Chapter 1 A gentle introduction to Python

Python is a popular high-level programming language in scientific computation. This chapter is written for readers who have little or no previous experience with Python - so that they can familiarize themselves with the basic syntax and building blocks of the language. Experienced Python users can skip this chapter and move on to the chapters focusing on experimental task scripting, eye-tracking, and eye movement data analysis and visualization. This chapter will only cover the features of Python that one needs to implement simple psychological experiments. For a complete introduction to Python, I would recommend the tutorial on the official Python website.[1]

Install Python

When it comes to Python, there are quite a few distributions (e.g., Anaconda, Canopy), which bundle features and packages that are not part of the official CPython distribution. It would not be productive to describe all the unique features of these Python distributions, and I have no intention to favor one over the others. This book will feature the official CPython distribution that is freely available from python.org. The example scripts presented in this book are based on Python 3, but should also work in Python 2 with little change.

macOS and the various Linux distributions (e.g., Ubuntu) all have Python 2 preinstalled. To check if you have Python 3 on your machine, launch a terminal type *python3* at the command line prompt. As shown in the shell output below, my MacBook has Python 3.6.6. If you do not have Python 3 on your Mac, please download the Mac OS X installer from python.org and install it.

```
Zhiguos-MacBook-Pro:~ zhiguo$ python3.6
```

[1] https://docs.python.org/3.6/tutorial

```
Python 3.6.6 (v3.6.6:4cf1f54eb7, Jun 26 2018, 19:50:54)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you use Ubuntu or another Debian-based Linux distribution, please run the following command in the shell to install Python 3.6.

```
sudo apt-get install python3.6
```

To install Python 3.6 or later on a Windows PC, please download the relevant installer and tick the "Add Python 3.x to PATH" option. One may also customize the installation to enable the "Install for all users" option (requires administrator privilege).

Figure 1-1 *Python install options on Windows.*

Python modules

Thousands of Python modules are actively developed and maintained by the community. These modules greatly expanded the functionality of Python; they allow users to access solutions others have already created for common

problems, as well as to contribute their solutions to the shared pool. The preferred installer for Python modules is *pip*, which has been part of the official CPython distribution since version 2.7.9. With *pip*, users can search, download, and install Python modules from either a public (e.g., Python Packaging Index, or PyPI) or private online repository. To install a module, type the following command at a command-line prompt.[2]

```
pip install module_name
```

The *pip* command will install the requested module, together with all its dependencies from the PyPI repository. On macOS and Linux, one may need to add *sudo* to the above command to obtain system administration privilege.

```
Zhiguos-MacBook-Pro:~ zhiguo$ sudo pip install pygame
Collecting pygame
  Downloading
https://files.pythonhosted.org/packages/dc/cd/dd9225d2eaef6f0d0131ae228d033a
1fabcc6537d9ecfa78786394277993/pygame-1.9.6-cp36-cp36m-macosx_10_11_intel.wh
l (4.9MB)
     |████████████████████████████████| 4.9MB 1.4MB/s
Installing collected packages: pygame
Successfully installed pygame-1.9.6
```

Python shell

Python is an interpreted language, which means that users do not need to deal with tasks like source code linking and compiling (like in C/C++). In other words, you do not need to compile the source code to generate executable files (such as the Windows Notepad program), but rather the source code is evaluated line by line by an interpreter, which converts the source code into instructions recognizable by the computer hardware. In the simplest case, you send a statement (command) to the Python interpreter; the Python interpreter evaluates the statement and sends back its response. It is as simple as using a calculator. When being used interactively in a Python *shell*,[3] the Python

[2] On Windows, search for "CMD" (without quotes) to launch the Command Prompt.

[3] A shell is a user interface for accessing the computer operating system services. The use of a shell usually involves entering a single command and get responses from the operating system (printed in the shell as

interpreter allows you to experiment with the various features of the language or to test functions during program development.

On macOS and Linux, entering *python3.6* into at a command-line prompt will evoke the Python interpreter. Then, the Python interpreter will first print a welcome message, stating with its version number and a copyright notice, before showing the prompt (">>>"—three angle brackets), which suggests that the Python interpreter is ready to accept commands.

```
Zhiguos-MacBook-Pro:~ zhiguo$ python3.6
Python 3.6.6 (v3.6.6:4cf1f54eb7, Jun 26 2018, 19:50:54)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To quit the Python interpreter, press Ctrl-D or type the following command.

```
>>> quit()
```

On Windows, a Python shell (see Figure 1-2) is accessible from the Start menu. One can also open a Windows Command Prompt to enter and quit the Python interpreter with the above commands.

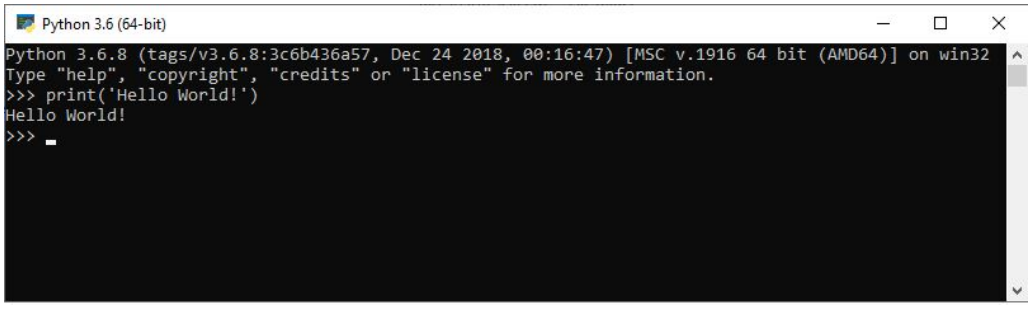


Figure 1-2 *A Python shell is included in the Windows version of the CPython distribution; it is accessible from the Start menu.*

messages). The Python shell works in the same way, but processes Python commands instead of system commands.

As noted, you can use the Python interpreter in an interactive manner: you enter a command or a statement; the interpreter will print the return value.

```
>>> 3 + 4
7
```

Typing *help* in the Python shell will show a notice that you can access the help information with either the *help()* function or the interactive **help** mode.

```
>>> help
Type help() for interactive help, or help(object) for help about object.
>>>
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Type the *help()* function without any argument will enter the interactive help utility. As noted in the welcome messages, type *quit* will terminate the help utility. To get a list of available Python modules, type *modules* (as shown below).

```
>>> help()

Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.6/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contains a given string such as "spam", type "modules spam".

help> modules
```

```
Please wait a moment while I gather a list of all available modules...

AppKit                atexit                ggplot                pyzotero
Cocoa                 audioop               glob                  queue
CoreFoundation        base64                greenlet              quopri
Foundation            bdb                   grp                   random
....

help> quit
```

Script editors

The Python shell is useful for testing out basic features and functions, but real-world applications usually involve many lines of code. It is impractical to type them into the Python shell line-by-line, so many lines of code are usually placed in a Python script, i.e., a text file that ends with a "*.py*" extension. By sending a script to the Python interpreter, one can implement complicated tasks (e.g., a computer-based reaction time task).

You can use any plain text editors, Vim, Emacs, NotePad++, etc., to write your Python script. There are feature-rich development tools that offer not only a script editor but also features like code debugging (e.g., PyCharm, Spyder, Visual Studio Code).[4] These tools are known as Integrated Development Environments, or IDEs for short, and are indispensable to large-scale software development. For writing a short script measuring reaction times to geometric shapes presented on the screen, a full-fledged IDE is not necessary. Which editor or IDE to use is very much a personal preference, but a good Python editor should at least be Python-aware, i.e., should have functions like syntax highlighting and auto-completion.[5]

IDLE is a pure Python IDE. It is part of the official CPython distribution, featuring a Python shell and a script editor. By default, launching IDLE will bring up the shell; from File -> New, you can then open the script editor. We need to save the lines of code that we put into the editor before we can pass it to

[4] See the Python Wiki for a list of Python IDEs, https://wiki.python.org/moin/IntegratedDevelopmentEnvironments.

[5] Auto-completion can greatly improve coding efficiency. For instance, type in "math.lo" and then press the Tab key in the IDLE shell will list all math functions with names starting with "lo", e.g., log, log2, log10.

the Python interpreter. The saved file should have a "*.py*" extension. In the script editor, click Run-> Run module (or press F5) will run the script from beginning to end.

In the short script shown in the screenshot below, we first import the *pygame* library, then we use the *display*, *draw*, and *time* modules of *pygame* to present a red circle in a window. There is no need to delve into the details of this script at this point, but it is worth noting that the strings following the "#" (hash) are comments that help to improve code readability.

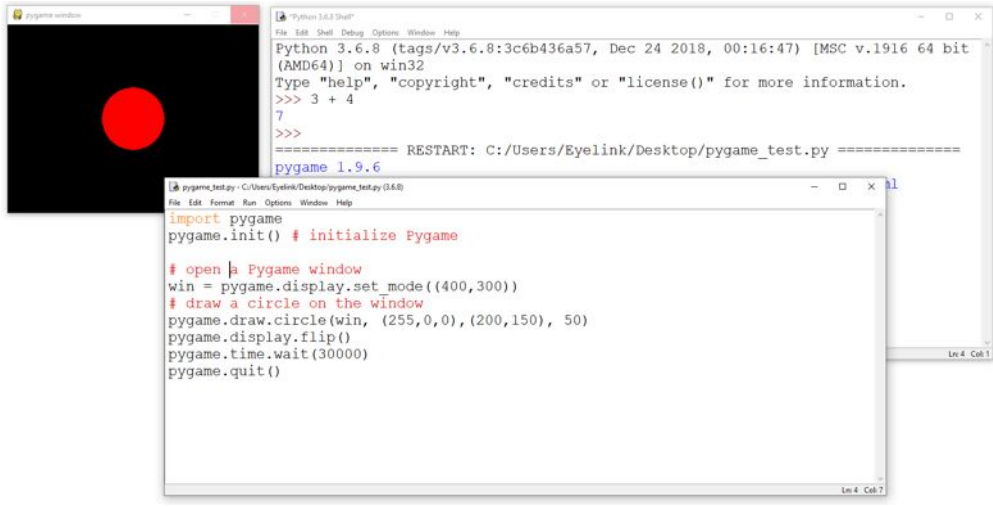


Figure 1-3 *IDLE: a simple Python IDE with a shell and a script editor. The script shown here will open a small window and then draw a red disk.*

Data types

Programming, in essence, is data manipulation. In Python, frequently used data types include numbers, strings, lists, tuples, dictionaries, and Booleans.

Numbers

Numbers can be integers, floats, fractions, or complex numbers. Integers (e.g., 2, 4, 20) are of type *int*, whereas numbers with a fractional part (e.g., 5.0, 1.6) are of type *float*. The mathematical operations which can be performed on

numbers include + (addition), - (subtraction), * (multiplication), and / (division), etc. (see the Operators section for details). In Python 3, division always returns a float number, whereas in Python 2, dividing an integer by another will return the result of a floor division (e.g., 5/2=2, -5/2=-3). In Python 3, floor division is done with the operator "//" only. The modulus operator, which performs remainder division on integers, is "%" (e.g., 5%2 will return the remainder 1). The power operator is "**" (two asterisks; e.g., 2**3 = 8). For rational number arithmetic, one needs to load the "*fractions*" module.

Strings

Besides numbers, Python can also manipulate strings or text. Strings are expressed in single quotes ('...') or double quotes ("...").

```
>>> 'this is a string'
'this is a string'
>>> "this is a string"
'this is a string'
>>> print("this is a string")
this is a string
>>> print("\" can be used in a string")
" can be used in a string
```

The *print*() function helps to produce a more readable output in the shell, by omitting the enclosing quotes. Note that the above commands also print out an escaped character ("). "Escaping" is an operation necessary for many (if not all) programming languages to deal with special characters (e.g., "\n" represents a new line) and characters reserved for syntax purposes. Consider the above shell command, which printed out a double quote in a string. Because the Python syntax dictates that double quotes mark the start or end of a string, we need to let the Python interpreter know that this double quote is just an ordinary double quote in a string by using the Python escape character "\" (backslash). For instance, "\"" escapes a double quote and "\\" escapes a backslash.

```
>>> print("\"")
"
>>> print("\\")
\
```

```
>>> print("this is a tab\t see it?\nstart a new line")
this is a tab         see it?
start a new line
```

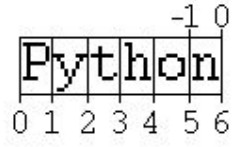
Strings that contain special formatting characters also require the escape character, for instance, tab ("\t") and new-line ("\n") characters. If you add an "r" before the first quote helps to create a *raw* string, the Python interpreter will treat "\" as a regular character instead of the escaping operator.

```
>>> print(r"this is a tab\t see it?\nstart a new line")
this is a tab\t see it?\nstart a new line
```

One feature of strings is that they can be concatenated with the + operator and repeated with the * operator. In the shell example below, the string "Python" is repeated three times and then gets "glued together" with another string "123".

```
>>> 'Python' * 3 + '123'
'PythonPythonPython123'
```

The characters in a string are indexed (subscripted), with the first character being indexed by 0, the second character is indexed by 1, and so on. The indices start from -1 instead when going in the right-to-left direction. One can use an index in a pair of square brackets to retrieve a character, or a range of indices to extract multiple characters (i.e., string slicing).




```
>>> s = "Python"
>>> s[0]
'P'
>>> s[-2]
'o'
>>> s[0:3]
'Pyt'
```

In the shell expressions shown above, the index range [0:3] will retrieve the first three characters from the string "Python"—"Pyt". Wait, this command does not extract the fourth character, although its index (3) is in the brackets? It is best to think of an index as an invisible marker that represents the element right to it. In the illustration above, [0:6] returns all characters up to index 6. When going in the right-to-left direction, there is no character right to index 0, so the first character we can retrieve ("n") has an index of -1.

List

A list is a sequence of comma-separated values of the same or different data types. One can access the items of a list through indexing or slicing, just like strings.

```
>>> a = [1, 2, 3, 4]
>>> b = ['a', 1, [1, 2], 3.0]
>>> b[2]
[1, 2]
>>> b[-1]
3.0
>>> a + b
[1, 2, 3, 4, 'a', 1, [1, 2], 3.0]
```

As with strings, lists can be concatenated with the "+" operator (see the example above) and repeated with the "*" operator. Lists are mutable. That is, you can change a list, such as adding items to the end with *append*(), or sorting the items with *sort*(), etc. To show the properties and methods available for lists, call the *dir*() function. As shown in the shell output below, other operations that can be performed on a list include *count*(), *extend*(), *insert*(), *pop*(), *remove*(), etc.

```
>>> dir(a)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getslice__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__',
'__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
```

```
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__setslice__', '__sizeof__', '__str__', '__subclasshook__', 'append',
'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

It is unnecessary to cover all the list methods here; you can use the *help*() function to show the help information of a list property or method. For instance, the help information below clearly states that *remove*() will delete the first occurrence of a value in a list.

```
>>> help(a.remove)
Help on built-in function remove:

remove(...) method of builtins.list instance
    L.remove(value) -> None -- remove first occurrence of value.
    Raises ValueError if the value is not present.
```

Tuple

A tuple consists of some values separated by commas. As with lists, tuples can contain a heterogeneous sequence of elements, e.g., numbers, strings, lists, and even tuples. The items of a tuple are usually enclosed in a pair of parentheses, but we can omit the parentheses when passing a tuple to a variable. Unlike lists, tuples are immutable. You cannot remove or replace the elements of a tuple.

```
>>> t = 1, 'a', 3.567, [1,2,3]
>>> t
(1, 'a', 3.567, [1, 2, 3])
```

For an empty tuple or a tuple with a single item, things can appear a bit confusing. An empty tuple can be declared with a pair of parentheses ();a tuple with a single element can be declared with the item, followed by a comma.

```
>>> t = 0.4,
>>> t
(0.4,)
```

The comma is necessary, so the Python interpreter will not mistake the parenthesis as operants.

```
>>> t = ('a')
>>> t
'a'
```

One can use tuples to store data that you do not want to change while the script is running; for instance, the resolution of a monitor.

Sets

A set contains unique (non-duplicating), immutable, and unsorted values in a pair of curly brackets; it conforms to the mathematical definition of a set. One can use the *set*() function to declare an empty set.

```
>>> {'a', 'a', 'a'}
{'a'}
>>> ('a', 'a', 'a')
('a', 'a', 'a')
```

Contrary to tuples and lists, items in a set are all unique. Taking advantage of this feature, one can quickly remove duplicates in a list by converting it into a set and back into a list.

```
>>> x = [1,2,3,4,1,2,4,5]
>>> list(set(x))
[1, 2, 3, 4, 5]
```

There are a few other handy operations for sets. For instance, *pop*() will randomly remove an item from a set and return it (random selection without replacement); the *union*() operation combines two sets, and the *intersection*() operation returns the items common to both sets.

```
>>> x = {1,2,3,4,5,6,7}
>>> x.pop()
1
>>> x
{2, 3, 4, 5, 6, 7}
>>> y = {'a','b','c',2,3}
>>> x.union(y)
```

```
{'a', 2, 3, 4, 5, 6, 7, 'c', 'b'}
>>> x.intersection(y)
{2, 3}
>>> x.difference(y)
{4, 5, 6, 7}
```

Dictionary

Dictionaries are sometimes referred to as “associative memories”, a concept borrowed from cognitive psychology. A dictionary is a set of key-value pairs, with the constraint that the keys are unique (within one dictionary) and immutable. A dictionary contains one or more key-value pairs with the key and value separated by a colon; an empty dictionary can be instantiated with a pair of curly brackets. The keys can be used to retrieve the associated value from the dictionary. For instance, in the shell command below d['A'] returns the value of the element associated with the key ‘A’ which is 65. To add a new key-value pair to an existing dictionary, simply specify the new key in a pair of brackets and pass a value to it, for instance, *d['D'] = 68*.

```
>>> d = {'A':65, 'B':66, 'C':67}
>>> d['A']
65
>>> d['D'] = 68
>>> d
{'A': 65, 'B': 66, 'C': 67, 'D': 68}
```

You can use the *list*() function to extract the keys of a dictionary. This function returns the keys in the order that the *key-value* pairs were added to the dictionary. To sort the keys, call *sorted*(). To check whether a single key is in the dictionary, use the *in* operation.

```
>>> list(d)
['A', 'B', 'C', 'D']
>>> 'C' in d
True
```

Booleans

Booleans are either *True* or *False*. They are the return values for comparisons and Boolean operations. For instance, the statement “*3 < 5*” will return *True*;

the statement "*True and False*" will return False, while "*True and True*" will return *True*. Booleans are frequently used in control flow statements, for instance, in an *if* statement (see the following sections for details).

Booleans can be interpreted as numbers, with *True* = 1 and *False* = 0. So, "*True + True*" will return 2. You can also perform Boolean operations on other Python data types. With numbers, 0 is *False,* and all non-zero numbers are *True*; with lists, tuples, sets, and dictionaries, empty ones are *False,* and non-empty ones are *True*. Python also has a constant *None* for a null value; comparing *None* to any other data type will always return *False*.

Operators

Operators are reserved keywords or characters that one can use to perform operations on values and variables. As has been discussed in the Numbers section, the arithmetic operators supported in Python include addition (+), subtraction (-), multiplication (*), division (/), floor division (//), modulus (%), and power (**).

For logical operations (AND, OR, and NOT), the Python operators are keywords *and*, *or*, and *not* (all in lower cases). For instance, the expression "*True or False*" will return *True*.

```
>>> True and True
True
>>> True or False
True
```

Python also allows us to compare two values or variable to see if they are equal (==), different (!=), or one is greater than (>) or smaller than (<)the other, or one is greater or equal to (>=) or smaller or equal to (<=)the other.

Assignment operators are used in Python to assign values to variables. = is the simplest assignment operator; for instance, "*x = 5*" assigns value "5" to the variable "x". Python also supports compound assignment operators; for instance, "*x += 1*" is equivalent to "*x = x +1*" (i.e., add 1 to x and assign the result

to x). We will use these operators a lot when manipulating dynamic stimuli in PsychoPy, e.g., moving a Gabor patch on the screen (see Chapter 2).

Other operators frequently used in Python include identity operators (*is*, *is not*) and membership operators (*in*, *not in*). These operators help make your code resemble a natural language. In the example statements below, "*1 in x*" returns *True* because 1 is a member of x ([1,2,3]).

```
>>> x = [1,2,3]
>>> x is False
False
>>> 1 in x
True
>>> 3 not in x
False
```

The last thing we need to discuss is operator precedence, that is, which operators are evaluated first by the Python interpreter. For instance, in "5 + 3*2", the multiplication operation was carried out first. There is no need to delve into the details; the official Python documentation has a section listing the operators in ascending order of precedence. [6]

Data type conversion

One can use the built-in *type*() function to reveal the type of a number or any other Python data types. The null value *None* has its own data type, i.e., *NoneType*.

```
>>> a = 3.14
>>> type(a)
<class 'float'>
>>> type(True)
<class 'bool'>
>>> type(None)
<class 'NoneType'>
```

[6] https://docs.python.org/3/reference/expressions.html#operator-precedence

There are built-in Python functions for data type conversion. For instance, the *int*() function converts a floating-point number or a string into an integer. The *hex*() function converts an integer to a hexadecimal string.

```
>>> int(10.24)
10
>>> int('00000011', 2)
3
>>> hex(255)
'0xff'
```

The *ord*() function converts a character to an integer, whereas the *chr*() function converts an integer to a Unicode character. The function *str*() converts a number or any other object into a string.

```
>>> ord('Z')
90
>>> chr(90)
'Z'
```

Control flow

In the simplest format, a Python script may contain multiple statements that can be evaluated and executed line-by-line (like the script shown in Figure 1-2). However, a useful script typically involves more than the sequential execution of statements and relies on sophisticated control flow mechanisms. Before we delve into the frequently used control flow features of Python, it is worth noting that Python statements are grouped by indentation instead of curly brackets or parentheses, as in some programming languages. The code snippet shown below will sum up all integers that are smaller than 100. We will discuss the for-loop in a later section; for now, it is sufficient to know that the for-loop below contains a single statement, x += i. We can tell this from the indentation of this line, which indicates it is part of the for-loop.

```
x = 0
for i in range(100):
    x += i
```

```
print(x)
```

if statement

Often a statement needs to be executed only when certain conditions are met (i.e., conditional execution). For instance, we would like to present the target stimulus only after a leading fixation cross is on the screen for 1000 milliseconds, or we will move on to the testing stage only if the accuracy of the subject is over 80% in the practice stage. In Python, a simple conditional execution usually involves *if* statements in the following format.

```
if expression_a:
    do_somehting_a
elif expression_b:
    do_somehing_b
elif expression_c:
    do_something_c
else:
    do_something_d
```

If “expression_a” is *True*, the Python interpreter will execute “do_something_a”; if not, the interpreter will evaluate the other expressions. The *elif* (if-else) and *else* statements are optional, but they are helpful when you need to determine if one of the multiple conditions are met. In the sample script below, we first show a prompt in the shell, asking users to enter a string and press ENTER to confirm. Then, we examine the length of the string.

```
s = input('Enter a string here: ')
if len(s) < 5:
    print('Length of string: s < 5')
elif len(s) < 10:
    print('Length of string: 4 < s < 10')
else:
    print('Length of string: s > 9')
```

for statement

A *for* statement is used to iterate over the elements of a sequence (e.g., string, list, tuple) or other iterable objects (e.g., an *xrange*() object).

```
for target_list in iterable_object:
    do_something_a
else:
    do_something_b
```

When evaluating a *for*-statement, the items in the "iterable_object" will be assigned to "target_list" in order. The else clause is optional; it is executed (if present) when the items in the "iterable_object" are exhausted. One can use a break statement to break out a for loop if needed. Note that the Python interpreter will not remove the variable names in "target_list" when the *for*-loop finishes.

```
iterable = list(zip(range(1,10), range(101,110)))
print(iterable,'\n')

for i, j in iterable:
    print(i, j)
    if i > 4:
        break

print('Variables i and j will persist after the loop is finished')
print(i, j)
```

In this short script, we first constructed a list of tuples. The "target_list" in the *for* statement contains *i* and *j*; When evaluating the for-loop, the values in the tuples are passed to *i* and *j* on each iteration. As is clear from the output, the Python interpreter does not automatically clear the variables *i* and *j* when the *for*-loop finishes executing.

```
[(1, 101), (2, 102), (3, 103), (4, 104), (5, 105), (6, 106), (7, 107), (8,
108), (9, 109)]

1 101
2 102
3 103
4 104
5 105
Variables i and j will persist after the loop is finished
5 105
```

while statement

Statements in a *while*-loop are repeatedly executed as long as a conditional expression is true. A while loop can use an optional "else" clause; the interpreter will execute the else clause when the *while*-loop terminates (e.g., when the conditional expression is false). As with the for loop, a break statement can be used to exit the loop (without executing the else clause).

```
while conditional_expression:
    do_something_a
else:
    do_something_b
```

A good example is worth a thousand words. In the short script below, the conditional expression is "*i < 100*". The *else* part will print out a notification after 1 to 100 have been printed out in the shell.

```
i = 0
while i < 100:
    i += 1
    print(i)
else:
    print('while loop ended')
```

More on looping

The *items*() of a dictionary is iterable and we can use it to iterate over all the key-value pairs in a dictionary. For instance,

```
dict = {'a':1, 'b':2, 'c':3}
for k, v in dict.items():
    print(k, v)
```

The *enumerate*() function also allows us to loop over and index the elements of a sequence (e.g., list, tuple, sets). The indices start from 0 by default, but one can optionally specify the starting index (e.g., 1 in the command below).

```
>>> s = ('a', 'b', 'c')
```

```
>>> for i, v in enumerate(s, 1):  print(i, v)

1 a
2 b
3 c
```

The *zip*() function is another handy tool when you need to loop over two or more sequences at the same time. For instance,

```
for i, j in zip(('a', 'b', 'c'), (1, 2, 3)):
    print(i, j)
```

The *range*() function, which returns an object that is essentially an iterator, offers yet another way of looping. For instance,

```
>>> for i in range(1,5): print('This is item: ', i)

This is item:  1
This is item:  2
This is item:  3
This is item:  4
```

List comprehension

List comprehension is a concise way of creating a list from another list. For instance, the *range*() function can be used to create a list of integers, e.g., *list(range(5))* will return [0, 1, 2, 3, 4]. What if we would like to create a list of even numbers that are smaller than 20? One cumbersome approach would be to declare an empty list and then use a loop to add in each of the even numbers.

```
evens = []
for i in range(20):
    if i%2 == 0:
        evens.append(i)
```

With list comprehension, one line of code is enough.

```
evens = [i for i in range(20) if i%2 == 0]
```

As shown in the example above, a list comprehension consists of brackets containing an expression (*i*) followed by a *for*-statement, then zero or more *for* or *if* statement. Here is a more complicated one,

```
>>> [(x, y) for x in [1, 2] for y in [1, 2, 3, 4] if x == y]
[(1, 1), (2, 2)]
```

Without the "*if x == y*" statement, we get

```
>>> [(x, y) for x in [1, 2] for y in [1, 2, 3, 4]]
[(1, 1), (1, 2), (1, 3), (1, 4), (2, 1), (2, 2), (2, 3), (2, 4)]
```

With list comprehension, one can also apply a function to a list to create a new list, for instance,

```
>>> [abs(x) for x in range(-5, 5)]
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4]
```

Set comprehension and dictionary comprehension are also supported in Python, though the returning values are sets and dictionaries instead. In the dictionary comprehension example below, the keys and values are swapped.

```
>>> x = set(range(10))
>>> {i*2 for i in x}
{0, 2, 4, 6, 8, 10, 12, 14, 16, 18}
>>> y = {'a':1, 'b':2, 'c':3}
>>> {key:value for value, key in y.items()}
{1: 'a', 2: 'b', 3: 'c'}
```

Functions

Functions are very useful when the same task needs to be performed multiple times in a script. Python comes with lots of built-in functions, such as *int*(), *list*(), *abs*(), etc. We can also define custom functions to add additional features that are useful in our experimental script. For instance, drawing a Gabor on the

screen is something worth implementing in a function, so we can easily vary its position, size, and spatial frequency, etc. each time we want to draw it. A function definition starts with the keyword *def*, followed by a function name and a list of arguments one can pass to the function. The arguments of a function are essentially variables one can feed to the function for processing (e.g., Gabor size, position, and spatial frequency). The body of a function usually contains one or more statements and an optional line specifying the value that the function "returns" (outputs).

```
def function_name(arg_1, arg_2, ...):
    """Doc string"""

    Statement 1
    Statement 2
    ...
    return value
```

In Python, the return value of a function is optional; if no return value is specified, the function will return *None*. For the Gabor drawing function, a pseudo function definition could be something like:

```
def draw_gabor(position, size, spatial_frequency):
    """ Draw a Gabor on-screen"""

        position - a 2-element tuple specifying screen coordinates
        size - diameter of the gabor in pixels
        spatial_frequency - cycles per pixel, e.g., 0.001"""

    various pygame draw commands

    return 0
```

The text included in the triple quotes is a doc-string, i.e., the help info for the "*draw_gabor*()" function. After declaring the "*draw_gabor*" function, all we need to do is to call this function with the relevant arguments next time we need to draw a Gabor.

```
draw_gabor((1024,768), 100, 0.002)
```

The use of a function usually involves the passing of certain arguments to the function. In the example above, the arguments are “position”, “size”, and “spatial_freuency”.

When defining a function, the arguments can take default values, for instance,

```
def draw_gabor(position, size=100, spatial_frequency=0.01):
    ...

    return
```

If no values are passed to these arguments when calling a function, the function will use the default values. So, you can omit the *size* and *spatial_frequency* arguments of the above-defined function; for instance, *draw_gabor((500,400))* will draw a Gabor at position (500, 400) that is 100 pixels large, with a spatial frequency of 0.01 cycle/pixel.

When calling a function, one can pass values to arguments with or without the keyword (e.g., “size”). If keywords are not used in a function call, you need to pass the values to the function in the same order that the function would expect (e.g., “position”, “size”, “spatial_frequency”). With the keywords, the sequence of the values does not matter anymore so that the above function call can take the following format:

```
draw_gabor(size=100, position=(400,300), spatial_frequency=0.002)
```

Output

For the output of a program, we can use the *print*() function to present it on the standard output device (screen) or write it to files.

Output formatting

It is usually helpful to format the output in a human-readable format. The “%” operator used to be the standard way of embedding values in a string. For instance, in the command below, three numbers are separated by “tabs” (\t).

The %ds are placeholders that will be replaced by the numbers in the tuple following the string, i.e., [1, 2, 3].

```
>>> print('%d\t%d\t%d' %(1,2,3))
1       2       3
```

With floating numbers, one can also specify how many decimal points to have in the formatted string with the % operator, for instance, to retain 3 decimal points, one can use "%.3f", i.e., formatting a float value ("f") with 3 decimal places.

```
>>> print('PI = %.3f' % 3.1415926)
PI = 3.142
```

One can still use this old fashion string formatting convention in Python 3, but there are other string formatting approaches.

The recommended method is the *format*() function, which works much like the old fashion way, but with a bit more flexibility. For instance, the values that we would like to include in the string do not necessarily need to appear in sequence. In the example below, the second value "eggs" appears first as the first {} expression is referring to the second item in the value list.

```
>>> '{1} and {0}'.format('spam', 'eggs')
'eggs and spam'
```

For the value list, i.e., the arguments for *format*(), one can also use keywords so that one can use the keywords in the formatted string.

```
>>> '{food} is {adjective}.'.format(food='Canadian Chinese food',
adjective='horrible')
'Canadian Chinese food is horrible.'
```

Without the keyword (e.g., "food" and "adjective" in the above command), the values are indexed according to their order in the value list, starting from 0.

One can also use f-strings (strings prefixed by “f” or “F”) to place the return value of Python expressions directly inside a string. In the example below, the Python expression is the statement in the {}; the value we would like to include in the string is *math.pi*. The part following the colon specifies the format of the value; the integer 10 means that the value occupies 10-character spaces; “.3f” requires Python to display 3 decimal places for the floating-point value (*math.pi*).

```
>>> f'Pi is close to {math.pi:10.3f}'
'Pi is close to      3.142'
```

Files

Programing an experimental task undoubtedly involves the opening of files to save the data collected in a testing session, e.g., reaction time. Of course, many experimental tasks may also require opening files that specify the trial parameters. The simplest solution would be to call the *open*() function, which returns a file object that we can manipulate (e.g., adding a new data line). The *open*() function requires two arguments: file name and mode.

```
open(filename, mode)
```

The first argument is a string containing the file name (including the path), whereas the second argument is a string specifying the operation mode of the file object. The frequently used operation modes are listed below. The mode argument is optional; *'r'* will be assumed if the mode argument is omitted.

- *'r'* –read-only, file cannot be modified
- *'w'*—write, overwrite the existing file with the same filename
- *'a'*—appending, append to the existing file with the same filename
- *'r+'*—read and write

Files usually operate in text mode, with which you read and write strings. To read the contents of a file, call the *read*(size) function. The *size* argument specifies the quantity of data one wants to read (e.g., numbers of characters); the entire file will be read into memory if you omit this option. One can also call the *readline*() function to extract a single line from a file. Repeatedly calling this

method will eventually reach the end of a file. In the example script below, we first write three lines into a file; then, open it to read its contents with the *read*() and *readline*() function. A faster and more memory efficient way for reading lines from a file is to loop over the file object (see example commands below).

```
# write text into a file
f = open('file_op.txt', 'w')
f.write('this is line 1\n')
f.write('this is line 2\n')
f.write('this is line 3\n')
f.close()

# open a file to read its contents
f_r = open('file_op.txt', 'r')
txt = f_r.read(7) # read 7 characters
f_r.close()

# readline()
f_r = open('file_op.txt', 'r')
line_1 = f_r.readline()# read line 1
line_2 = f_r.readline()# read line 2
line_3 = f_r.readline()# read line 2
print(line_1)
print(line_2)
print(line_3)
f_r.close()

# loop over a file object
f_r = open('file_op.txt', 'r')
for line in f_r:
    print(line)
```

In the output shown below, an empty line is printed in-between the lines because the *readline*() function will extract a line with the "newline" character (\n) at the end. The "newline" character is necessary for the Python interpreter to tell if it has reached the end of a file; the last line of a file does not end with the "newline" character.

```
this is line 1

this is line 2
```

```
this is line 3

this is line 1

this is line 2

this is line 3
```

Modules

Modules are files that you can import into a script to access the functions, classes, and constants defined within it. A module can be a script that you wrote for routines that you frequently call in your experiments. It usually ends with the ".py" extension, and the name of a module is accessible as a global variable "__name__" (two underscores). For instance, the following commands will print the name of the "pygame" module in the shell.

```
>>> import pygame
>>> pygame.__name__
'pygame'
```

In Python, everything (constant, function, and class, etc.) is an object. The objects in a module are accessible with the module name, followed by a period ('.') and the name of the object. For instance, *pygame.K_8* is a constant in the *pygame* library, which corresponds to the ASCII code of number 8 (i.e., 56). To call the various drawing functions in Pygame, one can refer to these functions with *pygame.draw.circle(*arguments)*.

To access the objects in a module, you need to import it into your script or the shell with the *import* command. You can import a module and rename it, for instance,

```
>>> import numpy as np
```

You can import all objects from a module (generally not recommended).

```
>>> from pygame import *
```

You can also selectively import a few objects from a module, for instance,

```
from math import sin, cos, hypot
```

To show what objects (names) are defined in a module, just call the function *dir*(). For instance, *sin* will show up in the returned list if you call *dir(math)*. If no argument is provided, *dir*() will return the names that have been imported or created in the current shell session.

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd',
'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

Choose a library to create your “video games”

A computer-based psychology task is essentially a “video game”, in which visual and auditory stimuli are presented, and the participants respond to various task manipulations with a keyboard, a mouse, or a gamepad. So, in principle, one can use any multimedia library to program an experimental task, though some libraries may not be suitable for tasks that require precise timing. For instance, programming a response time task with Flash sounds like a terrible idea to me.

There are lots of multimedia libraries that one can use to program experimental tasks in Python, for instance, Pygame and Pyglet. Some libraries are explicitly built for psychology experiments, for example, Psychopy, pyexperiment, and visionEGG. This latter category of libraries usually features functions for experimental purposes, for example, a staircase procedure or complex visual stimuli (e.g., checkerboards and gratings). In this book, I will cover PsychoPy for its growing popularity and its various handy features that are attractive to vision scientists (see Chapter 2). I will also include Pygame for its reliability and its minimal dependence on other packages or libraries (see Chapter 3).

Where to go from here

The Python basics covered in this chapter are by no means complete. For instance, we have not discussed any topic on object-oriented programming, which is usually unnecessary for experimental scripts that are a couple of hundred lines long. If you would like to learn more about Python, I strongly recommend the official CPython tutorial, especially if you have little experience in programming, https://docs.python.org/3/tutorial/index.html. The beginner's guide on python.org is also an excellent source to look for various introductory tutorials and guides, https://wiki.python.org/moin/BeginnersGuide. There are lots of Python books out there, for instance, *Dive into Python* by Mark Pilgrim. A free online version is available at https://www.diveinto.org/python3/table-of-contents.html.

For Pygame, I would recommend *Beginning Game Development with Python and Pygame: From Novice to Professional* by Will McGugan. You may also find some free online books on Pygame, for instance, *Making Games with Python & Pygame* by Al Sweigard, https://inventwithpython.com/pygame.

For PsychoPy, consider *Building Experiments in PsychoPy* by Jon Pierce, the author of PsychoPy. It is a useful reference for the graphical programming interface of PsychoPy--Builder, but it also covers coding in chapters targeting advanced users.