1` is $15000 \times 20 = 300000$.

```
def calculate_total(salary_dict):
    ### BEGIN SOLUTION
    return {
        emp: record['salary'] * record['working_time']
        for emp, record in salary_dict.items()
    }
    ### END SOLUTION
```

```
# tests
salary_dict = {
     'emp1': {'name': 'John', 'salary': 15000, 'working_time': 20},
     'emp2': {'name': 'Tom', 'salary': 16000, 'working_time': 13},
     'emp3': {'name': 'Jack', 'salary': 15500, 'working_time': 15},
}
assert calculate_total(salary_dict) == {'emp1': 300000, 'emp2': 208000, 'emp3':␣
↪232500}
### BEGIN HIDDEN TESTS
salary_dict = {
     'emp1': {'name': 'John', 'salary': 15000, 'working_time': 20},
     'emp2': {'name': 'Tom', 'salary': 16000, 'working_time': 13},
     'emp3': {'name': 'Jack', 'salary': 15500, 'working_time': 15},
     'emp4': {'name': 'Bob', 'salary': 20000, 'working_time': 10}
}
assert calculate_total(salary_dict) == {'emp1': 300000, 'emp2': 208000, 'emp3':␣
↪232500, 'emp4': 200000}
### END HIDDEN TESTS
```

- Use `items` method of `dict` to return the list of key values pairs, and
- use a dictionary comprehension to create the desired dictionary by iterating through the list of items.

Exercise (Delete items with value 0 in dictionary) Define a function `zeros_removed` that

- takes a dictionary as an argument,
- mutates the dictionary to remove all the keys associated with values equal to `0`,
- and return `True` if at least one key is removed else `False`.

```
def zeros_removed(d):
    ### BEGIN SOLUTION
    to_delete = [k for k in d if d[k] == 0]
    for k in to_delete:
        del d[k]
    return len(to_delete) > 0


## Memory-efficient but not computationally efficient
# def zeros_removed(d):
#     has_deleted = False
#     while True:
#         for k in d:
```

(continues on next page)

(continued from previous page)

```
#             if d[k] == 0:
#                 del d[k]
#                 has_deleted = True
#                 break
#         else: return has_deleted

### END SOLUTION
```

```
# tests
d = {'a':0, 'b':1, 'c':0, 'd':2}
assert zeros_removed(d) == True
assert zeros_removed(d) == False
assert d == {'b': 1, 'd': 2}
### BEGIN HIDDEN TESTS
d = {'a':0, 'b':1, 'c':0, 'd':2, 'e':0, 'f':'0'}
assert zeros_removed(d) == True
assert zeros_removed(d) == False
assert d == {'b': 1, 'd': 2, 'f':'0'}
### END HIDDEN TESTS
```

- The main issue is that, for any dicionary `d`,

```
    for k in d:
        if d[k] == 0: del d[k]
```

raises the `RuntimeError: dictionary changed size during iteration`.

- One solution is to duplicate the list of keys, but this is memory inefficient especially when the list of keys is large.
- Another solution is to record the list of keys to delete before the actual deletion. This is memory efficient if the list of keys to delete is small.

Exercise (Fuzzy search a set) Define a function `search_fuzzy` that accepts two arguments `myset` and `word` such that

- `myset` is a `set` of `str`s;
- `word` is a `str`; and
- `search_fuzzy(myset, word)` returns `True` if `word` is in `myset` by changing at most one character in `word`, and returns `False` otherwise.

```
def search_fuzzy(myset, word):
    ### BEGIN SOLUTION
    for myword in myset:
        if len(myword) == len(word) and len(
            [True
             for mychar, char in zip(myword, word) if mychar != char]) <= 1:
            return True
    return False
    ### END SOLUTION
```

```
# tests
assert search_fuzzy({'cat', 'dog'}, 'car') == True
assert search_fuzzy({'cat', 'dog'}, 'fox') == False
### BEGIN HIDDEN TESTS
myset = {'cat', 'dog', 'dolphin', 'rabbit', 'monkey', 'tiger'}
assert search_fuzzy(myset, 'lion') == False
```

(continues on next page)

(continued from previous page)

```
assert search_fuzzy(myset, 'cat') == True
assert search_fuzzy(myset, 'cat ') == False
assert search_fuzzy(myset, 'fox') == False
assert search_fuzzy(myset, 'ccc') == False
### END HIDDEN TESTS
```

- Iterate over each word in `myset`.
- Check whether the length of the word is the same as that of the word in the arguments.
- If the above check passes, use a list comprehension check if the words differ by at most one character.

Exercise (Get keys by value) Define a function `get_keys_by_value` that accepts two arguments `d` and `value` where `d` is a dictionary, and returns a set containing all the keys in `d` that have `value` as its value. If no key has the query value `value`, then return an empty set.

```
def get_keys_by_value(d, value):
    ### BEGIN SOLUTION
    return {k for k in d if d[k] == value}
    ### END SOLUTION
```

```
# tests
d = {'Tom':'99', 'John':'88', 'Lucy':'100', 'Lily':'90', 'Jason':'89', 'Jack':'100'}
assert get_keys_by_value(d, '99') == {'Tom'}
### BEGIN HIDDEN TESTS
d = {'Tom':'99', 'John':'88', 'Lucy':'100', 'Lily':'90', 'Jason':'89', 'Jack':'100'}
assert get_keys_by_value(d, '100') == {'Jack', 'Lucy'}
d = {'Tom':'99', 'John':'88', 'Lucy':'100', 'Lily':'90', 'Jason':'89', 'Jack':'100'}
assert get_keys_by_value(d, '0') == set()
### END HIDDEN TESTS
```

- Use set comprehension to create the set of keys whose associated values is `value`.

Exercise (Count letters and digits) Define a function `count_letters_and_digits` which

- take a string as an argument,
- returns a dictionary that stores the number of letters and digits in the string using the keys 'LETTERS' and 'DIGITS' respectively.

```
def count_letters_and_digits(string):
    ### BEGIN SOLUTION
    check = {'LETTERS': str.isalpha, 'DIGITS': str.isdigit}
    counts = dict.fromkeys(check.keys(), 0)
    for char in string:
        for t in check:
            if check[t](char):
                counts[t] += 1
    return counts
    ### END SOLUTION
```

```
assert count_letters_and_digits('hello world! 2020') == {'DIGITS': 4, 'LETTERS': 10}
assert count_letters_and_digits('I love CS1302') == {'DIGITS': 4, 'LETTERS': 7}
### BEGIN HIDDEN TESTS
assert count_letters_and_digits('Hi CityU see you in 2021') == {'DIGITS': 4, 'LETTERS
↪': 15}
assert count_letters_and_digits('When a dog runs at you, whistle for him.␣
↪(Philosopher Henry David Thoreau, 1817-1862)') == {'DIGITS': 8, 'LETTERS': 58}
```

(continues on next page)

(continued from previous page)

```
### END HIDDEN TESTS
```

- Use the class method `fromkeys` of `dict` to initial the dictionary of counts.

Exercise (Dealers with lowest price) Suppose `apple_price` is a list in which each element is a `dict` recording the dealer and the corresponding price, e.g.,

```
apple_price = [{'dealer': 'dealer_A', 'price': 6799},
 {'dealer': 'dealer_B', 'price': 6749},
 {'dealer': 'dealer_C', 'price': 6798},
 {'dealer': 'dealer_D', 'price': 6749}]
```

Define a function `dealers_with_lowest_price` that takes `apple_price` as an argument, and returns the `set` of dealers providing the lowest price.

```
def dealers_with_lowest_price(apple_price):
    ### BEGIN SOLUTION
    dealers = {}
    lowest_price = None
    for pricing in apple_price:
        if lowest_price == None or lowest_price > pricing['price']:
            lowest_price = pricing['price']
        dealers.setdefault(pricing['price'], set()).add(pricing['dealer'])
    return dealers[lowest_price]

## Shorter code that uses comprehension
# def dealers_with_lowest_price(apple_price):
#     lowest_price = min(pricing['price'] for pricing in apple_price)
#     return set(pricing['dealer'] for pricing in apple_price
#                if pricing['price'] == lowest_price)
    ### END SOLUTION
```

```
# tests
apple_price = [{'dealer': 'dealer_A', 'price': 6799},
 {'dealer': 'dealer_B', 'price': 6749},
 {'dealer': 'dealer_C', 'price': 6798},
 {'dealer': 'dealer_D', 'price': 6749}]
assert dealers_with_lowest_price(apple_price) == {'dealer_B', 'dealer_D'}
### BEGIN HIDDEN TESTS
apple_price = [{'dealer': 'dealer_A', 'price': 6799},
 {'dealer': 'dealer_B', 'price': 6799},
 {'dealer': 'dealer_C', 'price': 6799},
 {'dealer': 'dealer_D', 'price': 6799}]
assert dealers_with_lowest_price(apple_price) == {'dealer_A', 'dealer_B', 'dealer_C',
↪'dealer_D'}
### END HIDDEN TESTS
```

- Use the class method `setdefault` of `dict` to create a dictionary that maps different prices to different sets of dealers.
- Compute the lowest price at the same time.
- Alternatively, use comprehension to find lowest price and then create the desired set of dealers with the lowest price.

13.2 Lists and Tuples

Exercise (Binary addition) Define a function `add_binary` that

- accepts two arguments of type `str` which represent two non-negative binary numbers, and
- returns the binary number in `str` equal to the sum of the two given binary numbers.

```
def add_binary(*binaries):
    ### BEGIN SOLUTION
    def binary_to_decimal(binary):
        return sum(2**i * int(b) for i, b in enumerate(reversed(binary)))

    def decimal_to_binary(decimal):
        return ((decimal_to_binary(decimal // 2) if decimal > 1 else '') +
                str(decimal % 2)) if decimal else '0'

    return decimal_to_binary(sum(binary_to_decimal(binary) for binary in binaries))

## Alternative 1 using recursion
# def add_binary(bin1, bin2, carry=False):
#     if len(bin1) > len(bin2):
#         return add_binary(bin2, bin1)
#     if bin1 == '':
#         return add_binary('1', bin2, False) if carry else bin2
#     s = int(bin1[-1]) + int(bin2[-1]) + carry
#     return add_binary(bin1[:-1], bin2[:-1], s > 1) + str(s % 2)

## Alternatve 2 using iteration
# def add_binary(a, b):
#     answer = []
#     n = max(len(a), len(b))
#     # fill necessary '0' to the beginning to make a and b have the same length
#     if len(a) < n: a = str('0' * (n -len(a))) + a
#     if len(b) < n: b = str('0' * (n -len(b))) + b
#     carry = 0
#     for i in range(n-1, -1, -1):
#         if a[i] == '1': carry += 1
#         if b[i] == '1': carry += 1
#         answer.insert(0, '1') if carry % 2 == 1 else answer.insert(0, '0')
#         carry //= 2
#     if carry == 1: answer.insert(0, '1')
#     answer_str = ''.join(answer) # you can also use "answer_str = '';  for x in␣
↪answer: answer_str += x"
#     return answerastr

    ### END SOLUTION
```

```
# tests
assert add_binary('0', '0') == '0'
assert add_binary('11', '11')  == '110'
assert add_binary('101', '101')  == '1010'
### BEGIN HIDDEN TESTS
assert add_binary('1111', '10')  == '10001'
assert add_binary('111110000011','110000111')  == '1000100001010'
### END HIDDEN TESTS
```

- Use comprehension to convert the binary numbers to decimal numbers.

- Use comprehension to convert the sum of the decimal numbers to a binary number.
- Alternatively, perform bitwise addition using a recursion or iteration.

Exercise (Even-digit numbers) Define a function `even_digit_numbers`, which finds all numbers between `lower_bound` and `upper_bound` such that each digit of the number is an even number. Please return the numbers as a list.

```
def even_digit_numbers(lower_bound, upper_bound):
    ### BEGIN SOLUTION
    return [
        x for x in range(lower_bound, upper_bound)
        if not any(int(d) % 2 for d in str(x))
    ]
    ### END SOLUTION
```

```
# tests
assert even_digit_numbers(1999, 2001) == [2000]
assert even_digit_numbers(2805, 2821) == [2806,2808,2820]
### BEGIN HIDDEN TESTS
assert even_digit_numbers(1999, 2300) == [2000,2002,2004,2006,2008,2020,2022,2024,
↪2026,2028,2040,2042,2044,2046,2048,2060,2062,2064,2066,2068,2080,2082,2084,2086,
↪2088,2200,2202,2204,2206,2208,2220,2222,2224,2226,2228,2240,2242,2244,2246,2248,
↪2260,2262,2264,2266,2268,2280,2282,2284,2286,2288]
assert even_digit_numbers(8801, 8833) == [8802,8804,8806,8808,8820,8822,8824,8826,
↪8828]
assert even_digit_numbers(3662, 4001) == [4000]
### END HIDDEN TESTS
```

- Use list comprehension to generate numbers between the bounds, and
- use comprehension and the `any` function to filter out those numbers containing odd digits.

Exercise (Maximum subsequence sum) Define a function `max_subsequence_sum` that

- accepts as an argument a sequence of numbers, and
- returns the maximum sum over nonempty contiguous subsequences.

E.g., when `[-6, -4, 4, 1, -2, 2]` is given as the argument, the function returns `5` because the nonempty subsequence `[4, 1]` has the maximum sum `5`.

```
def max_subsequence_sum(a):
    ### BEGIN SOLUTION
    ## see https://en.wikipedia.org/wiki/Maximum_subarray_problem
    t = s = 0
    for x in a:
        t = max(0, t + x)
        s = max(s, t)
    return s

## Alternative (less efficient) solution using list comprehension
# def max_subsequence_sum(a):
#     return max(sum(a[i:j]) for i in range(len(a)) for j in range(i,len(a)+1))

    ### END SOLUTION
```

```
# tests
assert max_subsequence_sum([-6, -4, 4, 1, -2, 2]) == 5
```

(continues on next page)

(continued from previous page)

```
assert max_subsequence_sum([2.5, 1.4, -2.5, 1.4, 1.5, 1.6]) == 5.9
### BEGIN HIDDEN TESTS
seq = [-24.81, 25.74, 37.29, -8.77, 0.78, -15.33, 30.21, 34.94, -40.64, -20.06]
assert round(max_subsequence_sum(seq),2) == 104.86
### BEGIN HIDDEN TESTS
```

```
# test of efficiency
assert max_subsequence_sum([*range(1234567)]) == 762077221461
```

- For a list $[a_0, a_1, \dots]$, let

$$t_k := \max_{j<k} \sum_{i=j}^{k-1} a_i = \max\{t_{k-1} + a_{k-1}, 0\},$$

namely the maximum tail sum of $[a_0, \dots, a_{k-1}]$.

- Then, the maximum subsequence sum of $[a_0, \dots, a_{k-1}]$ is

$$s_k := \max_{j \le k} t_j.$$

Exercise (Mergesort) *For this question, do not use the* `sort` *method or* `sorted` *function.*

Define a function called `merge` that

- takes two sequences sorted in ascending orders, and
- returns a sorted list of items from the two sequences.

Then, define a function called `mergesort` that

- takes a sequence, and
- return a list of items from the sequence sorted in ascending order.

The list should be constructed by

- recursive calls to `mergesort` the first and second halves of the sequence individually, and
- merge the sorted halves.

```
def merge(left,right):
    ### BEGIN SOLUTION
    if left and right:
        if left[-1] > right[-1]: left, right = right, left
        return merge(left,right[:-1]) + [right[-1]]
    return list(left or right)
    ### END SOLUTION
```

```
def mergesort(seq):
    ### BEGIN SOLUTION
    if len(seq) <= 1:
        return list(seq)
    i = len(seq)//2
    return merge(mergesort(seq[:i]),mergesort(seq[i:]))
    ### END SOLUTION
```

```
# tests
assert merge([1,3],[2,4]) == [1,2,3,4]
assert mergesort([3,2,1]) == [1,2,3]
```

(continues on next page)

(continued from previous page)

```
### BEGIN HIDDEN TESTS
assert mergesort([3,5,2,4,2,1]) == [1,2,2,3,4,5]
### END HIDDEN TESTS
```

13.3 More Functions

Exercise (Arithmetic geometric mean) Define a function `arithmetic_geometric_mean_sequence` which

- takes two floating point numbers `x` and `y` and
- returns a generator that generates the tuple \((a_n, g_n)\) where

$$
\begin{aligned}
a_0 &= x, g_0 = y \\
a_n &= \frac{a_{n-1} + g_{n-1}}{2} \quad \text{for } n > 0 \\
g_n &= \sqrt{a_{n-1} g_{n-1}}
\end{aligned}
$$

```
def arithmetic_geometric_mean_sequence(x, y):
    ### BEGIN SOLUTION
    a, g = x, y
    while True:
        yield a, g
        a, g = (a + g)/2, (a*g)**0.5
    ### END SOLUTION
```

```
# tests
agm = arithmetic_geometric_mean_sequence(6,24)
assert [next(agm) for i in range(2)] == [(6, 24), (15.0, 12.0)]
### BEGIN HIDDEN TESTS
agm = arithmetic_geometric_mean_sequence(100,400)
for sol, ans in zip([next(agm) for i in range(5)], [(100, 400), (250.0, 200.0), (225.
↪0, 223.60679774997897), (224.30339887498948, 224.30231718318308), (224.
↪30285802908628, 224.30285802843423)]):
    for a, b in zip(sol,ans):
        assert round(a,5) == round(b,5)
### END HIDDEN TESTS
```

- Use the `yield` expression to return each tuple of (a_n, g_n) efficiently without redundant computations.