

Python for you and me

Kushal Das

2024-05-15T22:29:15Z

Contents

Welcome to Python for you and me	1
Installation	7
On Windows	7
On GNU/Linux	7
The Beginning	9
Using the Python interpreter	9
Using a source file	9
Whitespaces and indentation	10
Comments	11
Modules	12
Evaluation your code from a Python file in the interpreter	12
Using mu editor	15
Installation	15
Using mu	15
Executing code	15
Variables and Datatypes	17
Keywords and Identifiers	17
Reading input from the Keyboard	18
Some Examples	19
Average of N numbers	19
Temperature conversion	20
Multiple assignments in a single line	21
Formatting strings	22
.format method	23
Left of right aligned	23
Operators and expressions	25
Operators	25
Example of integer arithmetic	25
Relational Operators	26

Relational Operators	26
Logical Operators	27
Shorthand Operator	27
Expressions	28
Type Conversions	28
evaluatequ.py	29
quadraticequation.py	29
salesmansalary.py	30
If-else , the control flow	31
If statement	31
Else statement	31
Truth value testing	32
match statements	32
Learning using hardware simulation	35
Circuit Playground Express	35
Turning on all NeoPixels on the board with Red colour	37
Yellow light solution	39
Learning about time.sleep	39
RGB problem	39
RGB solution	39
Looping	43
While loop	43
Fibonacci Series	44
Power Series	45
Multiplication Table	45
Some printing * examples	46
Lists	48
For loop	49
range() function	50
Continue statement	50
Else statement in a loop	51
Game of sticks	51
Looping over hardware	53
One NeoPixel at a time	53
First Red and then Blue	54
Data Structures	57
Lists	57
Using lists as stack and queue	59
List Comprehensions	60
A few special functions related to lists	60
Tuples	60

Sets	62
Dictionaries	62
Looping over a dictionary	63
students.py	65
Assignment problems	66
Create a calculator	66
Strings	67
Different methods available for Strings	68
Strip the strings	69
Justifying text	70
Finding text	70
Palindrome checking	70
Number of words	71
Iterating over all characters of a string	71
Functions	73
Defining a function	73
Local and global variables	76
Default argument value	77
Keyword arguments	78
Keyword only argument	78
Positional only argument	79
Docstrings	79
Higher-order function	80
map	81
Parameters and arguments	81
*args and **kwargs in function definition	82
HOWTO Write a function	82
File handling	83
File opening	83
Closing a file	83
Reading a file	84
Using the with statement	85
Writing in a file	86
copyfile.py	86
Count spaces, tabs and new lines in a file	87
Let us write some real code	88
Exceptions	91
NameError	91
TypeError	91
How to handle exceptions?	91
Raising exceptions	93
Using finally for cleanup	93

Class	95
Your first class	95
__init__ method	96
Unique class level variables	97
__repr__ method	98
Inheritance	98
student_teacher.py	98
Multiple Inheritance	100
Encapsulation in Python	100
Deleting an object	101
Getters and setters in Python	101
Properties	102
Special dunder methods in classes	103
__len__ method	103
__contains__ method	103
__new__ method	104
Creating a new context manager	104
Deep down inside	105
 Modules	 109
Introduction	109
Importing modules	110
Packages and submodules	112
__all__ in __init__.py	112
Default modules	113
Module os	114
Requests Module	115
Getting a simple web pages	115
Command line arguments	116
TAB completion in your Python interpreter	116
 Collections module	 119
Counter	119
Counter example	119
defaultdict	120
defaultdict example	120
namedtuple	120
Named tuple	120
 Using VS Code as your primary Python editor	 123
Installing VS Code	123
Using VS Code	123
Install the Python extension	126
Start working on your code	126
Install the Device Simulator Express extension	128

PEP8 Guidelines	129
Introduction	129
A Foolish Consistency is the Hobgoblin of Little Minds	129
Code lay-out	130
Indentation	130
Tabs or Spaces?	131
Maximum Line Length	131
Blank Lines	132
Encodings (PEP 263)	132
Imports	133
Whitespace in Expressions and Statements	134
Pet Peeves	134
Other Recommendations	135
Comments	136
Block Comments	137
Inline Comments	137
Documentation Strings	137
Version Bookkeeping	138
Naming Conventions	138
Descriptive: Naming Styles	138
Prescriptive: Naming Conventions	139
References	142
Copyright	142
Iterators, generators and decorators	143
Iterators	143
Generators	144
Generator expressions	147
Closures	148
Decorators	148
Virtualenv	151
Installation	151
Usage	151
Pipenv	152
Installing pipenv	152
Using pipenv	153
Exiting from the virtualenv	154
Pipfile and Pipfile.lock	154
Pipfile	154
Type hinting and annotations	157
First example of type annotation	157
Using mypy and more examples	158
Installing mypy	158
Our example code	158

Using mypy	159
Enabling the first few type annotations	159
More examples of type annotations	161
Simple testing in Python	163
What we should test ?	163
Unit testing	163
unittest module	163
Factorial code	163
Which function to test ?	164
Our first test case	164
Description	165
Different assert statements	165
Testing exceptions	166
mounttab.py	167
After refactoring	167
Test coverage	169
Coverage Example	169
A project structure	171
Primary code	171
LICENSE file	172
Installing flit package	173
pyproject.toml	173
Building a package	174
Python Package Index (PyPI)	175
Creating account	175
Uploading your project	175
Install from the test PyPI	175
More readings	176
Building command line applications with Click	177
Installation, and development tips	177
Using echo for printing text	178
Boolean flags	179
Standard options in the command line	179
Same option multiple times	180
Help text for the script	180
Super fast way to accept password with confirmation	181
Must have arguments	182
Simple GUI application development using PyperCard	185
Installing PyperCard in a virtualenv	185
Hello World example	185
Two Cards and a button	186
Taking simple text input	187

Check name problem	190
Check name solution	190

Introduction to Flask **193**

What is flask?	193
What are template engines?	193
A “Hello world” application in flask	194
Using arguments in Flask	196
Additional work	197

Index **199**

—	199
A	199
C	199
D	199
E	199
F	200
I	200
L	200
N	200
P	200
R	200
S	200
T	200

Welcome to Python for you and me

This is a simple book to learn Python programming language, it is for the programmers who are new to Python.

You can find the latest version of the book here [<https://pymbook.readthedocs.io/en/latest/>].

If you are new to command line in Linux, you can read lym [<https://lym.readthedocs.io/>].

Contents:

- Installation
 - On Windows
 - On GNU/Linux
- The Beginning
 - Using the Python interpreter
 - Using a source file
 - Whitespaces and indentation
 - Comments
 - Modules
 - Evaluation your code from a Python file in the interpreter
- Using mu editor
 - Installation
 - Using mu
 - Executing code
- Variables and Datatypes
 - Keywords and Identifiers
 - Reading input from the Keyboard
 - Some Examples
 - Multiple assignments in a single line
 - Formatting strings
- Operators and expressions
 - Operators
 - Example of integer arithmetic
 - Relational Operators

- Logical Operators
 - Shorthand Operator
 - Expressions
 - Type Conversions
 - evaluateequ.py
 - quadraticequation.py
 - salesmansalary.py
- If-else , the control flow
 - If statement
 - Else statement
 - Truth value testing
 - match statements
- Learning using hardware simulation
 - Circuit Playground Express
 - Turning on all NeoPixels on the board with Red colour
- Looping
 - While loop
 - Fibonacci Series
 - Power Series
 - Multiplication Table
 - Some printing * examples
 - Lists
 - For loop
 - range() function
 - Continue statement
 - Else statement in a loop
 - Game of sticks
- Looping over hardware
 - One NeoPixel at a time
 - First Red and then Blue
- Data Structures
 - Lists
 - Using lists as stack and queue
 - List Comprehensions
 - A few special functions related to lists
 - Tuples
 - Sets
 - Dictionaries
 - students.py
 - Assignment problems
 - Create a calculator
- Strings
 - Different methods available for Strings
 - Strip the strings
 - Justifying text
 - Finding text

- Palindrome checking
 - Number of words
 - Iterating over all characters of a string
- Functions
 - Defining a function
 - Local and global variables
 - Default argument value
 - Keyword arguments
 - Keyword only argument
 - Positional only argument
 - Docstrings
 - Higher-order function
 - map
 - Parameters and arguments
 - *args and **kwargs in function definition
 - HOWTO Write a function
- File handling
 - File opening
 - Closing a file
 - Reading a file
 - Using the with statement
 - Writing in a file
 - copyfile.py
 - Count spaces, tabs and new lines in a file
 - Let us write some real code
- Exceptions
 - NameError
 - TypeError
 - How to handle exceptions?
 - Raising exceptions
 - Using finally for cleanup
- Class
 - Your first class
 - `__init__` method
 - Unique class level variables
 - `__repr__` method
 - Inheritance
 - Multiple Inheritance
 - Encapsulation in Python
 - Deleting an object
 - Getters and setters in Python
 - Properties
 - Special dunder methods in classes
 - Creating a new context manager
 - Deep down inside
- Modules

- Introduction
 - Importing modules
 - Packages and submodules
 - `__all__` in `__init__.py`
 - Default modules
 - Module `os`
 - Requests Module
 - Command line arguments
 - TAB completion in your Python interpreter
- Collections module
 - Counter
 - `defaultdict`
 - `namedtuple`
- Using VS Code as your primary Python editor
 - Installing VS Code
 - Using VS Code
 - Install the Python extension
 - Start working on your code
 - Install the Device Simulator Express extension
- PEP8 Guidelines
 - Introduction
 - A Foolish Consistency is the Hobgoblin of Little Minds
 - Code lay-out
 - Whitespace in Expressions and Statements
 - Comments
 - Version Bookkeeping
 - Naming Conventions
 - References
 - Copyright
- Iterators, generators and decorators
 - Iterators
 - Generators
 - Generator expressions
 - Closures
 - Decorators
- Virtualenv
 - Installation
 - Usage
 - Pipenv
 - Pipfile and Pipfile.lock
- Type hinting and annotations
 - First example of type annotation
 - Using `mypy` and more examples
 - More examples of type annotations
- Simple testing in Python
 - What we should test ?

- Unit testing
 - unittest module
 - Factorial code
 - Which function to test ?
 - Our first test case
 - Description
 - Different assert statements
 - Testing exceptions
 - mounttab.py
 - After refactoring
 - Test coverage
 - Coverage Example
- A project structure
 - Primary code
 - LICENSE file
 - Installing flit package
 - pyproject.toml
 - Building a package
 - Python Package Index (PyPI)
 - Install from the test PyPI
 - More readings
- Building command line applications with Click
 - Installation, and development tips
 - Using echo for printing text
 - Boolean flags
 - Standard options in the command line
 - Same option multiple times
 - Help text for the script
 - Super fast way to accept password with confirmation
 - Must have arguments
- Simple GUI application development using PyperCard
 - Installing PyperCard in a virtualenv
 - Hello World example
 - Two Cards and a button
 - Taking simple text input
 - Check name problem
 - Check name solution
- Introduction to Flask
 - What is flask?
 - What are template engines?
 - A “Hello world” application in flask
 - Using arguments in Flask
 - Additional work

Installation

In this chapter you will learn how to install Python 3, the latest version of the language.

On Windows

Download the latest Windows(TM) installer from the Python site, x86_64 [<https://www.python.org/ftp/python/3.10.1/python-3.10.1-amd64.exe>]. Install it just as any other Windows software.

On GNU/Linux

Install the latest Python from the distribution's repository.

For Fedora 23 and above Python3 is in the system by default.

For Debian/Ubuntu

```
[user@host]$ sudo apt-get install python3-all python3-venv python3-pip
```

The Beginning

Let's look at our first code, hello world. Because Python is an interpreted language, you can write the code into the Python interpreter directly or you can write the code in a file and then run the file. In this topic, we will first write the code using the interpreter, after starting Python in the command prompt (shell or terminal). If you are new to Linux command line, you can read about Linux commands in this book <<https://lym.readthedocs.io/en/latest/>>.

The following is from Fedora 34 machine running Python 3.10

```
Python 3.10.1 (main, Dec  9 2021, 00:00:00) [GCC 11.2.1 20211203 (Red Hat 11.2.1-7)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Using the Python interpreter

In our first code we are going to print “Hello World!” using the interpreter. To generate the output, type the following:

```
>>> print("Hello World!")
Hello World!
```

Using a source file

As a serious programmer, you might want to write the above code into a source file. Use any text editor you like to create the file called helloworld.py. I used vi. You can even use GUI based tools like Kate [<https://kate-editor.org/get-it/>] or gedit [<https://wiki.gnome.org/Apps/Gedit>]. In the later part of the book, I will help you to use a few different editors.

Open a shell or terminal and perform these steps.

1. Enter the following text:



2. Type the following command to make the file executable:

```
$ chmod +x helloworld.py
```

3. Run the code by typing the filename.

```
$ ./helloworld.py  
Hello World!
```

On the first line you can `#!/`, what we call it sha-bang. The sha-bang indicates that the Python interpreter should run this code. On the next line we are printing a text message. In Python we call all the lines of text “strings.”

Whitespaces and indentation

In Python whitespace is an important thing. We divide different identifiers using spaces. Whitespace in the beginning of the line is known as indentation, but if you give wrong indentation it will throw an error. Below are some examples:

```
>>> a = 12  
    File "<stdin>", line 1  
      a = 12  
IndentationError: unexpected indent
```

Warning

There is an extra space in the beginning of the second line which is causing the error, so always look for the proper indentation. You can even get into this indentation errors if you mix up tabs and spaces. Like if you use spaces and

only use spaces for indentation, don't use tabs in that case. For you it may look same, but the code will give you error if you try to run it.

This also introduce us to the first error. It is okay to see more errors while you are learning to code or learning Python for the first time. Start reading the errors from the end, and slowly go up (if there are many lines). We will learn more in the exceptions chapter of this book.

We can have few basic rules ready for spaces and indentation.

- Use 4 spaces for indentation.
- Never mix tab and spaces.
- One blank line between functions.
- Two blank lines between classes.

There are more places where you should be following the same type of whitespace rules:

- Add a space after “,” in dicts, lists, tuples, and argument lists and after “:” in dicts.
- Spaces around assignments and comparisons (except in argument list)
- No spaces just inside parentheses.

Note

99% of programming errors I saw are due to typing mistakes. People can learn logic well, but you should be able to type well too. Learn touch typing, that one single skill will help you to become much better programmer than many other skills.

Comments

Comments are snippets of English text that explain what this code does. Write comments in the code so that is easier for others to understand. A comment line starts with `#`. Everything after that is ignored as a comment and does not affect the program.

```
>>> # This is a comment
>>> # The next line will add two numbers
>>> a = 12 + 34
>>> print(c) #this is a comment too :)
```

Comments are mainly for people who *develop* or *maintain* the codebase. So if you have any complex code, you should write enough comments inside so that anyone else can understand the code by reading the comments. Always give a space after `#` and then start writing the comment. You can also use some standard comments like:

```
# FIXME -- fix these code later
# TODO -- in future you have to do this
```

Modules

Modules are Python files that contain different function definitions or variables that can be reused. Module files should always end with a .py extension. Python itself has a vast module library with the default installation. We will use some of them later. To use a module you have to import it first.

```
>>> import math
>>> print(math.e)
2.71828182846
```

We will learn more about modules in the Modules chapter.

Evaluation your code from a Python file in the interpreter

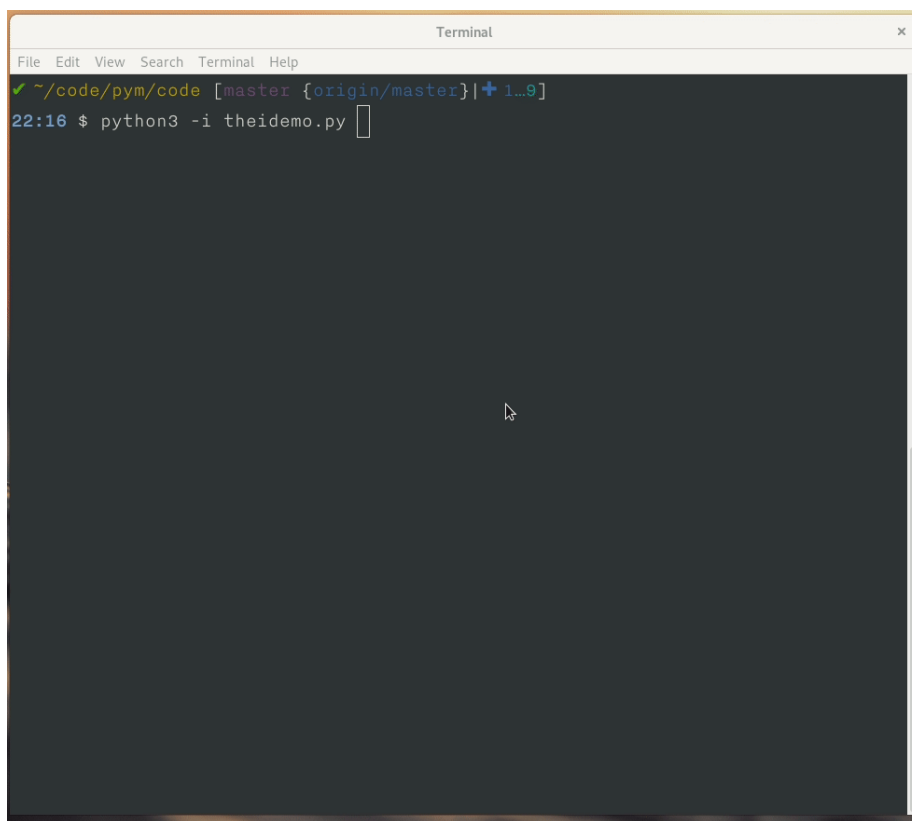
Many times we want to see how the code is working, and values of different variables inside of the code. We can do this interactively by copy-pasting the related code in the interpreter, but there is a better solution. We can use *-i* flag to the *python* interpreter and then interprets the given code, and provide the interpreter shell.

We will have the following code in a file name *theidemo.py*.

```
a = 10
name = "kushal"
c = 44
a = 20
```

Now let us see how the *-i* flag can be used.

EVALUATION YOUR CODE FROM A PYTHON FILE IN THE INTERPRETER13



A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows a prompt at the top: `~/code/pym/code [master {origin/master}|+ 1..9]`. Below this, the command `python3 -i theidemo.py` has been entered, and the cursor is positioned at the end of the line.

```
~/code/pym/code [master {origin/master}|+ 1..9]
22:16 $ python3 -i theidemo.py
```

Using mu editor

Using a proper editor to write code can help a lot while programming or while working on actual projects. In this chapter we will learn about **mu editor**, which is a very simple and beginner friendly editor for anyone starting with Python. It is developed by Nicholas Tollervey.

If you already have an editor of your choice, you can skip this chapter.

Installation

The total download size is around 150MB. Means it will take some time. Give the following command as your normal user.

```
$ python3 -m pip install -U mu-editor --user
```

Using mu

You can open the editor from the command line, type **mu** in the terminal.

```
$ python3 -m mu
```

Select Python 3 at the starting screen. Now, your mu editor is ready.

Executing code

You will have to write any code you want to execute, and then save it with a proper filename (ends with .py). Then click on the **Run** button to execute the code. If you want a REPL, then click on the **REPL** button.

Note

Remember not to give any space in the filename. Use `_` if required, example: `download_photos.py`

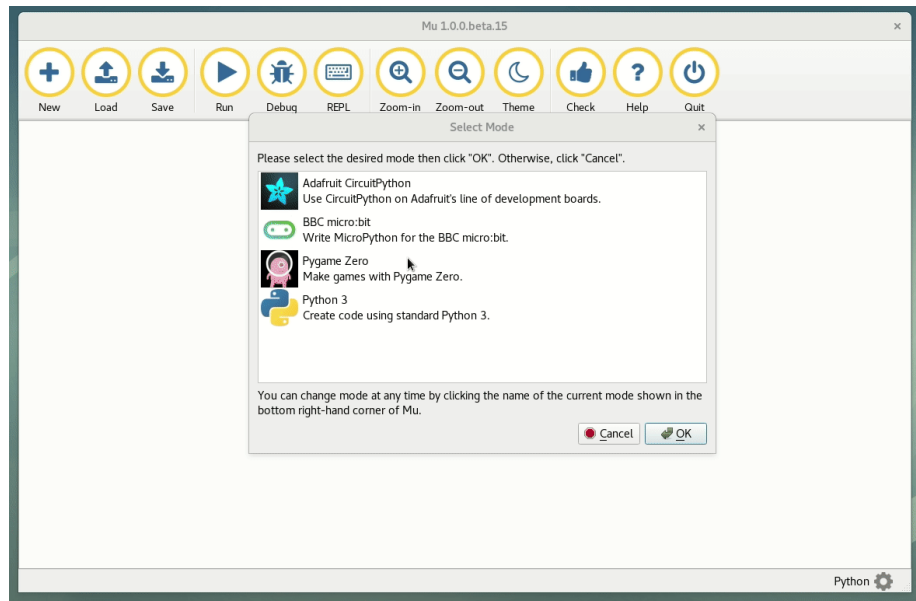


Figure 1:

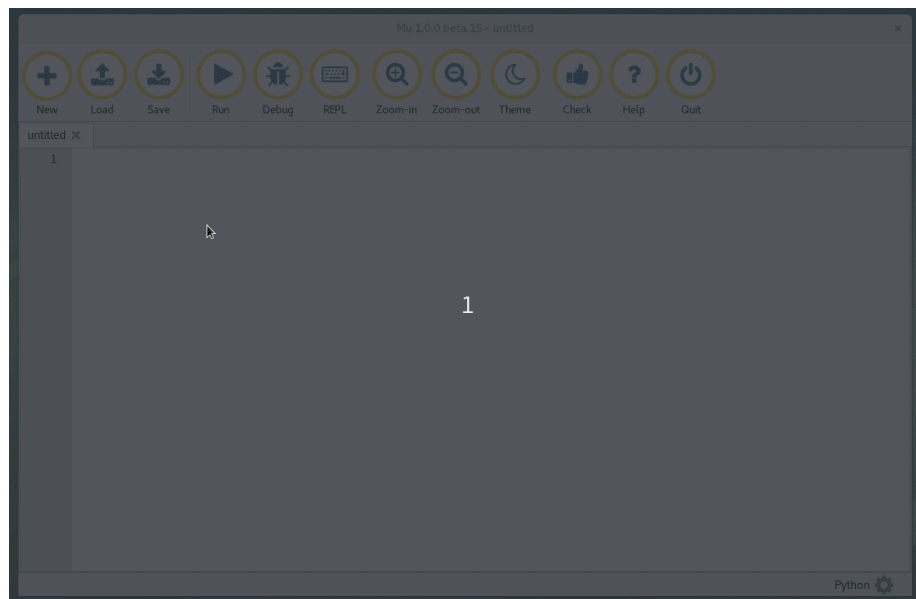


Figure 2:

Variables and Datatypes

Every programming language has its own grammar rules just like the languages we speak.

Keywords and Identifiers

The following identifiers are used as reserved words, or keywords of the language, and cannot be used as ordinary identifiers. They must be typed exactly as written here:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

In Python we don't specify what kind of data we are going to put in a variable. So you can directly write `abc = 1` and `abc` will become an integer datatype. If you write `abc = 1.0` `abc` will become of floating type. Here is a small program to add two given numbers

```
>>> a = 13
>>> b = 23
>>> a + b
36
```

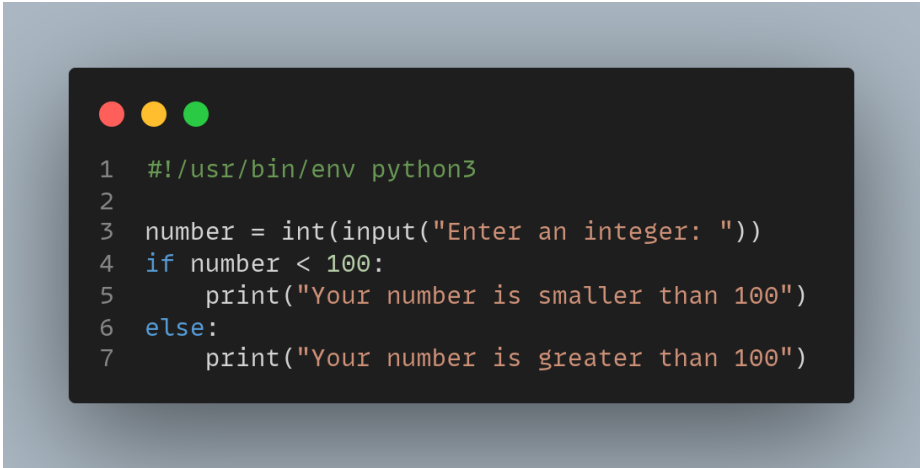
From the above example you can understand that to declare a variable in Python, what you need is just to type the name and the value. Python can also manipulate strings. They can be enclosed in single quotes or double quotes like

```
>>> 'India'
'India'
>>> 'India\'s best'
"India's best"
```

```
>>> "Hello World!"  
'Hello World!'
```

Reading input from the Keyboard

Generally the real life Python codes do not need to read input from the keyboard. In Python we use input function to do input. *input("String to show")* , this will return a string as output. Let us write a program to read a number from the keyboard and check if it is less than 100 or not. Name of the program is *testhundred.py*

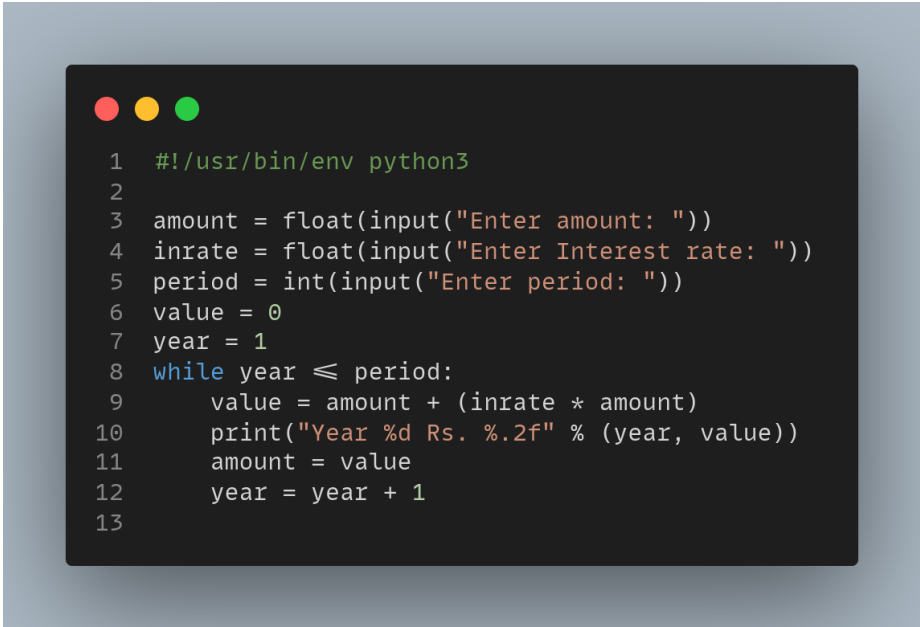


```
1  #!/usr/bin/env python3  
2  
3  number = int(input("Enter an integer: "))  
4  if number < 100:  
5      print("Your number is smaller than 100")  
6  else:  
7      print("Your number is greater than 100")
```

The output

```
$ ./testhundred.py  
Enter an integer: 13  
Your number is smaller than 100  
$ ./testhundred.py  
Enter an integer: 123  
Your number is greater than 100
```

In the next program we are going to calculate investments.



```
1  #!/usr/bin/env python3
2
3  amount = float(input("Enter amount: "))
4  inrate = float(input("Enter Interest rate: "))
5  period = int(input("Enter period: "))
6  value = 0
7  year = 1
8  while year <= period:
9      value = amount + (inrate * amount)
10     print("Year %d Rs. %.2f" % (year, value))
11     amount = value
12     year = year + 1
13
```

The output

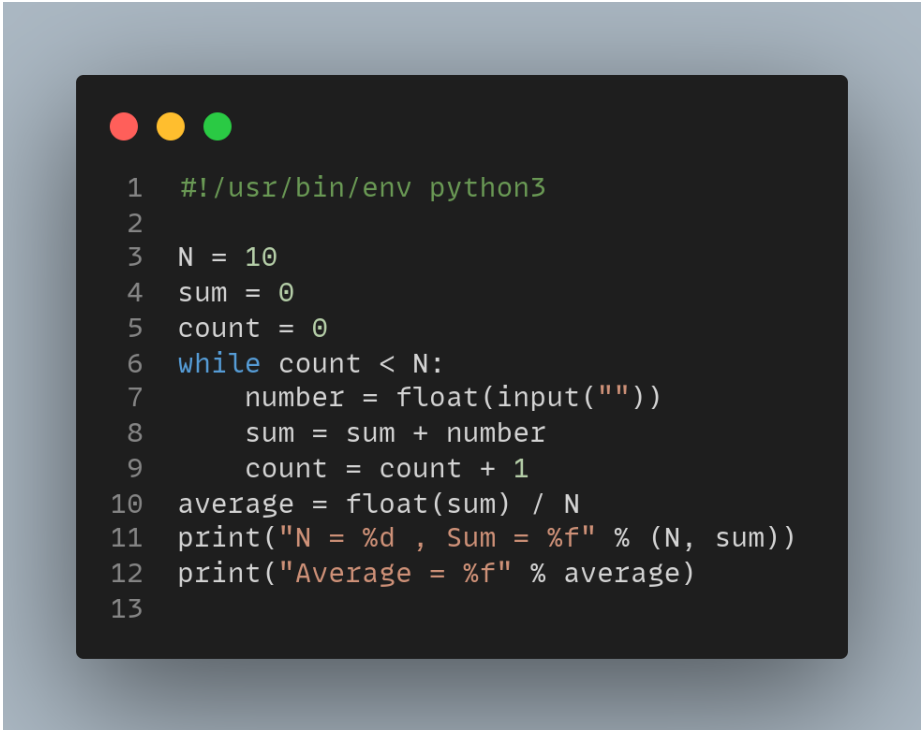
```
$ ./investment.py
Enter amount: 10000
Enter Interest rate: 0.14
Enter period: 5
Year 1 Rs. 11400.00
Year 2 Rs. 12996.00
Year 3 Rs. 14815.44
Year 4 Rs. 16889.60
Year 5 Rs. 19254.15
```

Some Examples

Some examples of variables and datatypes:

Average of N numbers

In the next program we will do an average of N numbers.



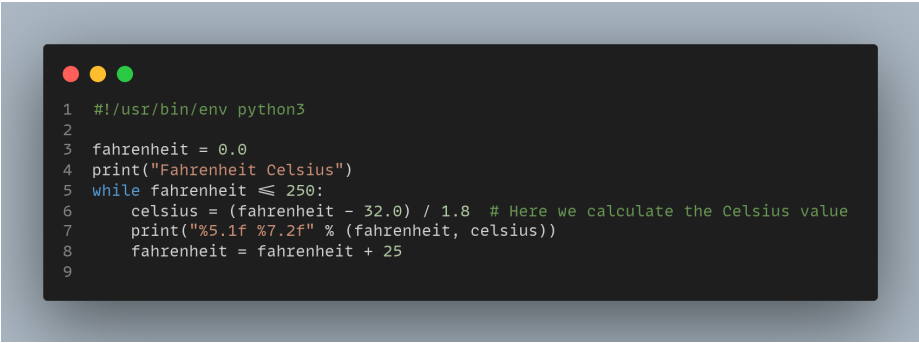
```
1  #!/usr/bin/env python3
2
3  N = 10
4  sum = 0
5  count = 0
6  while count < N:
7      number = float(input(""))
8      sum = sum + number
9      count = count + 1
10 average = float(sum) / N
11 print("N = %d , Sum = %f" % (N, sum))
12 print("Average = %f" % average)
13
```

The output

```
$ ./averagen.py
1
2.3
4.67
1.42
7
3.67
4.08
2.2
4.25
8.21
N = 10 , Sum = 38.800000
Average = 3.880000
```

Temperature conversion

In this program we will convert the given temperature to Celsius from Fahrenheit by using the formula $C = (F - 32) / 1.8$



```
1 #!/usr/bin/env python3
2
3 fahrenheit = 0.0
4 print("Fahrenheit Celsius")
5 while fahrenheit <= 250:
6     celsius = (fahrenheit - 32.0) / 1.8 # Here we calculate the Celsius value
7     print("%5.1f %7.2f" % (fahrenheit, celsius))
8     fahrenheit = fahrenheit + 25
9
```

The output

```
$ ./temperature.py
Fahrenheit Celsius
0.0  -17.78
25.0  -3.89
50.0  10.00
75.0  23.89
100.0  37.78
125.0  51.67
150.0  65.56
175.0  79.44
200.0  93.33
225.0  107.22
250.0  121.11
```

Multiple assignments in a single line

You can even assign values to multiple variables in a single line, like

```
>>> a , b = 45, 54
>>> a
45
>>> b
54
```

Using this swapping two numbers becomes very easy

```
>>> a, b = b , a
>>> a
54
>>> b
45
```

To understand how this works, you will have to learn about a data type called *tuple*. We use *comma* to create tuple. In the right hand side we create the tuple

(we call this as tuple packing) and in the left hand side we do tuple unpacking into a new tuple.

Below we have another example of tuple unpacking.

```
>>> data = ("Kushal Das", "India", "Python")
>>> name, country, language = data
>>> name
'Kushal Das'
>>> country
'India'
>>> language
'Python'
```

Tuples can not be modified. You will have to create another new tuple if you want have any changes. Many times, we create variables written in CAPS to mark values which will not be changed when the program is running (constants). For example, if we think about colours as tuples of RGB values, then we can define them as:

```
>>> RED = (255, 0, 0)
>>> GREEN = (0, 255, 0)
>>> BLUE = (0, 0, 255)
>>> print(RED)
(255, 0, 0)
```

Formatting strings

In Python 3, there are a few different ways to format a string. We use these methods to format a text dynamically. I will go through a few examples below.

In Python 3.6, we have a new way to do string formatting. PEP 498 [<https://www.python.org/dev/peps/pep-0498/>] introduces the concept called **f-strings**.

Here is the same example using *f-strings*:

```
>>> name = "Kushal"
>>> language = "Python"
>>> msg = f"{name} loves {language}."
>>> print(msg)
Kushal loves Python.
```

F-strings provide a simple and readable way to embed Python expressions in a string. Here are a few more examples.

```
>>> answer = 42
>>> print(f"The answer is {answer}")
The answer is 42
```

```
>>> import datetime
>>> d = datetime.date(2004, 9, 8)
>>> f"{d} was a {d:%A}, we started the mailing list back then."
'2004-09-08 was a Wednesday, we started the mailing list back then.'
```

If you want to know more about how this feature came into Python, watch this talk [<https://www.youtube.com/watch?v=M4w4wKveVo4>] from Mariatta Wijaya [<https://twitter.com/mariatta>].

From Python3.8 we can use the following style to help printing values along with the variable name. This is very useful while debugging the code.

```
>>> a = 1
>>> b = 2
>>> print(f"{a=} and {b=}")
a=1 and b=2
```

.format method

We can also use format method in a string inside.

```
>>> name = "Kushal"
>>> language = "Python"
>>> msg = "{0} loves {1}.".format(name, language)
>>> print(msg)
Kushal loves Python.
```

Left of right aligned

We can use `:<` or `:>` in `.format` call to print left or right aligned. In the following example we are printing first left aligned (10 characters) and right aligned.

```
>>> print("{:<10}".format("Sweden"))
Sweden
>>> print("{:>10}".format("Sweden"))
      Sweden
```

Operators and expressions

In Python most of the lines you will write will be expressions. Expressions are made of operators and operands. An expression is like $2 + 3$.

Operators

Operators are the symbols which tells the Python interpreter to do some mathematical or logical operation. Few basic examples of mathematical operators are given below:

```
>>> 2 + 3
5
>>> 23 - 3
20
>>> 22.0 / 12
1.8333333333333333
```

To get floating result you need to the division using any of operand as floating number. To do modulo operation use % operator

```
>>> 14 % 3
2
```

Example of integer arithmetic

The code

```
#!/usr/bin/env python3
days = int(input("Enter days: "))
months = days / 30
days = days % 30
print("Months = %d Days = %d" % (months, days))
```

The output

```
$ ./integer.py
```

```
Enter days: 265
Months = 8 Days = 25
```

In the first line I am taking the input of days, then getting the months and days and at last printing them. You can do it in a easy way

```
#!/usr/bin/env python3
days = int(input("Enter days: "))
print("Months = %d Days = %d" % (divmod(days, 30)))
```

The `divmod(num1, num2)` function returns two values , first is the division of num1 and num2 and in second the modulo of num1 and num2.

Relational Operators

You can use the following operators as relational operators

Relational Operators

Operator	Meaning
<	Is less than
<=	Is less than or equal to
>	Is greater than
>=	Is greater than or equal to
==	Is equal to
!=	Is not equal to

Some examples

```
>>> 1 < 2
True
>>> 3 > 34
False
>>> 23 == 45
False
>>> 34 != 323
True
```

// operator gives the floor division result

```
>>> 4.0 // 3
1.0
>>> 4.0 / 3
1.3333333333333333
```

Logical Operators

To do logical AND , OR we use *and* ,*or* keywords. *x and y* returns *False* if *x* is *False* else it returns evaluation of *y*. If *x* is *True*, it returns *True*.

```
>>> 1 and 4
4
>>> 1 or 4
1
>>> -1 or 4
-1
>>> 0 or 4
4
```

Shorthand Operator

x op = expression is the syntax for shorthand operators. It will be evaluated like *x = x op expression* , Few examples are

```
>>> a = 12
>>> a += 13
>>> a
25
>>> a /= 3
>>> a
8.333333333333334
>>> a += (26 * 32)
>>> a
840.3333333333334
```

shorthand.py example

```
#!/usr/bin/env python3
N = 100
a = 2
while a < N:
    print("%d" % a)
    a *= a
```

The output

```
$ ./shorthand.py
2
4
16
```

Expressions

Generally while writing expressions we put spaces before and after every operator so that the code becomes clearer to read, like

```
a = 234 * (45 - 56.0 / 34)
```

One example code used to show expressions

```
#!/usr/bin/env python3
a = 9
b = 12
c = 3
x = a - b / 3 + c * 2 - 1
y = a - b / (3 + c) * (2 - 1)
z = a - (b / (3 + c) * 2) - 1
print("X = ", x)
print("Y = ", y)
print("Z = ", z)
```

The output

```
$ ./evaluationexp.py
X = 10
Y = 7
Z = 4
```

At first x is being calculated. The steps are like this

```
9 - 12 / 3 + 3 * 2 - 1
9 - 4 + 3 * 2 - 1
9 - 4 + 6 - 1
5 + 6 - 1
11 - 1
10
```

Now for y and z we have parentheses, so the expressions evaluated in different way. Do the calculation yourself to check them.

Type Conversions

We have to do the type conversions manually. Like

```
float(string) -> float value
int(string) -> integer value
str(integer) or str(float) -> string representation
>>> a = 8.126768
>>> str(a)
'8.126768'
```


evaluateequ.py

This is a program to evaluate $1/x + 1/(x+1) + 1/(x+2) + \dots + 1/n$ series upto n , in our case $x = 1$ and $n = 10$

```
#!/usr/bin/env python3
sum = 0.0
for i in range(1, 11):
    sum += 1.0 / i
    print("%2d %6.4f" % (i, sum))
```

The output

```
$ ./evaluateequ.py
1 1.0000
2 1.5000
3 1.8333
4 2.0833
5 2.2833
6 2.4500
7 2.5929
8 2.7179
9 2.8290
10 2.9290
```

In the line $sum += 1.0 / i$ what is actually happening is $sum = sum + 1.0 / i$.

quadraticequation.py

This is a program to evaluate the quadratic equation

```
#!/usr/bin/env python3
import math
a = int(input("Enter value of a: "))
b = int(input("Enter value of b: "))
c = int(input("Enter value of c: "))
d = b * b - 4 * a * c
if d < 0:
    print("ROOTS are imaginary")
else:
    root1 = (-b + math.sqrt(d)) / (2.0 * a)
    root2 = (-b - math.sqrt(d)) / (2.0 * a)
    print("Root 1 = ", root1)
    print("Root 2 = ", root2)
```

salesmansalary.py

In this example we are going to calculate the salary of a camera salesman. His basic salary is 1500, for every camera he will sell he will get 200 and the commission on the month's sale is 2 %. The input will be number of cameras sold and total price of the cameras.

```
#!/usr/bin/env python3
basic_salary = 1500
bonus_rate = 200
commision_rate = 0.02
numberofcamera = int(input("Enter the number of inputs sold: "))
price = float(input("Enter the total prices: "))
bonus = (bonus_rate * numberofcamera)
commision = (commision_rate * numberofcamera * price)
print("Bonus          = %6.2f" % bonus)
print("Commision      = %6.2f" % commision)
print("Gross salary = %6.2f" % (basic_salary + bonus + commision))
```

The output

```
$ ./salesmansalary.py
Enter the number of inputs sold: 5
Enter the total prices: 20450
Bonus          = 1000.00
Commision      = 2045.00
Gross salary = 4545.00
```

If-else , the control flow

While working on real life of problems we have to make decisions. Decisions like which camera to buy or which cricket bat is better. At the time of writing a computer program we do the same. We make the decisions using if-else statements, we change the flow of control in the program by using them.

If statement

The syntax looks like

```
if expression:
    do this
```

If the value of *expression* is true (anything other than zero), do the what is written below under indentation. Please remember to give proper indentation, all the lines indented will be evaluated on the True value of the expression. One simple example is to take some number as input and check if the number is less than 100 or not.

```
#!/usr/bin/env python3
number = int(input("Enter a number: "))
if number < 100:
    print("The number is less than 100")
```

Then we execute the file.

```
$ ./number100.py
Enter a number: 12
The number is less than 100
```

Else statement

Now in the above example we want to print “Greater than” if the number is greater than 100. For that we have to use the *else* statement. This works when the *if* statement is not fulfilled.

```
#!/usr/bin/env python3
number = int(input("Enter a number: "))
if number < 100:
    print("The number is less than 100")
else:
    print("The number is greater than 100")
```

The output

```
$ ./number100.py
Enter a number: 345
The number is greater than 100
```

Another very basic example

```
>>> x = int(input("Please enter an integer: "))
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
```

Truth value testing

The elegant way to test Truth values is like

```
if x:
    pass
```

Warning

Don't do this

```
if x == True:
    pass
```

New in version 3.10.

match statements

From Python 3.10 we have match statements. We can use these instead of if/elif/else blocks. For example, in the following code, we are taking an integer as input and then matching the value with some predefined HTTP status codes and print the names.

```
status = int(input("Give us a status code: "))
match status:
    case 200:
        print("OK")
    case 201:
        print("Created")
    case 301:
        print("Moved Permanently")
    case 302:
        print("Found")
    case _:
        print("The status is unknown to us.")
```

The last line, we are matching against anything else, we call it *wildcard* matching.

We can even match complex objects, say against tuples, or other objects.

```
status = (100,500)
match status:
    case (x,y):
        print(f"X and Y are: {x}, {y}")
    case x:
        print(x)

status = 42
match status:
    case (x,y):
        print(f"X and Y are: {x}, {y}")
    case x:
        print(x)
```

Learning using hardware simulation

From this section and in the future chapters, there will be examples and problems which is using a special device simulation in VS Code editor.

We will use the simulation to have a Circuit Playground Express device [<https://adafruit.com/product/3333>], and learn to interact with the device. We will be able to turn on lights of different colours, and many other nice examples.

Note

Visit Adafruit [<https://www.adafruit.com>] website to find many amazing hardware and learning materials.

Circuit Playground Express

The above is the image of the real hardware, in the simulation we will use, we can use the following:

- Green LED
- Red LED
- Push Buttons A and B
- Slider Switch
- Speaker: Play .wav file
- 10 NeoPixels
- Light sensor
- Motion sensors
- Acceleration detection
- Device shake detection

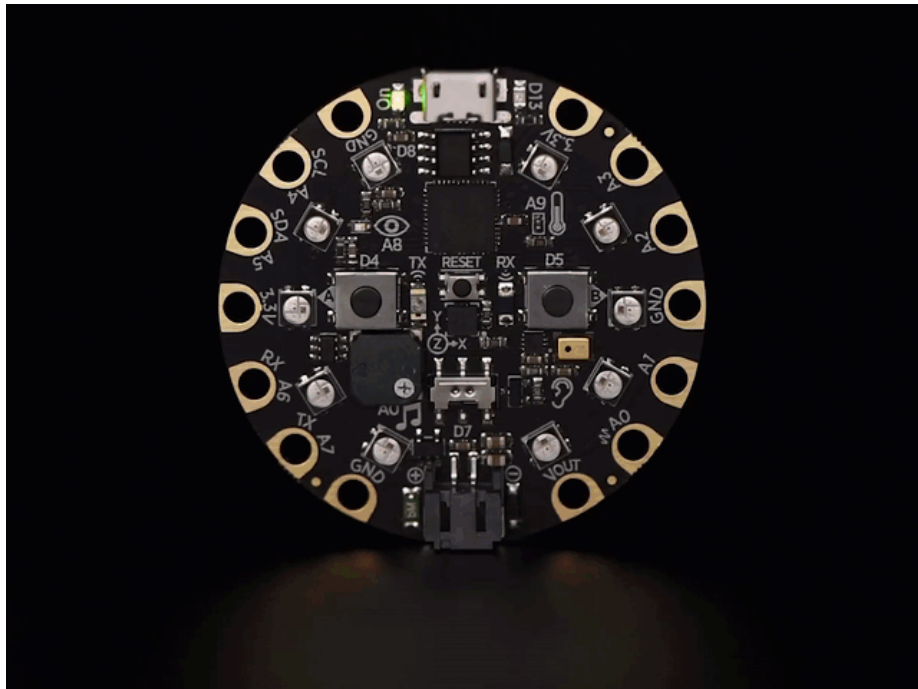


Figure 3:

TURNING ON ALL NEOPIXELS ON THE BOARD WITH RED COLOUR37

- Temperature sensor
- 7 Capacitive Touch sensors

See also

Go to the (Using VS Code as your primary Python editor) to learn how to install VS Code and the device simulation, and then come back to this chapter.

Turning on all NeoPixels on the board with Red colour

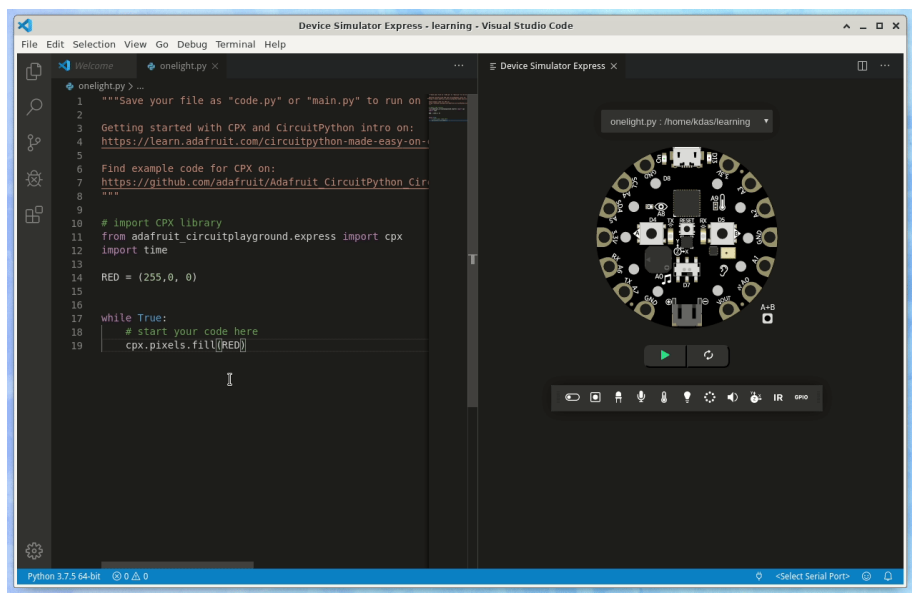


Figure 4:

```
# import CPX library
from adafruit_circuitplayground.express import cpx
import time
```

```
RED = (255, 0, 0)
```

```
while True:
    # start your code here
    cpx.pixels.fill(RED)
```

On the top of the code, we are writing a line starting with **from** to get the Python module which can interact with the device, as **cpx**. Next line, we are

also importing *time* module, which we will use later.

Then we defined a colour Red in the *RED* variable as a (R, G, B) tuple.

Next, the *while True:* line starts a loop and it will keep running, and inside using the same indentation, we call a function *cpx.pixels.fill* with the desired color (in this case RED).

Can you now write a code which will turn all the NeoPixel lights on the device Yellow? For this, you will have to search the RGB value of Yellow.

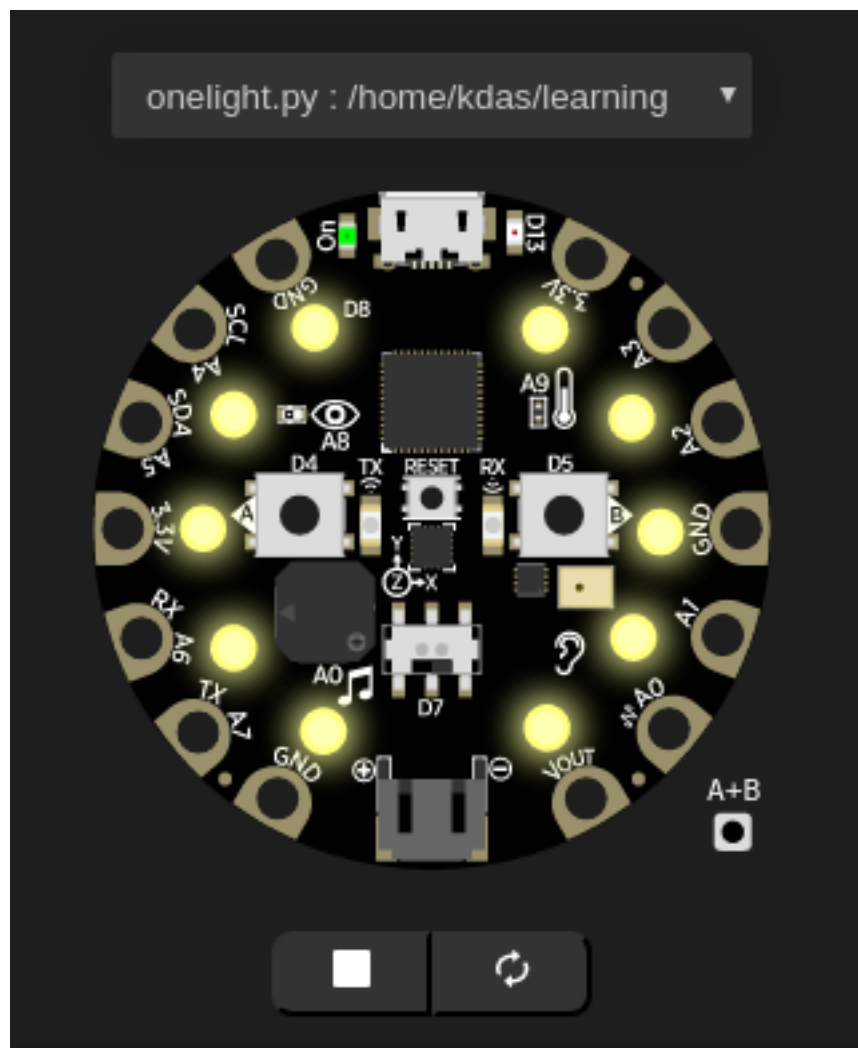


Figure 5:

Yellow light solution

```
from adafruit_circuitplayground.express import cpx
import time

YELLOW = (255, 255, 0)

while True:
    # start your code here
    cpx.pixels.fill(YELLOW)
```

Learning about time.sleep

The time module has an important function, *sleep*, which takes the amount of time the code will sleep (means it will do nothing), and then, the next instruction will execute. We can use this to keep the light on or off for certain amount of time. Take the following example, where we are keeping the NeoPixels on for 1 second and then, turning them off (RGB value (0,0,0)) for 0.5 seconds.

```
# import CPX library
from adafruit_circuitplayground.express import cpx
import time

WHITE = (255, 255, 255)
OFF = (0, 0, 0)

while True:
    # start your code here
    cpx.pixels.fill(WHITE)
    time.sleep(1)
    cpx.pixels.fill(OFF)
    time.sleep(0.5)
```

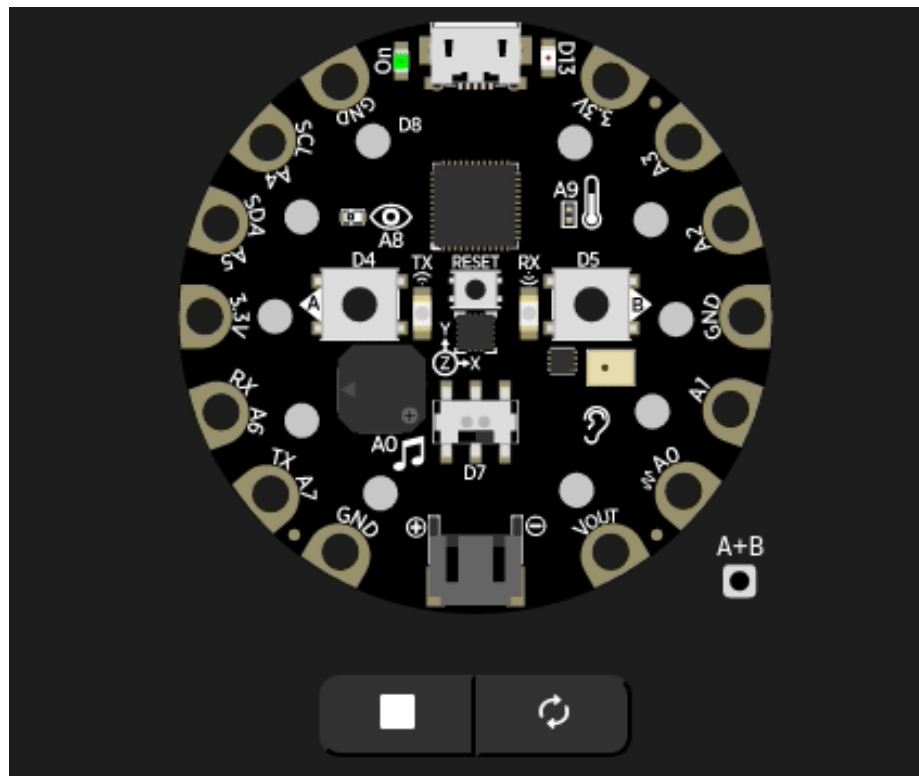
RGB problem

Can you modify the code in a such a way that it shows Red, and then Green and then Blue on all NeoPixels? It will look like the following image.

RGB solution

```
# import CPX library
from adafruit_circuitplayground.express import cpx
import time

RED = (255, 0, 0)
GREEN = (0, 255, 0)
```



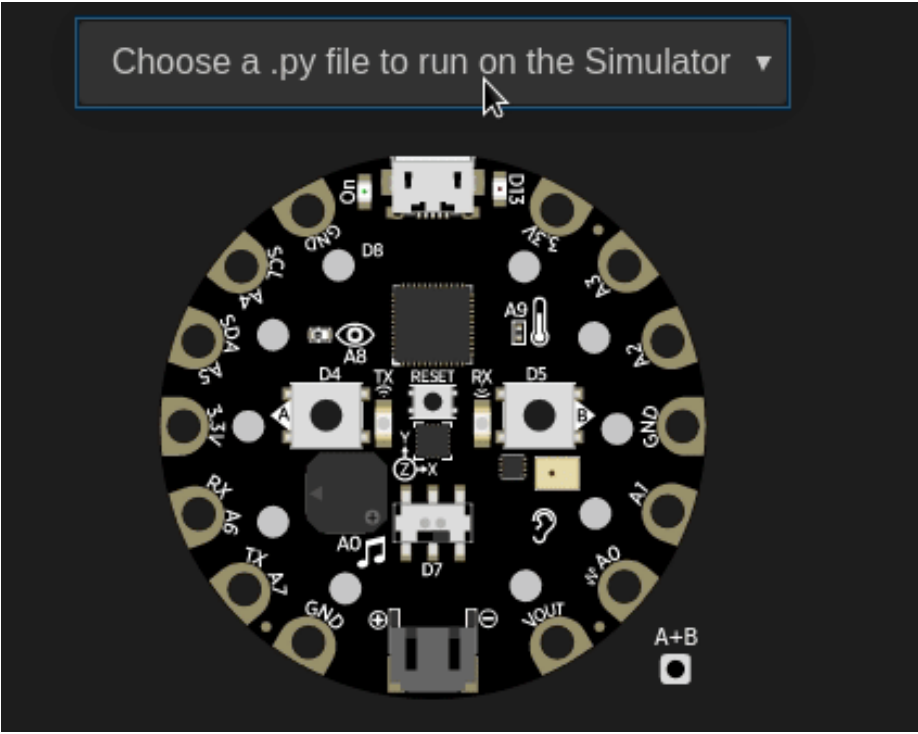


Figure 7:

```
BLUE = (0, 0, 255)
```

```
while True:
    cpx.pixels.fill(RED)
    time.sleep(1)
    cpx.pixels.fill(GREEN)
    time.sleep(1)
    cpx.pixels.fill(BLUE)
    time.sleep(1)
```

Looping

In the examples we used before, sometimes it was required to do the same work couple of times. We use a counter to check how many times the code needs to be executed. This technique is known as looping. First we are going to look into while statement for looping.

While loop

The syntax for *while* statement is like

```
while condition:
    statement1
    statement2
```

The code we want to reuse must be indented properly under the while statement. They will be executed if the *condition* is true. Again like in *if-else* statement any non zero value is true. Let us write a simple code to print numbers 0 to 10

```
>>> n = 0
>>> while n < 11:
...     print(n)
...     n += 1
...
0
1
2
3
4
5
6
7
8
9
10
```

In the first line we are setting $n = 0$, then in the while statement the condition

is $n < 11$, that means what ever line indented below that will execute until n becomes same or greater than 11. Inside the loop we are just printing the value of n and then increasing it by one.

How is this code going to help us in any real life? Think about the situation where you have to turn on 10 light bulbs one by one. May be you can run a loop from 1 to 10 and for each value on n , turn on the n th bulb.

Fibonacci Series

Let us try to solve *Fibonacci* series. In this series we get the next number by adding the previous two numbers. So the series looks like *1,1,2,3,5,8,13*

```
#!/usr/bin/env python3
a, b = 0, 1
while b < 100:
    print(b)
    a, b = b, a + b
```

The output

```
$ ./fibonacci1.py
1
1
2
3
5
8
13
21
34
55
89
```

In the first line of the code we are initializing a and b , then looping while b 's value is less than 100. Inside the loop first we are printing the value of b and then in the next line putting the value of b to a and $a + b$ to b in the same line.

In your print function call if you pass another argument called end and pass a space string, it will print in the same line with space delimiter. The default value for end is '\n'.

```
#!/usr/bin/env python3
a, b = 0, 1
while b < 100:
    print(b, end=' ')
    a, b = b, a + b
```

The output

```
$ ./fibonacci2.py
1 1 2 3 5 8 13 21 34 55 89
```

Power Series

Let us write a program to evaluate the power series. The series looks like $e^x = 1 + x + x^2/2! + x^3/3! + \dots + x^n/n!$ where $0 < x < 1$

```
#!/usr/bin/env python3
x = float(input("Enter the value of x: "))
n = term = num = 1
sum = 1.0
while n <= 100:
    term *= x / n
    sum += term
    n += 1
    if term < 0.0001:
        break
print("No of Times= %d and Sum= %f" % (n, sum))
```

The output

```
$ ./powerseries.py
Enter the value of x: 0
No of Times= 2 and Sum= 1.000000
$ ./powerseries.py
Enter the value of x: 0.1
No of Times= 5 and Sum= 1.105171
$ ./powerseries.py
Enter the value of x: 0.5
No of Times= 7 and Sum= 1.648720
```

In this program we introduced a new keyword called *break*. What *break* does is stop the innermost loop. In this example we are using *break* under the *if* statement

```
if term < 0.0001:
    break
```

This means if the value of *term* is less than *0.0001* then get out of the loop.

Multiplication Table

In this example we are going to print the multiplication table up to 10.

```
#!/usr/bin/env python3
i = 1
print("-" * 50)
```



```

while i < 11:
    n = 1
    while n <= 10:
        print("%4d" % (i * n), end=' ')
        n += 1
    print()
    i += 1
print("-" * 50)

```

The output

```
$ ./multiplication.py
```

```

-----
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
-----

```

Here we used one while loop inside another loop, this is known as nested looping. You can also see one interesting statement here

```
print("-" * 50)
```

In a *print* statement if we multiply the string with an integer *n*, the string will be printed *n* many times. Some examples

```

>>> print("*" * 10)
*****
>>> print("#" * 20)
#####
>>> print("--" * 20)
-----

```

```

>>> print("-" * 40)
-----

```

Some printing * examples

Here are some examples which you can find very often in college lab reports

Design 1

```
#!/usr/bin/env python3
```

```

row = int(input("Enter the number of rows: "))
n = row
while n >= 0:
    x = "*" * n
    print(x)
    n -= 1

```

The output

```

$ ./design1.py
Enter the number of rows: 5
*****
****
***
**
*

```

Design 2

```

#!/usr/bin/env python3
n = int(input("Enter the number of rows: "))
i = 1
while i <= n:
    print("*" * i)
    i += 1

```

The output

```

$ ./design2.py
Enter the number of rows: 5
*
**
***
****
*****

```

Design 3

```

#!/usr/bin/env python3
row = int(input("Enter the number of rows: "))
n = row
while n >= 0:
    x = "*" * n
    y = " " * (row - n)
    print(y + x)
    n -= 1

```

The output

```

$ ./design3.py
Enter the number of rows: 5

```

```
*****
****
***
**
*
```

Lists

We are going to learn a data structure called list before we go ahead to learn more on looping. Lists can be written as a list of comma-separated values (items) between square brackets.

```
>>> a = [ 1, 342, 223, 'India', 'Fedora']
>>> a
[1, 342, 223, 'India', 'Fedora']
```

You can imagine the above as a list of boxes, each box contains the value mentioned. We can also access value of each box by using the index number (mentioned in red numbers). You can understand that the index starts with 0.



Now we can access the values of each box by using the index number.

```
>>> a[0]
1
>>> a[4]
'Fedora'
```

If we use a negative number as index, that means we are counting from the end of the list. Like

```
>>> a[-1]
'Fedora'
```

You can even slice it into different pieces, examples are given below

```
>>> a[4]
'Fedora'
>>> a[-1]
'Fedora'
>>> a[-2]
'India'
```

```
>>> a[0:-1]
[1, 342, 223, 'India']
>>> a[2:-2]
[223]
>>> a[: -2]
[1, 342, 223]
>>> a[0:2]
[1, 223, 'Fedora']
```

In the last example we used two `:(s)` , the last value inside the third brackets indicates step. `s[i:j:k]` means slice of `s` from `i` to `j` with step `k`.

To check if any value exists within the list or not you can do

```
>>> a = ['Fedora', 'is', 'cool']
>>> 'cool' in a
True
>>> 'Linux' in a
False
```

That means we can use the above statement as *if* clause expression. The built-in function `len()` can tell the length of a list.

```
>>> len(a)
3
```

Note

If you want to test if the list is empty or not, do it like this

```
if list_name: # This means the list is not empty
    pass
else: # This means the list is empty
    pass
```

For loop

There is another to loop by using *for* statement. In Python the *for* statement is different from the way it works in *C*. Here *for* statement iterates over the items of any sequence (a list or a string). Example given below

```
>>> a = ['Fedora', 'is', 'powerful']
>>> for x in a:
...     print(x)
...
Fedora
is
powerful
```

We can also do things like

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> for x in a[::2]:
...     print(x)
1
3
5
7
9
```

range() function

range() is a builtin class. From the help document

```
class range(object) range(stop) -> range object
        range(start, stop[, step]) -> range object
```

Return a virtual sequence of numbers from start to stop by step.

Methods defined here:

Now if you want to see this help message on your system type *help(range)* in the Python interpreter. *help(s)* will return help message on the object *s*. Examples of *range* function

```
>>> list(range(1, 5))
[1, 2, 3, 4]
>>> list(range(1, 15, 3))
[1, 4, 7, 10, 13]
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Continue statement

Just like *break* we have another statement, *continue*, which skips the execution of the code after itself and goes back to the start of the loop. That means it will help you to skip a part of the loop. In the below example we will ask the user to input an integer, if the input is negative then we will ask again, if positive then we will square the number. To get out of the infinite loop user must input 0.

```
#!/usr/bin/env python3
while True:
```

```

n = int(input("Please enter an Integer: "))
if n < 0:
    continue # this will take the execution back to the starting of the loop
elif n == 0:
    break
print("Square is ", n ** 2)
print("Goodbye")

```

The output

```

$ ./continue.py
Please enter an Integer: 34
Square is 1156
Please enter an Integer: 4
Square is 16
Please enter an Integer: -9
Please enter an Integer: 0
Goodbye

```

Else statement in a loop

We can have an optional *else* statement after any loop. It will be executed after the loop unless a *break* statement stopped the loop.

```

>>> for i in range(0, 5):
...     print(i)
... else:
...     print("Bye bye")
...
0
1
2
3
4
Bye bye

```

We will see more example of *break* and *continue* later in the book.

Game of sticks

This is a very simple game of sticks. There are 21 sticks, first the user picks number of sticks between 1-4, then the computer picks sticks(1-4). Who ever will pick the last stick will lose. Can you find out the case when the user will win ?

```

#!/usr/bin/env python3
sticks = 21

```

```
print("There are 21 sticks, you can take 1-4 number of sticks at a time.")
print("Whoever will take the last stick will lose")

while True:
    print("Sticks left: " , sticks)
    sticks_taken = int(input("Take sticks(1-4):"))
    if sticks == 1:
        print("You took the last stick, you lose")
        break
    if sticks_taken >= 5 or sticks_taken <= 0:
        print("Wrong choice")
        continue
    print("Computer took: " , (5 - sticks_taken) , "\n")
    sticks -= 5
```

Looping over hardware

In this chapter we learn about looping using hardware (simulation).

One NeoPixel at a time

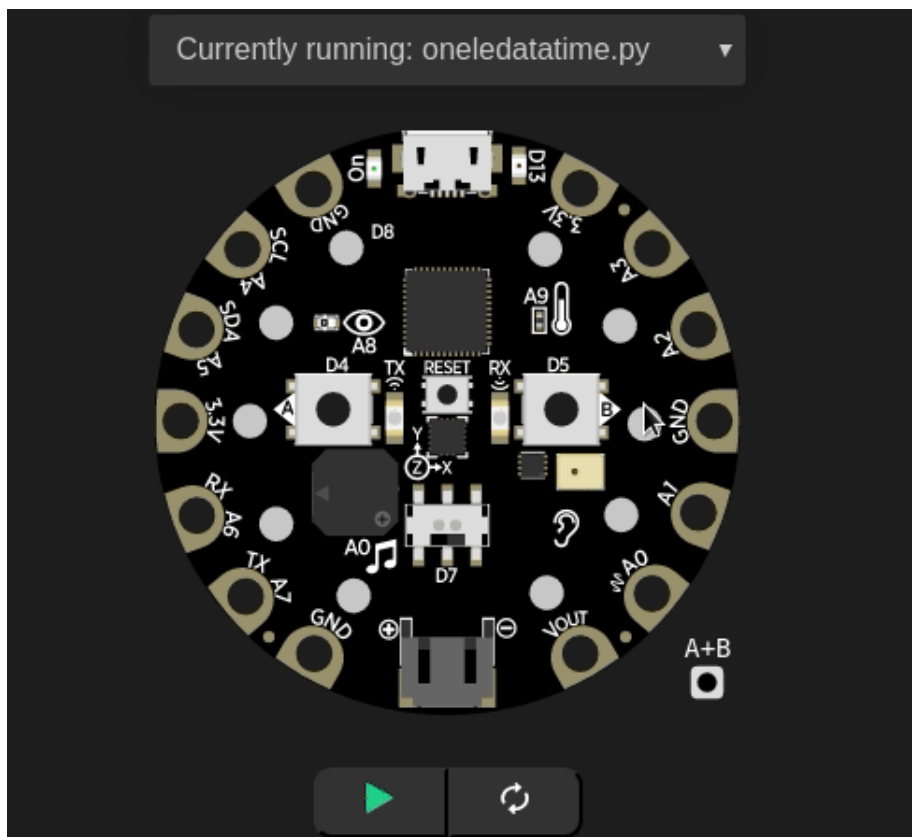


Figure 8:

The `cpx.pixels` can be accessed by index numbers, from 0 to 9. This way, we can turn on one NeoPixel at a time.

```

1# import CPX library
2from adafruit_circuitplayground.express import cpx
3import time
4
5BLUE = (0, 0, 255)
6
7i = 0
8
9while True:
10     # start your code here
11     if i == 10:
12         time.sleep(10)
13         continue
14     cpx.pixels[i] = BLUE
15     i += 1
16     time.sleep(0.5)

```

On line number 7, we declared a variable which selects the correct NeoPixel, and then on line number 14 we are turning on that NeoPixel to Blue color, and then increasing the variable to go to the next NeoPixel. The conditional *if statement* on line 11 makes sure that when we turn on all the lights, we sleep for 10 seconds and continue to do so.

First Red and then Blue

Here using two for loops, we are first turning on each NeoPixel in Red and then in Blue.

```

1# import CPX library
2from adafruit_circuitplayground.express import cpx
3import time
4
5RED = (255, 0, 0)
6BLUE = (0, 0, 255)
7
8while True:
9     for i in range(0, 10):
10         cpx.pixels[i] = RED
11         time.sleep(0.5)
12
13     time.sleep(0.5)
14
15     for i in range(0, 10):

```



```
16     cpx.pixels[i] = BLUE
17     time.sleep(0.5)
```

Data Structures

Python has a few built-in data structures. If you are wondering what a data structure is, it is nothing but a way to store data and having particular methods to retrieve or manipulate it. We already encountered lists before; now we will go in some depth.

Lists

```
>>> a = [23, 45, 1, -3434, 43624356, 234]
>>> a.append(45)
>>> a
[23, 45, 1, -3434, 43624356, 234, 45]
```

At first we created a list *a*. Then to add *45* at the end of the list we call the *a.append(45)* method. You can see that *45* is added at the end of the list. Sometimes it is necessary to insert data at any place within the list, for that we have *insert()* method.

```
>>> a.insert(0, 1) # 1 added at the 0th position of the list
>>> a
[1, 23, 45, 1, -3434, 43624356, 234, 45]
>>> a.insert(0, 111)
>>> a
[111, 1, 23, 45, 1, -3434, 43624356, 234, 45]
```

count(s) will return you the number of times *s* is in the list. Here we are going to check how many times *45* is there in the list.

```
>>> a.count(45)
2
```

If you want to remove any particular value from the list you have to use the *remove()* method.

```
>>> a.remove(234)
>>> a
[111, 1, 23, 45, 1, -3434, 43624356, 45]
```

Now to reverse the whole list

```
>>> a.reverse()
>>> a
[45, 43624356, -3434, 1, 45, 23, 1, 111]
```

We can store anything in the list, so first we are going to add another list *b* in *a*; then we will learn how to add the values of *b* into *a*.

```
>>> b = [45, 56, 90]
>>> a.append(b)
>>> a
[45, 43624356, -3434, 1, 45, 23, 1, 111, [45, 56, 90]]
>>> a[-1]
[45, 56, 90]
>>> a.extend(b) # To add the values of b not the b itself
>>> a
[45, 43624356, -3434, 1, 45, 23, 1, 111, [45, 56, 90], 45, 56, 90]
>>> a[-1]
90
```

Above you can see how we used the *a.extend()* method to extend the list. To sort any list we have *sort()* method. The *sort()* method will only work if elements in the list are comparable. We will remove the list *b* from the list and then sort.

```
>>> a.remove(b)
>>> a
[45, 43624356, -3434, 1, 45, 23, 1, 111, 45, 56, 90]
>>> a.sort()
>>> a
[-3434, 1, 1, 23, 45, 45, 45, 56, 90, 111, 43624356]
```

Note

Remember that sort method does not return the sorted list, instead the list itself will be sorted. This is done to keep performance of the code in mind. More details can be found here [<https://docs.python.org/3/faq/design.html?highlight=walrus#why-doesn-t-list-sort-return-the-sorted-list>].

You can also delete an element at any particular position of the list using the *del* keyword.

```
>>> del a[-1]
>>> a
[-3434, 1, 1, 23, 45, 45, 45, 56, 90, 111]
```

Using lists as stack and queue

Stacks are often known as LIFO (Last In First Out) structure. It means the data will enter into it at the end, and the last data will come out first. The easiest example can be of couple of marbles in an one side closed pipe. So if you want to take the marbles out of it you have to do that from the end where you inserted the last marble. To achieve the same in code

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> a
[1, 2, 3, 4, 5, 6]
>>> a.pop()
6
>>> a.pop()
5
>>> a.pop()
4
>>> a.pop()
3
>>> a
[1, 2]
>>> a.append(34)
>>> a
[1, 2, 34]
```

We learned a new method above *pop()*. *pop(i)* will take out the *i*th data from the list.

In our daily life we have to encounter queues many times, like at ticket counters or in the library or in the billing section of any supermarket. Queue is the data structure where you can append more data at the end and take out data from the beginning. That is why it is known as FIFO (First In First Out).

```
>>> a = [1, 2, 3, 4, 5]
>>> a.append(1)
>>> a
[1, 2, 3, 4, 5, 1]
>>> a.pop(0)
1
>>> a.pop(0)
2
>>> a
[3, 4, 5, 1]
```

To take out the first element of the list we are using *a.pop(0)*.

List Comprehensions

List comprehensions provide a concise way to create lists. Each list comprehension consists of an expression followed by a for clause, then zero or more for or if clauses. The result will be a list resulting from evaluating the expression in the context of the for and if clauses which follow it.

For example if we want to make a list out of the square values of another list, then

```
>>> a = [1, 2, 3]
>>> [x ** 2 for x in a]
[1, 4, 9]
>>> z = [x + 1 for x in [x ** 2 for x in a]]
>>> z
[2, 5, 10]
```

Above in the second case we used two list comprehensions in a same line.

A few special functions related to lists

sum takes a list as argument and sums all the values inside of it.

```
>>> numbers = [1, 2, 3, 4, 5]
>>> sum(numbers)
15
```

min and max tells us about the minimum and maximum value from a list.

```
>>> min(numbers)
1
>>> max(numbers)
5
```

There are two other functions related to boolean value checking. any and all. any tells us if any of the value in the list is True. all tells us if all the values in the list are True or not.

```
>>> numbers = [1, 2, 3, 4, 5, 0]
>>> any(numbers)
True
>>> all(numbers)
False
```

Here all returned False because we have 0 in the list.

Tuples

Tuples are data separated by commas.

```
>>> a = 'Fedora', 'Debian', 'Kubuntu', 'Pardus'
>>> a
('Fedora', 'Debian', 'Kubuntu', 'Pardus')
>>> a[1]
'Debian'
>>> for x in a:
...     print(x, end=' ')
...
Fedora Debian Kubuntu Pardus
```

You can also unpack values of any tuple into variables, like

```
>>> divmod(15,2)
(7, 1)
>>> x, y = divmod(15,2)
>>> x
7
>>> y
1
```

Tuples are immutable, meaning that you can not del/add/edit any value inside the tuple. Here is another example

```
>>> a = (1, 2, 3, 4)
>>> del a[0]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
```

As you can see above, Python gives an error when we try to delete a value in the tuple.

To create a tuple which contains only one value, type a trailing comma.

```
>>> a = (123)
>>> a
123
>>> type(a)
<class 'int'>
>>> a = (123, ) #Look at the trailing comma
>>> a
(123,)
>>> type(a)
<class 'tuple'>
```

Using the built-in function *type()* you can know the data type of any variable. Remember the *len()* function we used to find the length of any sequence?

```
>>> type(len)
<class 'builtin_function_or_method'>
```


Sets

Sets are another type of data structure with no duplicate items. We can apply mathematical set operations on sets.

```
>>> a = set('abcthabcjhethddda')
>>> a
{'a', 'c', 'b', 'e', 'd', 'h', 'j', 't', 'w'}
```

And some examples of the set operations

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                     # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                                 # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                                 # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                                 # letters in both a and b
{'a', 'c'}
>>> a ^ b                                 # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

To add or pop values from a set

```
>>> a
{'a', 'c', 'b', 'e', 'd', 'h', 'j', 'q', 't', 'w'}
>>> a.add('p')
>>> a
{'a', 'c', 'b', 'e', 'd', 'h', 'j', 'q', 'p', 't', 'w'}
```

Dictionaries

Dictionaries are unordered set of *key: value* pairs where keys are unique. We declare dictionaries using {} braces. We use dictionaries to store data for any particular key and then retrieve them.

```
>>> data = {'kushal': 'Fedora', 'kart_': 'Debian', 'Jace': 'Mac'}
>>> data
{'kushal': 'Fedora', 'Jace': 'Mac', 'kart_': 'Debian'}
>>> data['kart_']
'Debian'
```

We can add more data to it by simply

```
>>> data['parthan'] = 'Ubuntu'
>>> data
{'kushal': 'Fedora', 'Jace': 'Mac', 'kart_': 'Debian', 'parthan': 'Ubuntu'}
```

To delete any particular *key:value* pair

```
>>> del data['kushal']
>>> data
{'Jace': 'Mac', 'kart_': 'Debian', 'parthan': 'Ubuntu'}
```

To check if any *key* is there in the dictionary or not you can use *in* keyword.

```
>>> 'Soumya' in data
False
```

You must remember that no mutable object can be a *key*, that means you can not use a *list* as a *key*.

dict() can create dictionaries from tuples of *key,value* pair.

```
>>> dict((( 'Indian', 'Delhi'), ('Bangladesh', 'Dhaka')))
{'Indian': 'Delhi', 'Bangladesh': 'Dhaka'}
```

Many times it happens that we want to add more data to a value in a dictionary and if the key does not exists then we add some default value. You can do this efficiently using *dict.setdefault(key, default)*.

```
>>> data = {}
>>> data.setdefault('names', []).append('Ruby')
>>> data
{'names': ['Ruby']}
>>> data.setdefault('names', []).append('Python')
>>> data
{'names': ['Ruby', 'Python']}
>>> data.setdefault('names', []).append('C')
>>> data
{'names': ['Ruby', 'Python', 'C']}
```

When we try to get value for a key which does not exists we get *KeyError*. We can use *dict.get(key, default)* to get a default value when they key does not exists before.

```
>>> data['foo']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'foo'
>>> data.get('foo', 0)
0
```

Looping over a dictionary

If you just do a for loop over a dictionary, it will provide you all the available keys in the dictionary.

```
>>> data
```

```
{'Kushal': 'Fedora', 'Jace': 'Mac', 'kart_': 'Debian', 'parthan': 'Ubuntu'}
>>> for x in data:
...     print(f"Key = {x}")
...
Kushal
Jace
kart_
parthan
```

If you want to loop through a dict use *items()* method.

```
>>> data
{'Kushal': 'Fedora', 'Jace': 'Mac', 'kart_': 'Debian', 'parthan': 'Ubuntu'}
>>> for x, y in data.items():
...     print("%s uses %s" % (x, y))
...
Kushal uses Fedora
Jace uses Mac
kart_ uses Debian
parthan uses Ubuntu
```

From Python 3.7 dictionaries maintain the insertion order of the items. While looping over *items()* method, you will get the key/value combination based on the insertion order of those items.

If you want to loop through a list (or any sequence) and get iteration number at the same time you have to use *enumerate()*.

```
>>> for i, j in enumerate(['a', 'b', 'c']):
...     print(i, j)
...
0 a
1 b
2 c
```

If you want to start the sequence from a different number, you can do that using *start* value in the *enumerate* function.

```
>>> for i, j in enumerate(['a', 'b', 'c'], start=1):
...     print(i, j)
...
1 a
2 b
3 c
```

You may also need to iterate through two sequences same time, for that use *zip()* function.

```
>>> a = ['Pradeepto', 'Kushal']
>>> b = ['OpenSUSE', 'Fedora']
```

```
>>> for x, y in zip(a, b):
...     print("%s uses %s" % (x, y))
...
Pradeepto uses OpenSUSE
Kushal uses Fedora
```

students.py

In this example , you have to take number of students as input , then ask marks for three subjects as 'Physics', 'Maths', 'History', if the total marks for any student is less 120 then print he failed, or else say passed.

```
#!/usr/bin/env python3
n = int(input("Enter the number of students:"))
data = {} # here we will store the data
languages = ('Physics', 'Maths', 'History') #all languages
for i in range(0, n): #for the n number of students
    name = input('Enter the name of the student %d: ' % (i + 1)) #Get the name of the student
    marks = []
    for x in languages:
        marks.append(int(input('Enter marks of %s: ' % x))) #Get the marks for languages
    data[name] = marks
for x, y in data.items():
    total = sum(y)
    print("%s 's total marks %d" % (x, total))
    if total < 120:
        print("%s failed :(" % x)
    else:
        print("%s passed :)" % x)
```

The output

```
$ ./students.py
Enter the number of students:2
Enter the name of the student 1: Babai
Enter marks of Physics: 12
Enter marks of Maths: 45
Enter marks of History: 40
Enter the name of the student 2: Tesla
Enter marks of Physics: 99
Enter marks of Maths: 98
Enter marks of History: 99
Babai 's total marks 97
Babai failed :(
Tesla 's total marks 296
Tesla passed :)
```

Assignment problems

Remember that in Python, variables are just names pointing to the value. In the following example, both `x` and `y` points to same value, means when `x` changes, `y` also changes.

```
>>> x = [1, 2, 3]
>>> y = x
>>> x.append(20)
>>> print(x)
[1, 2, 3, 20]
>>> print(y)
[1, 2, 3, 20]
```

If you want a full copy of the data assigned to a new variable, call `obj.copy()` method.

This also happens when you pass them into functions. For example, in the below function, we are passing a list, and appending new numbers into it. This also changes the variable outside of the function.

```
numbers = [1, 2, 4]

def modify(numbers):
    numbers.append(42)

modify(numbers)
print(numbers)
[1, 2, 4, 42]
```

Note

Please go through the documentation for the mentioned data types in docs.python.org, at least have a look at the different methods available on them. For most of the day to day work, we use these data types in our Python code. So, it is important to understand them well. Before you move into next chapter, read the help documentation if nothing else. For example: `help(dict)` or `help(list)`.

Create a calculator

Here is a small problem for you. You will have to use list, and dictionary to create a tool, which will take input like `(* (+ 3 4) 2)` and return the answer like 14. The four valid operators are `+`, `-`, `/*`. Every operator will need two operands to work on. Another input `(* 2 3)` and the output is 6.

Strings

Strings are nothing but simple text. In Python we declare strings in between “” or ‘ ’ or “ ” or “ ” “ ”. The examples below will help you to understand string in a better way.

```
>>> s = "I love Python "  
>>> s  
'I love Python '  
>>> s = 'I love Python '  
>>> s = "Here is a line \  
... split in two lines"  
>>> s  
'Here is a line split in two lines'  
>>> s = "Here is a line \n split in two lines"  
>>> s  
'Here is a line \n split in two lines'  
>>> print(s)  
Here is a line  
    split in two lines
```

Now if you want to multiline strings you have to use triple single/double quotes.

```
>>> s = """ This is a  
... multiline string, so you can  
... write many lines"""  
>>> print(s)  
This is a  
multiline string, so you can  
write many lines
```

We can have two string literals side by side, and it will behave like a single string. For example

```
>>> s = "Hello " "World"  
>>> print(s)  
Hello World
```

This will help you to split a long string into smaller chunks in function calls.

You can find length of any string using the `len` function call.

```
>>> s = "Python"
>>> len(s)
6
```

Different methods available for Strings

Every string object is having couple of builtin methods available, we already saw some of them like `s.split()`.

```
>>> s = "kushal das"
>>> s.title()
'Kushal Das'
```

`title()` method returns a titlecased version of the string, words start with uppercase characters, all remaining cased characters are lowercase.

```
>>> z = s.upper()
>>> z
'KUSHAL DAS'
>>> z.lower()
'kushal das'
```

`upper()` returns a total uppercase version whereas `lower()` returns a lower case version of the string.

```
>>> s = "I am A pRoGraMMer"
>> s.swapcase()
'i AM a PrOgRAmMER'
```

`swapcase()` returns the string with case swapped :)

```
>>> s = "jdwb 2323bjb"
>>> s.isalnum()
False
>>> s = "jdwb2323bjb"
>>> s.isalnum()
True
```

Because of the space in the first line `isalnum()` returned `False`, it checks for all characters are alpha numeric or not.

```
>>> s = "SankarshanSir"
>>> s.isalpha()
True
>>> s = "Sankarshan Sir"
>>> s.isalpha()
```

False

isalpha() checks for only alphabets.

```
>>> s = "1234"
>>> s.isdigit() # To check if all the characters are digits or not
True
>>> s = "Fedora9 is coming"
>>> s.islower() # To check if all chracters are lower case or not
False
>>> s = "Fedora9 Is Coming"
>>> s.istitle() # To check if it is a title or not
True
>>> s = "INDIA"
>>> s.isupper() # To check if characters are in upper case or not
True
```

To split any string we have *split()*. It takes a string as an argument , depending on that it will split the main string and returns a list containing splitted strings.

```
>>> s = "We all love Python"
>>> s.split(" ")
['We', 'all', 'love', 'Python']
>>> x = "Nishant:is:waiting"
>>> x.split(':')
['Nishant', 'is', 'waiting']
```

The opposite method for *split()* is *join()*. It takes a list contains strings as input and join them.

```
>>> "-".join("GNU/Linux is great".split(" "))
'GNU/Linux-is-great'
```

In the above example first we are splitting the string “GNU/Linux is great” based on the white space, then joining them with “-”.

Strip the strings

Strings do have few methods to do striping. The simplest one is *strip(chars)*. If you provide the chars argument then it will strip any combination of them. By default it strips only whitespace or newline characters.

```
>>> s = "  abc\n "
>>> s.strip()
'abc'
```

You can particularly strip from the left hand or right hand side also using *lstrip(chars)* or *rstrip(chars)*.

```
>>> s = "www.foss.in"
```



```
>>> s.lstrip("cwsd.")
'foss.in'
>>> s.rstrip("cnwdi.")
'www.foss'
```

Justifying text

We can use `rjust` or `ljust` methods to either right justify or left justify any given string. We have to provide the number of characters we want to justify, and the character we want use for the justification (while a whitespace too).

```
>>> "ACAB".rjust(10, "-")
'-----ACAB'
>>> "ACAB".ljust(10, "-")
'ACAB-----'
```

Finding text

Strings have some methods which will help you in finding text/substring in a string. Examples are given below:

```
>>> s = "faulty for a reason"
>>> s.find("for")
7
>>> s.find("fora")
-1
>>> s.startswith("fa") #To check if the string startswith fa or not
True
>>> s.endswith("reason") #To check if the string endswith reason or not
True
```

find() helps to find the first occurrence of the substring given, if not found it returns -1.

Palindrome checking

Palindrome are the kind of strings which are same from left or right whichever way you read them. Example “madam”. In this example we will take the word as input from the user and say if it is palindrome or not.

```
#!/usr/bin/env python3
s = input("Please enter a string: ")
z = s[::-1]
if s == z:
    print("The string is a palindrome")
else:
```

```
print("The string is not a palindrome")
```

The output

```
$ ./palindrome.py
Please enter a string: madam1
The string is not a palindrome
$ ./palindrome.py
Please enter a string: madam
The string is a palindrome
```

Number of words

In this example we will count the number of words in a given line

```
#!/usr/bin/env python3
s = input("Enter a line: ")
print("The number of words in the line are %d" % (len(s.split(" "))))
```

The output

```
$ ./countwords.py
Enter a line: Sayamindu is a great programmer
The number of words in the line are 5
```

Iterating over all characters of a string

You can iterate over a string using simple for loop.

```
>>> for ch in "Python":
...     print(ch)
...
P
y
t
h
o
n
```

Functions

Reusing the same code is required many times within a same program. Functions help us to do so. We write the things we have to do repeatedly in a function then call it where ever required. We already saw build in functions like *len()*, *divmod()*.

Defining a function

We use *def* keyword to define a function. General syntax is like

```
def functionname(params):  
    statement1  
    statement2
```

Let us write a function which will take two integers as input and then return the sum.

```
>>> def sum(a, b):  
...     return a + b
```

In the second line with the *return* keyword, we are sending back the value of *a + b* to the caller. You must call it like

```
>>> res = sum(234234, 34453546464)  
>>> res  
34453780698L
```

What about a function to find if a given string is bigger than certain length or not? Yes, we can do it in one line, but also helps to learn about writing simple functions.

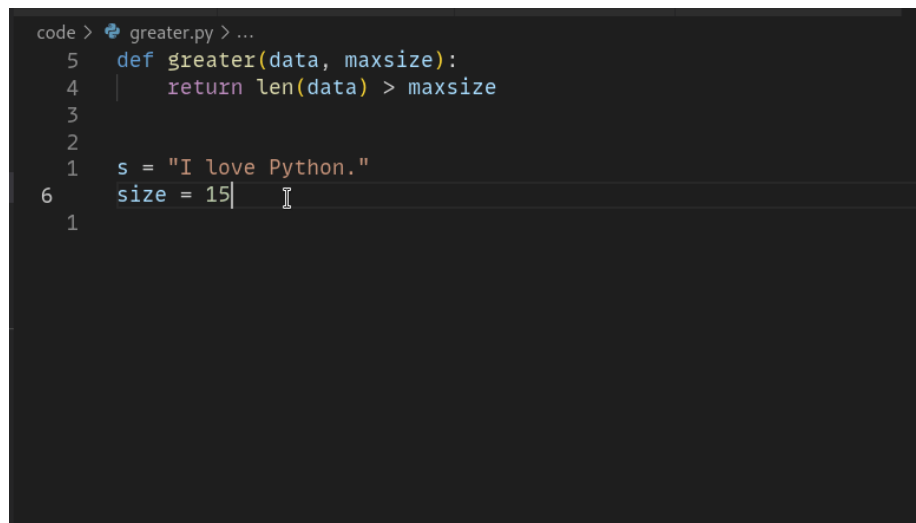
```
def greater(data, maxsize):  
    """Return if the length of data is bigger than given maxsize.  
    """  
    return len(data) > maxsize
```

Here the string just after the function definition is called docstring. We will learn about it more below in the chapter. For now we can try to use the function.

```
>>> s = "I am Kushal."
>>> print(greater(s, 10))
False
```

In our daily lives as programmers, we keep using various functions written by others in different Python modules. We stitch together those functions in our code and create our own functions, and then consume those functions in different parts of our code/project. Generally when you can see that you are repeating some amount of code again and again, you can start thinking about putting that into a function.

But while consuming those functions in Python, it is difficult to know what exactly the function arguments are, what kind of data they expect. We read the documentation to find out as much as we can, and then in many case that means continuously switching between the code and the documentation. Modern editors tries to help, for example let us see what VS Code shows us when we try to use the above greater function.



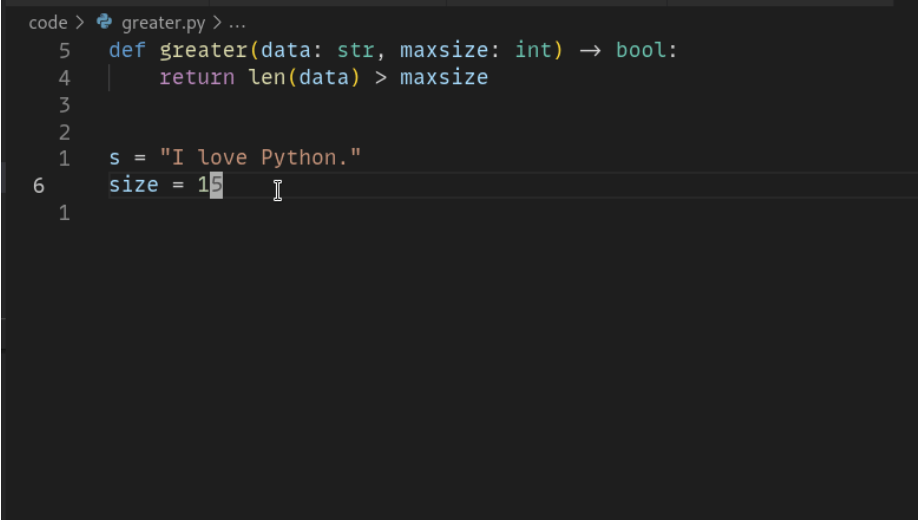
```
code > greater.py > ...
5  def greater(data, maxsize):
4  |     return len(data) > maxsize
3
2
1  s = "I love Python."
6  size = 15|
1
```

Here we don't know what kind of arguments we should pass in the greater function.

We can fix this with something called type annotation. Which increases the readability of our code and also helps the developer who will use the function in future. Automated tools in editors like VS Code (or even in Vim/Emacs). So we will rewrite our function as below.

```
def greater(data: str, maxsize: int) -> bool:
    """Return if the length of data is bigger than given maxsize.
    """
    return len(data) > maxsize
```

Here after the variable name and a colon we are mentioning what kind of data it is expecting and we are also mentioning the type of data returned by the function with the `->`. This is not checked during run time like normal typed programming languages, but there are special separate tools to help. But, this increases the readability of the code instantly. Now we know that we have to pass a string and an integer into the function. Even VS Code will tell us so.



```
code > greater.py > ...
5  def greater(data: str, maxsize: int) -> bool:
4  |     return len(data) > maxsize
3
2
1  s = "I love Python."
6  size = 15
1
```

Now if we add the annotation to our sum function, it will look like following.

```
def sum(a: int, b: int) -> int:
    return a + b
```

Remember the palindrome program we wrote in the last chapter. Let us write a function which will check if a given string is palindrome or not, then return *True* or *False*.

```
#!/usr/bin/env python3

def palindrome(s):
    return s == s[::-1]

if __name__ == "__main__":
    s = input("Enter a string: ")
    if palindrome(s):
        print("Yay a palindrome")
    else:
        print("Oh no, not a palindrome")
```

Now execute the file :)

Local and global variables

To understand local and global variables we will go through two examples.

```
#!/usr/bin/env python
```

```
def change(a):
    a = 90
    print(f"Inside of the change function {a}")
```

```
a = 9
print(f"Before the function call {a}")
change(a)
print(f"After the function call {a}")
```

The output

```
$ ./local.py
Before the function call 9
Inside of the change function 90
After the function call 9
```

First we are assigning *9* to *a*, then calling *change* function, inside of that we are assigning *90* to *a* and printing *a*. After the function call we are again printing the value of *a*. When we are writing *a = 90* inside the function, it is actually creating a new variable called *a*, which is only available inside the function and will be destroyed after the function finished. So though the name is same for the variable *a* but they are different in and out side of the function.

```
#!/usr/bin/env python3
def change(b):
    global a
    a = 90
    print(a)
a = 9
print("Before the function call ", a)
print("inside change function", end=' ')
change(a)
print("After the function call ", a)
```

Here by using *global* keyword we are telling that *a* is globally defined, so when we are changing *a*'s value inside the function it is actually changing for the *a* outside of the function also.

The output

```
$ ./local.py
Before the function call 9
```

```
inside change function 90
After the function call 90
```

Default argument value

In a function variables may have default argument values, that means if we don't give any value for that particular variable it will be assigned automatically.

```
>>> def test(a , b=-99):
...     if a > b:
...         return True
...     else:
...         return False
```

In the above example we have written $b = -99$ in the function parameter list. That means if no value for b is given then b 's value is -99 . This is a very simple example of default arguments. You can test the code by

```
>>> test(12, 23)
False
>>> test(12)
True
```

Important

Important

Remember that you can not have an argument without default argument if you already have one argument with default values before it. Like $f(a, b=90, c)$ is illegal as b is having a default value but after that c is not having any default value.

Also remember that default value is evaluated only once, so if you have any mutable object like list it will make a difference. See the next example

```
>>> def f(a, data=[]):
...     data.append(a)
...     return data
...
>>> print(f(1))
[1]
>>> print(f(2))
[1, 2]
>>> print(f(3))
[1, 2, 3]
```

To avoid this you can write more idiomatic Python, like the following

```
>>> def f(a, data=None):
...     if data is None:
```

```

...     data = []
...     data.append(a)
...     return data
...
>>> print(f(1))
[1]
>>> print(f(2))
[2]

```

Note

To understand more read this url [<https://docs.python.org/3/tutorial/controlflow.html#default-argument-values>].

Keyword arguments

```

>>> def func(a, b=5, c=10):
...     print('a is', a, 'and b is', b, 'and c is', c)
...
>>> func(12, 24)
a is 12 and b is 24 and c is 10
>>> func(12, c = 24)
a is 12 and b is 5 and c is 24
>>> func(b=12, c = 24, a = -1)
a is -1 and b is 12 and c is 24

```

In the above example you can see we are calling the function with variable names, like *func(12, c = 24)*, by that we are assigning 24 to *c* and *b* is getting its default value. Also remember that you can not have without keyword based argument after a keyword based argument. like

```

>>> def func(a, b=13, v):
...     print(a, b, v)
...
File "<stdin>", line 1
SyntaxError: non-default argument follows default argument

```

Keyword only argument

We can also mark the arguments of function as keyword only. That way while calling the function, the user will be forced to use correct keyword for each parameter.

```

>>> def hello(*, name: str = 'User'):
...     print("Hello %s" % name)
...
>>> hello('Kushal')

```



```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: hello() takes 0 positional arguments but 1 was given
>>> hello(name='Kushal')
Hello Kushal
```

Note

To learn more please read PEP-3102 [<https://www.python.org/dev/peps/pep-3102/>].

Positional only argument

From Python3.8, we can also mark any function to have only positional arguments. Write / at the end of all positional arguments in the function definition to have this feature.

```
>>> def add(a: int, b: int, /):
...     return a + b
...
>>> add(2, 3)
5
>>> add(a=2, b=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() got some positional-only arguments passed as keyword arguments: 'a, b'
```

You can see that when we tried to call the add with keyword arguments, it raised a `TypeError`.

Docstrings

In Python we use docstrings to explain how to use the code, it will be useful in interactive mode and to create auto-documentation. Below we see an example of the docstring for a function called *longest_side*.

```
#!/usr/bin/env python3
import math

def longest_side(a: int, b: int):
    """
    Function to find the length of the longest side of a right triangle.

    :arg a: Side a of the triangle
    :arg b: Side b of the triangle

    :return: Length of the longest side c as float
```

```

    """
    return math.sqrt(a*a + b*b)

if __name__ == '__main__':
    print(longest_side(4, 5))

```

We will learn more on docstrings in reStructuredText chapter.

Higher-order function

Higher-order function or a functor is a function which does at least one of the following step inside:

- Takes one or more functions as argument.
- Returns another function as output.

In Python any function can act as higher order function.

```

>>> def high(func, value):
...     return func(value)
...
>>> lst = high(dir, int)
>>> print(lst[-3:])
['imag', 'numerator', 'real']
>>> print(lst)

```

Let us look at another example. We will create a function, which in turn returns another function to add 5 to the given argument.

```

def givemefive():
    def add5(x: int) -> int:
        return x + 5
    return add5

>>> myadder = givemefive()
>>> print(myadder(10))
15
>>> print(type(myadder))
<class 'function'>

```

Here when we call givemefive, it is actually returning the function add5, and storing into myadder. Finally when we call the myadder, it adds 5 to the given argument, and returns it. We also printed the type of myadder

```

def givemeadder(num):
    def internal(x):
        return num + x
    return internal

```

```
>>> add10 = givemeadder(10)
>>> print(add10(20))
30
```

In this example the internal function is using the x variable from the outer scope. This is also known as closure where the function is using the environment enclosed. If we need a new function which will add 100 to the given number, we can do it easily like this.

```
add_big = givemeadder(100)
>>> print(add_big(1))
101
```

Note

To know more read this link [<http://docs.python.org/3/faq/programming.html#how-do-you-make-a-higher-order-function-in-python>].

map

map is a very useful class in python. It takes one function and an iterator as input and then applies the function on each value of the iterator and returns an iterator.

Example:

```
>>> lst = [1, 2, 3, 4, 5]
>>> def square(num: int) -> int:
...     "Returns the square of a given number."
...     return num * num
...
>>> print(list(map(square, lst)))
[1, 4, 9, 16, 25]
```

In Python2, map was a function and used to return list.

Parameters and arguments

Parameters are names defined in a function definition, and arguments are the actual values passed that function call.

```
def hello(name, age):
    return f"Hello {name}, you are {age} years old"

hello("kushal", 90)
```

In the above example, name and age are the parameters of the hello, and kushal and 90 are the arguments passed to the function.

***args and **kwargs in function definition**

There are times when we don't know the number of arguments before hand. There can be any number of positional or keyword arguments passed to the function. This is where we use *args and **kwargs in the function.

```
def unknown(*args, **kwargs):
    print(f"We received {len(args)} positional arguments. And they are:")
    for arg in args:
        print(arg, end= " ")
    print("")
    print(f"We received {len(kwargs)} keyword arguments. And they are:")
    for k, v in kwargs.items():
        print(f"key={k} and value={v}")
```

```
unknown(30, 90, "kushal", lang="python", editor="vim")
```

```
We received 3 positional arguments. And they are:
30 90 kushal
We received 2 keyword arguments. And they are:
key=lang and value=python
key=editor and value=vim
```

This is really helpful when you are writing code which will take another function as input, and you don't know about parameters of that function before hand. We will see more examples later in this book.

HOWTO Write a function

Watch this talk [<https://www.youtube.com/watch?v=rrBJVMYD-Gs>] by Jack Diederich at PyCon US 2018 to learn more about how to write clean Python functions and many other tips.

File handling

A file is some information or data which stays in the computer storage devices. You already know about different kinds of file, like your music files, video files, text files. Python gives you easy ways to manipulate these files. Generally we divide files in two categories, text file and binary file. Text files are simple text where as the binary files contain binary data which is only readable by computer.

File opening

To open a file we use *open()* function. It requires two arguments, first the file path or file name, second which mode it should open. Modes are like

- “r” -> open read only, you can read the file but can not edit / delete anything inside
- “w” -> open with write power, means if the file exists then delete all content and open it to write
- “a” -> open in append mode

The default mode is read only, ie if you do not provide any mode it will open the file as read only. Let us open a file

```
>>> fobj = open("love.txt")
>>> fobj
<_io.TextIOWrapper name='love.txt' mode='r' encoding='UTF-8'>
```

Closing a file

After opening a file one should always close the opened file. We use method *close()* for this.

```
>>> fobj = open("love.txt")
>>> fobj
<_io.TextIOWrapper name='love.txt' mode='r' encoding='UTF-8'>
```

```
>>> fobj.close()
```

Important

Always make sure you *explicitly* close each open file, once its job is done and you have no reason to keep it open. Because

- There is an upper limit to the number of files a program can open. If you exceed that limit, there is no reliable way of recovery, so the program could crash.
- Each open file consumes some main-memory for the data-structures associated with it, like file descriptor/handle or file locks etc. So you could essentially end-up wasting lots of memory if you have more files open that are not useful or usable.
- Open files always stand a chance of corruption and data loss.

Reading a file

To read the whole file at once use the *read()* method.

```
>>> fobj = open("sample.txt")
>>> fobj.read()
'I love Python\nPradeepto loves KDE\nSankarshan loves Openoffice\n'
```

If you call *read()* again it will return empty string as it already read the whole file. *readline()* can help you to read one line each time from the file.

```
>>> fobj = open("sample.txt")
>>> fobj.readline()
'I love Python\n'
>>> fobj.readline()
'Pradeepto loves KDE\n'
```

To read all the lines in a list we use *readlines()* method.

```
>>> fobj = open("sample.txt")
>>> fobj.readlines()
['I love Python\n', 'Pradeepto loves KDE\n', 'Sankarshan loves Openoffice\n']
```

You can even loop through the lines in a file object.

```
>>> fobj = open("sample.txt")
>>> for x in fobj:
...     print(x, end=' ')
...
I love Python
Pradeepto loves KDE
Sankarshan loves Openoffice
```

Let us write a program which will take the file name as the input from the user and show the content of the file in the console.

```
#!/usr/bin/env python3
name = input("Enter the file name: ")
fobj = open(name)
print(fobj.read())
fobj.close()
```

In the last line you can see that we closed the file object with the help of close() method.

The output

```
$ ./showfile.py
Enter the filename: sample.txt
I love Python
Pradeepto loves KDE
Sankarshan loves Openoffice
```

Using the with statement

In real life scenarios we should try to use with statement. It will take care of closing the file for you.

```
>>> with open('setup.py') as fobj:
...     for line in fobj:
...         print line,
...
#!/usr/bin/env python3
"""Factorial project"""
from setuptools import find_packages, setup

setup(name = 'factorial',
      version = '0.1',
      description = "Factorial module.",
      long_description = "A test module for our book.",
      platforms = ["Linux"],
      author="Kushal Das",
      author_email="kushaldas@gmail.com",
      url="https://pymbook.readthedocs.io/en/latest/",
      license = "http://www.gnu.org/copyleft/gpl.html",
      packages=find_packages()
    )
```

Writing in a file

Let us open a file then we will write some random text into it by using the *write()* method. We can also pass the file object to the print function call, so that it writes in the file.

```
>>> fobj = open("ircnicks.txt", 'w')
>>> fobj.write('powerpork\n')
>>> fobj.write('indrag\n')
>>> fobj.write('mishti\n')
>>> fobj.write('sankarshan')
>>> print("This is the last line.", file=fobj)
>>> fobj.close()
```

Now read the file we just created

```
>>> fobj = open('ircnicks.txt')
>>> s = fobj.read()
>>> print(s)
powerpork
indrag
mishti
sankarshan
This is the last line.
```

copyfile.py

In this example we will copy a given text file to another file.

```
#!/usr/bin/env python3
import sys
if len(sys.argv) < 3:
    print("Wrong parameter")
    print("./copyfile.py file1 file2")
    sys.exit(1)
with open(sys.argv[1]) as f1:
    s = f1.read()
with open(sys.argv[2], 'w') as f2:
    f2.write(s)
```

Note

This way of reading file is not always a good idea, a file can be very large to read and fit in the memory. It is always better to read a known size of the file and write that to the new file.

You can see we used a new module here *sys*. *sys.argv* contains all command line parameters. Remember *cp* command in shell, after *cp* we type first the file to be copied and then the new file name.

The first value in *sys.argv* is the name of the command itself.

```
#!/usr/bin/env python3
import sys
print("First value", sys.argv[0])
print("All values")
for i, x in enumerate(sys.argv):
    print(i, x)
```

The output

```
$ ./argvtest.py Hi there
First value ./argvtest.py
All values
0 ./argvtest.py
1 Hi
2 there
```

Here we used a new function *enumerate(iterableobject)*, which returns the index number and the value from the iterable object.

Count spaces, tabs and new lines in a file

Let us try to write an application which will count the spaces, tabs, and lines in any given file.

```
#!/usr/bin/env python3

import os
import sys

def parse_file(path):
    """
    Parses the text file in the given path and returns space, tab & new line
    details.

    :arg path: Path of the text file to parse

    :return: A tuple with count of spacaes, tabs and lines.
    """
    fd = open(path)
    i = 0
    spaces = 0
    tabs = 0
    for i, line in enumerate(fd):
        spaces += line.count(' ')
```

```

        tabs += line.count('\t')
    #Now close the open file
    fd.close()

    #Return the result as a tuple
    return spaces, tabs, i + 1

def main(path):
    """
    Function which prints counts of spaces, tabs and lines in a file.

    :arg path: Path of the text file to parse
    :return: True if the file exists or False.
    """
    if os.path.exists(path):
        spaces, tabs, lines = parse_file(path)
        print("Spaces %d. tabs %d. lines %d" % (spaces, tabs, lines))
        return True
    else:
        return False

if __name__ == '__main__':
    if len(sys.argv) > 1:
        main(sys.argv[1])
    else:
        sys.exit(-1)
    sys.exit(0)

```

You can see that we have two functions in the program , *main* and *parse_file* where the second one actually parses the file and returns the result and we print the result in *main* function. By splitting up the code in smaller units (functions) helps us to organize the codebase and also it will be easier to write test cases for the functions.

Let us write some real code

Do you know how many CPU(s) are there in your processor? or what is the model name? Let us write some code which can help us to know these things.

If you are in Linux, then you can actually view the output of the *lscpu* command first. You can actually find the information in a file located at */proc/cpuinfo*.

Now try to write code which will open the file in read only mode and then read the file line by line and find out the number of CPU(s).

Tip

Always remember to read files line by line than reading them as a whole. Sometimes you may have to read files which are way bigger than your available RAM.

After you do this, try to write your own lscpu command in Python :)

Exceptions

In this chapter we will learn about exceptions in Python and how to handle them in your code.

Any error which happens during the execution of the code is an exception. Each exception generally shows some error message.

NameError

When one starts writing code, this will be one of the most common exception he/she will find. This happens when someone tries to access a variable which is not defined.

```
>>> print(kushal)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'kushal' is not defined
```

The last line contains the details of the error message, the rest of the lines shows the details of how it happened (or what caused that exception).

TypeError

TypeError is also one of the most found exception. This happens when someone tries to do an operation with different kinds of incompatible data types. A common example is to do addition of Integers and a string.

```
>>> print(1 + "kushal")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

How to handle exceptions?

We use try...except blocks to handle any exception. The basic syntax looks like

```

try:
    statements to be inside try clause
    statement2
    statement3
    ...
except ExceptionName:
    statements to evaluated in case of ExceptionName happens

```

It works in the following way:

- First all lines between try and except statements.
- If ExceptionName happens during execution of the statements then except clause statements execute
- If no exception happens then the statements inside except clause does not execute.
- If the Exception is not handled in the except block then it goes out of try block.

The following examples showcase these scenarios.

```

>>> def get_number():
...     "Returns a float number"
...     number = float(input("Enter a float number: "))
...     return number
...
>>>
>>> while True:
...     try:
...         print(get_number())
...     except ValueError:
...         print("You entered a wrong value.")
...
Enter a float number: 45.0
45.0
Enter a float number: 24,0
You entered a wrong value.
Enter a float number: Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
  File "<stdin>", line 3, in get_number
KeyboardInterrupt

```

As the first input I provided a proper float value and it printed it back, next I entered a value with a comma, so the except clause executed and the print statement executed.

In the third time I pressed *Ctrl+c* which caused a `KeyboardInterrupt`, which is not handled in the except block so the execution stopped with that exception.

An empty except statement can catch any exception. Read the following example:

```
>>> try:
...     input() # Press Ctrl+c for a KeyboardInterrupt
... except:
...     print("Unknown Exception")
...
Unknown Exception
```

Raising exceptions

One can raise an exception using raise statement.

```
>>> raise ValueError("A value error happened.")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: A value error happened.
```

We can catch these exceptions like any other normal exceptions.

```
>>> try:
...     raise ValueError("A value error happened.")
... except ValueError:
...     print("ValueError in our code.")
...
ValueError in our code.
```

Using finally for cleanup

If we want to have some statements which must be executed under all circumstances, we can use finally clause, it will be always executed before finishing try statements.

```
>>> try:
...     fobj = open("hello.txt", "w")
...     res = 12 / 0
... except ZeroDivisionError:
...     print("We have an error in division")
... finally:
...     fobj.close()
...     print("Closing the file object.")
...
We have an error in division
Closing the file object.
```

In this example we are making sure that the file object we open, must get closed in the finally clause.

Class

Classes and objects are part of programming idea also known as Object-oriented programming [https://en.wikipedia.org/wiki/Object-oriented_programming]. Here the data, and functions working on the data stays together (we call those functions as methods in the objects). Simula [https://en.wikipedia.org/wiki/Object-oriented_programming#History] is the first language which featured these ideas. Java and C++ are two most known object oriented programming languages in the schools.

Your first class

Before writing your first class, you should know the syntax. We define a class in the following way.

```
class nameoftheclass(parent_class):
    statement1
    statement2
    statement3
```

In the statements you can write any Python statement, you can define functions (which we call methods of a class).

```
>>> class MyClass(object):
...     a = 90
...     b = 88
...
>>> p = MyClass()
>>> p
<__main__.MyClass instance at 0xb7c8aa6c>
```

In the above example you can see first we are declaring a class called MyClass, writing some random statements inside that class. After the class definition, we are creating an *object* p of the *class* MyClass. If you do a dir on that

```
>>> dir(p)
['__doc__', '__module__', 'a', 'b']
```


you can see the variables *a* and *b* inside it.

`__init__` method

`__init__` is a special method in Python classes, it is the constructor method for a class. In the following example you can see how to use it.

```
class Student(object):
    """
    Returns a ``Student`` object with the given name, branch and year.

    """
    def __init__(self, name, branch, year):
        self.name = name
        self.branch = branch
        self.year = year
        print("A student object is created.")

    def print_details(self):
        """
        Prints the details of the student.
        """
        print("Name:", self.name)
        print("Branch:", self.branch)
        print("Year:", self.year)
```

`__init__` is called when ever an object of the class is constructed. That means when ever we will create a student object we will see the message “A student object is created” in the prompt. You can see the first argument to the method is *self*. It is a special variable which points to the current object (like this in C++). The object is passed implicitly to every method available in it, but we have to get it explicitly in every method while writing the methods. Example shown below. Remember to declare all the possible attributes in the `__init__` method itself. Even if you are not using them right away, you can always assign them as *None*.

```
>>> std1 = Student()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 4 arguments (1 given)
>>> std1 = Student('Kushal','CSE','2005')
A student object is created
```

In this example at first we tried to create a Student object without passing any argument and Python interpreter complained that it takes exactly 4 arguments but received only one (self). Then we created an object with proper argument values and from the message printed, one can easily understand that `__init__`

method was called as the constructor method.

Now we are going to call *print_details()* method.

```
>>> std1.print_details()
Name: Kushal
Branch: CSE
Year: 2005
```

Note

__init__ is pronounced as dunder init, all functions with double underscore in the front and end is pronounced in this way. Example: dunder str or dunder repr.

Unique class level variables

All the values stored in the instance via *self*. are data inside of an instance. Each instance of the class can have different values for given attribute (anything we access via *.* is also known as attribute). But, when we define an variable in the class level, that is same accross all objects. In the following example, we define a class called *Point*, and we also have a special class level variable called *style* in it. After we create 2 objects of type *Point*, we can see that both has the same class attribute *style* and changing in the class level also changes in the all objects.

```
class Point:
    style="fun"

    def __init__(self, x, y):
        self.x = x
        self.y = y

p1 = Point(10, 10)
p2 = Point(100, 100)
for p in [p1, p2]:
    print(f"Object {p} has style value= {p.style}")

Point.style = "work"
for p in [p1, p2]:
    print(f"Object {p} has style value= {p.style}")
```

Output:

```
Object <__main__.Point object at 0x10de37210> has style value= fun
Object <__main__.Point object at 0x10de0bb50> has style value= fun
Object <__main__.Point object at 0x10de37210> has style value= work
Object <__main__.Point object at 0x10de0bb50> has style value= work
```

`__repr__` method

`__repr__` is a special method used by the print function to show the representation of an object. We can use the same to make our Point object look better as print output.

```
class Point:
    style="fun"

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"<Point x={self.x} y={self.y}>"

p1 = Point(10, 10)
p2 = Point(100, 100)
for p in [p1, p2]:
    print(f"Object {p}")
```

The output:

```
Object <Point x=10 y=10>
Object <Point x=100 y=100>
```

Inheritance

In general we human beings always know about inheritance. In programming it is almost the same. When a class inherits another class it inherits all features (like variables and methods) of the parent class. This helps in reusing codes.

In the next example we first create a class called Person and create two sub-classes Student and Teacher. As both of the classes are inherited from Person class they will have all methods of Person and will have new methods and variables for their own purpose.

student_teacher.py

```
#!/usr/bin/env python3

class Person(object):
    """
    Returns a ``Person`` object with given name.

    """
    def __init__(self, name):
```

```
        self.name = name

    def get_details(self):
        "Returns a string containing name of the person"
        return self.name

class Student(Person):
    """
    Returns a ``Student`` object, takes 3 arguments, name, branch, year.

    """
    def __init__(self, name, branch, year):
        super().__init__(name)
        self.branch = branch
        self.year = year

    def get_details(self):
        "Returns a string containing student's details."
        return "%s studies %s and is in %s year." % (self.name, self.branch, self.year)

class Teacher(Person):
    """
    Returns a ``Teacher`` object, takes a list of strings (list of papers) as
    argument.
    """
    def __init__(self, name, papers):
        super().__init__(name)
        self.papers = papers

    def get_details(self):
        return "%s teaches %s" % (self.name, ','.join(self.papers))

person1 = Person('Sachin')
student1 = Student('Kushal', 'CSE', 2005)
teacher1 = Teacher('Prashad', ['C', 'C++'])

print(person1.get_details())
print(student1.get_details())
print(teacher1.get_details())

The output:

$ ./student_teacher.py
Sachin
```

Kushal studies CSE and is in 2005 year.
Prashad teaches C,C++

In this example you can see how we called the `__init__` method of the parent class using the `super()` in both Student and Teacher classes' `__init__` method. We also reimplemented `get_details()` method of Person class in both Student and Teacher class. So, when we are calling `get_details()` method on the `teacher1` object it returns based on the object itself (which is of teacher class) and when we call `get_details()` on the `student1` or `person1` object it returns based on `get_details()` method implemented in it's own class.

When a class inherits another class, the child class is also known as the instance of the parent class. Here is an example based on the above class.

```
`Python isinstance(student1, Person) True `
```

Multiple Inheritance

One class can inherit more than one classes. It gets access to all methods and variables of the parent classes. The general syntax is:

```
class MyClass(Parentclass1, Parentclass2,...):
    def __init__(self):
        Parentclass1.__init__(self)
        Parentclass2.__init__(self)
        ...
        ...
```

Encapsulation in Python

Encapsulation is a way to provide details on how a data can be accessed. In Python we have encapsulation as a programming style, which is different than many other programming languages. For example, we use a leading `__` before any variable name to tell that it is private. This way if the developer wants, they can have a different variable with similar name in the child class.

```
class Person():
    """
    Returns a ``Person`` object with given name.

    """
    def __init__(self, name):
        self._name = name

    def get_details(self):
        "Returns a string containing name of the person"
        return self._name
```

```
class Child(Person):
    def __init__(self, name):
        super().__init__(name)

    def tell(self):
        print(f"The name is {self._name}")

c = Child("kushal")
c.tell()
```

The output:

```
The name is kushal
```

You can see that we can still access the `_name` attribute. But, we are letting the developer know that `_name` is a private attribute. If you want to make sure that the attribute can not be accessed directly in the child class, you can use `__` in front of the attribute name. It uses something called name mangling <<https://docs.python.org/3/tutorial/classes.html#private-variables>>.

Deleting an object

As we already know how to create an object, now we are going to see how to delete an Python object. We use `del` for this.

```
>>> s = "I love you"
>>> del s
>>> s
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 's' is not defined
```

`del` actually decreases reference count by one. When the reference count of an object becomes zero the garbage collector will delete that object.

Getters and setters in Python

One simple answer, don't. If you are coming from other languages (read Java), you will be tempted to use getters or setters in all your classes. Please don't. Just use the attributes directly. The following shows a direct example.

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
... 
```

```
>>> std = Student("Kushal Das")
>>> print(std.name)
Kushal Das
>>> std.name = "Python"
>>> print(std.name)
Python
```

Properties

If you want more fine tuned control over data attribute access, then you can use properties. In the following example of a bank account, we will make sure that no one can set the money value to negative and also a property called *inr* will give us the INR values of the dollars in the account.

```
#!/usr/bin/env python3

class Account(object):
    """The Account class,
    The amount is in dollars.
    """
    def __init__(self, rate):
        self.__amt = 0
        self.rate = rate

    @property
    def amount(self):
        "The amount of money in the account"
        return self.__amt

    @property
    def inr(self):
        "Gives the money in INR value."
        return self.__amt * self.rate

    @amount.setter
    def amount(self, value):
        if value < 0:
            print("Sorry, no negative amount in the account.")
            return
        self.__amt = value

if __name__ == '__main__':
    acc = Account(rate=61) # Based on today's value of INR :(
    acc.amount = 20
    print("Dollar amount:", acc.amount)
```

```
print("In INR:", acc.inr)
acc.amount = -100
print("Dollar amount:", acc.amount)
```

Output:

```
$ python property.py
Dollar amount: 20
In INR: 1220
Sorry, no negative amount in the account.
Dollar amount: 20
```

Special dunder methods in classes

Below, we will see some special dunder methods (the methods which have double underscores `__` before and after the name, example: `__init__`, we call it *dunder init*).

`__len__` method

Dunder len is a method used by the `len` function to know the length of any iterator or similar objects. It should return an Integer. The `len` function verifies if the returned value is Integer or not.

```
class Foo:
    "Example class for __len__"
    def __init__(self, length=5):
        self.length = 5

    def __len__(self):
        return self.length

f = Foo()
length = len(f)
print(f"Length of the f object is {length}")
```

The output:

```
$ python3 code/lenexample.py
Length of the f object is 5
```

`__contains__` method

This method helps us to use in with our objects. For example, if we want to match “kushal” in `studnet1` to be True, we implement `__contains__` method in our class.


```

class Student(Person):
    """
    Returns a ``Student`` object, takes 3 arguments, name, branch, year.

    """
    def __init__(self, name, branch, year):
        super().__init__(name)
        self.branch = branch
        self.year = year

    def get_details(self):
        "Returns a string containing student's details."
        return "%s studies %s and is in %s year." % (self.name, self.branch, self.year)

    def __contains__(self, name):
        return self._name == name

student1 = Student("kushal", "cse", 2005)

print("kushal" in student1)
print("sachin" in student1)

```

`__new__` method

`__new__` is a special method. When we create an instance of the class, internally this method gets called first, and then `__init__` gets called on the returned object. It takes the class as the first argument. In the following example, we are using our Point class again.

```

p = Point.__new__(Point, 2, 3)
p.__init__(2, 3)
print(p)

```

```
<Point x=2 y=3>
```

Creating a new context manager

Do you remember the `with` statement from the files chapter? Where we used a context manager to make sure that the file is closed after we are done? The same style is used in many places where we want the resources to be cleaned up after the work is done; sometimes we want to call some extra functions when we are done. We can write our own context manager in our classes using `__enter__` and `__exit__` methods.

For example, we will create a new class called `TimeLog` which in turn will create

a file called tmpdata.txt in the current directory to log the time this context manager is created and when it is done.

```
import time

class TimeLog:

    def __init__(self):
        self.fobj = None

    def __enter__(self):
        self.fobj = open("tmpdata.txt", "w")
        self.fobj.write(f"Entering at {time.time()}\n")

    def __exit__(self, ty, value, tb):
        self.fobj.write(f"Done at {time.time()}\n")
        self.fobj.close()
        self.fobj = None

with TimeLog() as tl:
    a = [1, 2, 3]
    print(a)
```

Output in the tmpdata.txt file.

```
Entering at 1590551277.323565
Done at 1590551277.3238761
```

Later in the book we will learn even simpler methods to create context managers.

Deep down inside

If we look inside of our class definitions, we will find a dictionary at the center. Let us look at it in details in the following example.

```
class User:
    def __init__(self, name, uid, gid, home, sudo):
        self.name = name
        self.uid = uid
        self.gids = [gid,]
        self.home = home
        self.sudo = sudo

    def can_sudo(self):
        return self.sudo
```

```
u = User("kdas", 1000, 1000, "/home/kdas", True)
pprint(u.__dict__)
```

```
{'gids': [1000],
 'home': '/home/kdas',
 'name': 'kdas',
 'sudo': True,
 'uid': 1000}
```

All the attributes we defined via `self` in the `__init__` method, are stored in the `__dict__` dictionary inside of each instance. When we try access any of these attributes, Python first looks at this dictionary of the object, and then also in the `__dict__` of the class itself.

```
>>> pprint(User.__dict__)
mappingproxy({'__dict__': <attribute '__dict__' of 'User' objects>,
              '__doc__': None,
              '__init__': <function User.__init__ at 0x7fa8c6f1bd40>,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'User' objects>,
              'can_sudo': <function User.can_sudo at 0x7fa8c6f3e3b0>})
```

When we try to access any attribute via the `.` operator, Python first checks the `__getattr__` method to look at the `__dict__`. If the key can not be found, it tries to call the `__getattr__` method on the object.

```
class Magic:
    def __init__(self):
        self.name = "magic"

    def __getattr__(self, attr):
        return attr.upper()
```

Now, if we try to use this Magic class, we can access any random attribute even if they don't exist.

```
python3 -i deepinsideobjects.py
>>> m = Magic()
>>> m.name
'magic'
>>> m.what_is_this_magic
'WHAT_IS_THIS_MAGIC'
>>> m.this
'THIS'
>>> m.hello
'HELLO'
```

Using the same approach we took, to access the data stored inside another object of our class, we can also implement the `__setattr__` method, which is used to

set a value to any attribute.

```
class User:
```

```
    def __init__(self, name, uid, gid, home, sudo):
        self.__dict__["_internal"] = {"name": name, "uid": uid, "gids": [gid,], "home": home, "su

    def can_sudo(self):
        return self._internal["sudo"]

    def __getattr__(self, attr):
        print(f"Accessing attr: {attr}")
        return self._internal[attr]

    def __setattr__(self, attr, value):
        print(f"Setting attribute {attr} to {value}")
        self._internal[attr] = value
```

```
u = User("kdas", 1000, 1000, "/home/kdas", True)
```

When we try to access any attribute of the object `u`, we can see the following.

```
python3 -i deepinsideobjects.py
>>> u.name
Accessing attr: name
'kdas'
>>> u.uid
Accessing attr: uid
1000
>>> u.can_sudo()
True
```

There is also `__delattr__` method to delete any attribute of an instance. Feel free to add it to the class above and see how it behaves.

Modules

In this chapter we are going to learn about Python modules.

Introduction

Up until now, all the code we wrote in the Python interpreter was lost when we exited the interpreter. But when people write large programs they tend to break their code into multiple different files for ease of use, debugging and readability. In Python we use *modules* to achieve such goals. Modules are nothing but files with Python definitions and statements. The name of the file should be valid Python name (think about any variable name) and in lowercase.

You can find the name of the module by accessing the `__name__` variable. It is a global variable.

Now we are going to see how modules work. Create a file called `bars.py`. Content of the file is given below.

```
"""
Bars Module
=====
This is an example module with provide different ways to print bars.
"""

def starbar(num):
    """
    Prints a bar with *

    :arg num: Length of the bar

    """
    print("*" * num)
```

```
def hashbar(num):
    """
    Prints a bar with #

    :arg num: Length of the bar

    """
    print("#" * num)

def simplebar(num):
    """
    Prints a bar with -

    :arg num: Length of the bar

    """
    print("-" * num)
```

Now we are going to start the Python interpreter and import our module.

```
>>> import bars
>>>
```

This will import the module bars. We have to use the module name to access functions inside the module.

```
>>> bars.hashbar(10)
#####
>>> bars.simplebar(10)
-----
>>> bars.starbar(10)
*****
```

Importing modules

There are different ways to import modules. We already saw one way to do this. You can even import selected functions from modules. To do so:

```
>>> from bars import simplebar, starbar
>>> simplebar(20)
-----
```

Warning

It is suggested to avoid *from module import ** for importing from the modules.

Remember that when we import a module, the Python interpreter executes the whole file, and then imports it as a module. This is true even when we

import a single function from the module. Look at the following updated bars.py example.

```
"""
Bars Module
=====

This is an example module which provides different ways to print bars.

"""

ANSWER = 42


def starbar(num):
    """
    Prints a bar with *

    :arg num: Length of the bar

    """
    print("*" * num)


def hashbar(num):
    """
    Prints a bar with #

    :arg num: Length of the bar

    """
    print("#" * num)


def simplebar(num):
    """
    Prints a bar with -

    :arg num: Length of the bar

    """
    print("-" * num)


print(ANSWER)
```

Now if we try to import only the simplebar function from the bars module, it

will still print the value ANSWER.

```
>>> from bars import simplebar
42
```

After the modules get imported for the first time, they are stored (as cache) inside of `sys.modules` dictionary using the module name as key. If you try to import it again, Python checks if it is there in the `sys.modules` and returns the module from there. All imported modules are of type `module`.

```
>>> sys.modules["bars"]
<module 'bars' from '/home/kdas/code/pym/code/bars.py'>
>>> dir(bars)
['ANSWER', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__']
>>> type(bars)
<class 'module'>
>>> bars.__name__
'bars'
>>> bars.__file__
'/home/kdas/code/pym/code/bars.py'
>>> bars.__cached__
'/home/kdas/code/pym/code/__pycache__/bars.cpython-37.pyc'
```

Packages and submodules

We can have many submodules inside a module. A directory with a `__init__.py` can also be used as a module and all `.py` files inside it become submodules.

```
$ tree mymodule
mymodule
|-- bars.py
|-- __init__.py
`-- utils.py
```

In this example *mymodule* is the module name and *bars* and *utils* are two submodules in it. You can create an empty `__init__.py` using `touch` command.

```
$ touch mymodule/__init__.py
```

`__all__` in `__init__.py`

If `__init__.py` file contains a list called `__all__`, then only the names listed there will be public. If the *mymodule*'s `__init__.py` file contains the following

```
from .bars import simplebar
__all__ = [simplebar, ]
```


Then from `mymodule` only `simplebar` will be available.

Note

`from mymodule import *` will only work for module level objects, trying to use it to import functions or classes will cause syntax error.

You have also noticed that we used a new style while importing the `simplebar` function from inside of the `mymodule` package. Using this `.` notation is known as explicit import.

- `from . import foo` # imports `./foo.py`
- `from .bars import foo` # imports `foo` from `./bars.py`
- `from .. import foo` # imports `../foo.py`
- `from ..life import foo` # imports `../life/foo.py`

Default modules

Now your Python installation comes with different modules installed, you can use them as required and install new modules for any other special purposes. In the following few examples we are going to see many examples on the same.

```
>>> help()
```

```
Welcome to Python 3.5's help utility!
```

```
If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.5/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 contain a given string such as "spam", type "modules spam".
```

```
help> modules
```

The above example shows how to get the list of all installed modules in your system. I am not pasting them here as it is a big list in my system :)

You can also use `help()` function in the interpreter to find documentation about any module/classes. Say you want to know all available methods in strings, you can use the following method

```
>>> help(str)
```

Module os

`os` [<https://docs.python.org/3/library/os.html#module-os>] module provides operating system dependent functionality. You can import it using the following import statement.

```
>>> import os
```

getuid() function returns the current process's effective user's id.

```
>>> os.getuid()
500
```

getpid() returns the current process's id. *getppid()* returns the parent process's id.

```
>>> os.getpid()
16150
>>> os.getppid()
14847
```

uname() returns different information identifying the operating system, in Linux it returns details you can get from the *uname* command. The returned object is a tuple, (*sysname*, *nodename*, *release*, *version*, *machine*)

```
>>> os.uname()
('Linux', 'd80', '2.6.34.7-56.fc13.i686.PAE', '#1 SMP Wed Sep 15 03:27:15 UTC 2010', '')
```

getcwd() *returns the current working directory. *chdir(path)* changes the current working directory to path. In the example we first see the current directory which is my home directory and change the current directory to */tmp* and then again checking the current directory.

```
>>> os.getcwd()
'/home/kushal'
>>> os.chdir('/tmp')
>>> os.getcwd()
'/tmp'
```

So let us use another function provided by the `os` module and create our own function to list all files and directories in any given directory.

```
def view_dir(path='.'):
    """
    This function prints all files and directories in the given directory.
    :args path: Path to the directory, default is current directory
    """
    names = os.listdir(path)
```

```
names.sort()
for name in names:
    print(name, end = ' ')
```

Using the *view_dir* example.

```
>>> view_dir('/')
.readahead bin boot dev etc home junk lib lib64 lost+found media mnt opt
proc root run sbin srv sys tmp usr var
```

There are many other very useful functions available in the OS module, you can read about them here [<https://docs.python.org/3/library/os.html>]

Requests Module

requests is a Python module which changed the way people used to write code for many many projects. It helps you to do HTTP GET or POST calls in a very simple but elegant way. This is a third party module, that means you have to install it from your OS distribution packages, it does not come default.

```
# dnf install python3-requests
```

The above command will install Python3 version of the requests module in your system.

Getting a simple web pages

You can use the *get* method to fetch any website.

```
>>> import requests
>>> req = requests.get('http://google.com')
>>> req.status_code
200
```

The *text* attribute holds the HTML returned by the server.

Using this knowledge, let us write a command which can download a given file (URL) from Internet.

```
#!/usr/bin/env python3
import os
import os.path
import requests
```

```
def download(url):
    '''Download the given url and saves it to the current directory.

    :arg url: URL of the file to be downloaded.
    '''
    req = requests.get(url)
```

```

# First let us check non existing files.
if req.status_code == 404:
    print('No such file found at %s' % url)
    return
filename = url.split('/')[-1]
with open(filename, 'wb') as fobj:
    fobj.write(req.content)
print("Download over.")

if __name__ == '__main__':
    url = input('Enter a URL:')
    download(url)

```

Here we used something new, when the module name is `__main__`, then only ask for a user input and then download the given URL. This also prevents the same when some other Python code imports this file as a Python module.

To learn more about requests module, go to their wonderful documentation [<http://docs.python-requests.org>].

You can actually modify the above program to become more user friendly. For example, you can check if that given filename already exists in the current directory or not. Use `os.path` [<https://docs.python.org/3/library/os.path.html#module-os.path>] module for the name.

Command line arguments

Do you remember your `ls` command, you can pass different kind of options as command line arguments. You can do that too .. important:: your application. Read this how-to [<https://docs.python.org/3/howto/argparse.html>] guide to learn about it.

TAB completion in your Python interpreter

First create a file as `~/.pythonrc` and include the following in that file

```

import rlcompleter, readline
readline.parse_and_bind('tab: complete')

history_file = os.path.expanduser('~/.python_history')
readline.read_history_file(history_file)

import atexit
atexit.register(readline.write_history_file, history_file)

```

Next, just export PYTHONSTARTUP variable pointing to this file from your `~/.bashrc` file.

```
export PYTHONSTARTUP=~/.pythonrc
```

Now from future whenever you open a bash shell, you will have TAB completion and history of code entered in your Python interpreter.

To use it in the current shell, source the bashrc file.

```
$ source ~/.bashrc
```

Collections module

In this chapter we will learn about a module called *Collections*. This module implements some nice data structures which will help you to solve various real life problems.

```
>>> import collections
```

This is how you can import the module, now we will see the available classes which you can use.

Counter

Counter is a *dict* subclass which helps to count hashable objects. Inside it elements are stored as dictionary keys and counts are stored as values which can be zero or negative.

Below we will see one example where we will find occurrences of words in the Python LICENSE file.

Counter example

```
>>> from collections import Counter
>>> import re
>>> path = '/usr/share/doc/python-2.7.3/LICENSE'
>>> words = re.findall('\w+', open(path).read().lower())
>>> Counter(words).most_common(10)
[('2', 97), ('the', 80), ('or', 78), ('1', 76), ('of', 61), ('to', 50), ('and', 47), ('python', 4
```

Counter objects has a method called *elements* which returns an iterator over elements repeating each as many times as its count. Elements are returned in arbitrary order.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common is a method which returns most common elements and their counts from the most common to the least.

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
```

defaultdict

defaultdict is a dictionary like object which provides all methods provided by dictionary but takes first argument (*default_factory*) as default data type for the dictionary. Using *defaultdict* is faster than doing the same using *dict.setdefault* method.

defaultdict example

```
:: >>> from collections import defaultdict
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> d.items()
dict_items([('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])])
```

In the example you can see even if the key is not there in the *defaultdict* object, it automatically creates an empty list. *list.append* then helps to append the value to the list.

namedtuple

Named tuples helps to have meaning of each position in a tuple and allow us to code with better readability and self-documenting code. You can use them in any place where you are using *tuples*. In the example we will create a *namedtuple* to show hold information for points.

Named tuple

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y']) # Defining the namedtuple
>>> p = Point(10, y=20) # Creating an object
>>> p
Point(x=10, y=20)
>>> p.x + p.y
30
>>> p[0] + p[1] # Accessing the values in normal way
30
```

```
>>> x, y = p      # Unpacking the tuple
>>> x
10
>>> y
20
```


Using VS Code as your primary Python editor

If you have reached this chapter in the book, then I think you can take a step up in the programming world by using a new editor which has a lot more features than our starting editor (mu).

VS Code [<https://code.visualstudio.com/>] is an Open Source multi-platform code editor from Microsoft which I personally like a lot. In this chapter, I am going to show you how you can start using it.

Installing VS Code

Download and install VS Code in your Linux distribution following the official guidelines [<https://code.visualstudio.com/docs/setup/linux>]. Please make sure that you are following all the steps given in that page. You should add the project repository properly (either deb or rpm repo) and always verify the package (the gpg signed packages in the repo makes it easier).

Remember that VS Code ships monthly, means every month you will get update of this editor with many new features and bug fixes as required.

Using VS Code

The above is the starting screen of VS Code, you should notice the small information box in the right hand corner.

You can disable `telemetry` and thus sending the commands and other information that Microsoft collects. Go to the settings page by using `File->Preference->Settings` in the menu.

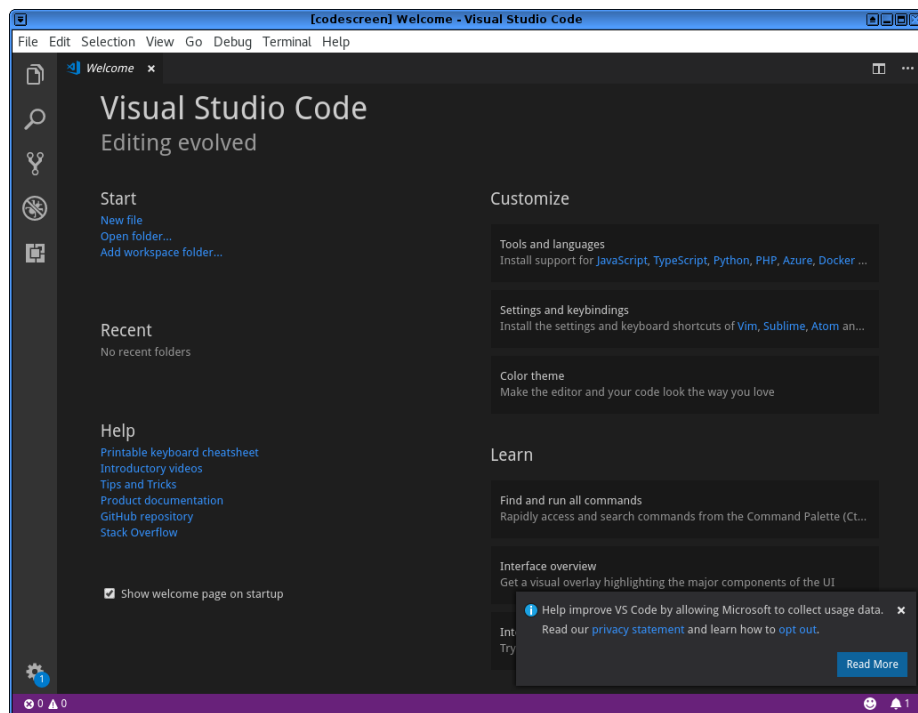


Figure 10:

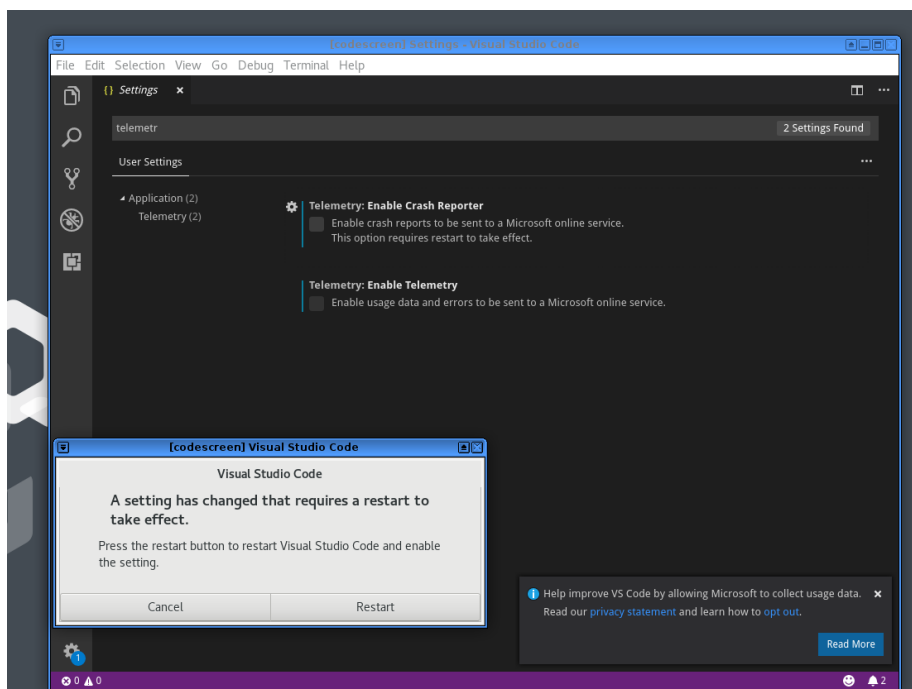


Figure 11:

Install the Python extension

The next step would be to install the Python extension [<https://marketplace.visualstudio.com/items?itemName=python.python>] in VS Code. Go to “Extensions” from the left hand activity bar, and install the Python extension.

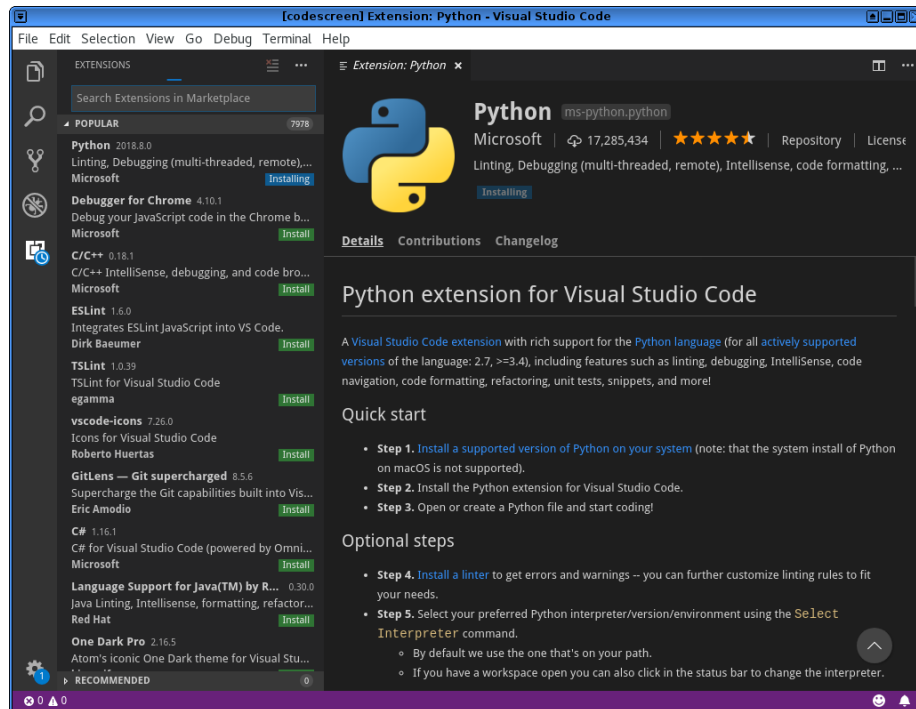


Figure 12:

Start working on your code

Now, you can start working on your favorite project. Open up the directory in VS Code, and start editing.

Feel free to go through the VS Code documentation [<https://code.visualstudio.com/docs/getstarted/tips-and-tricks>] to learn more about the editor.

Important

Learn about various shortcuts from this gist [<https://gist.github.com/bradtraversy/b28a0a361880141af92>]

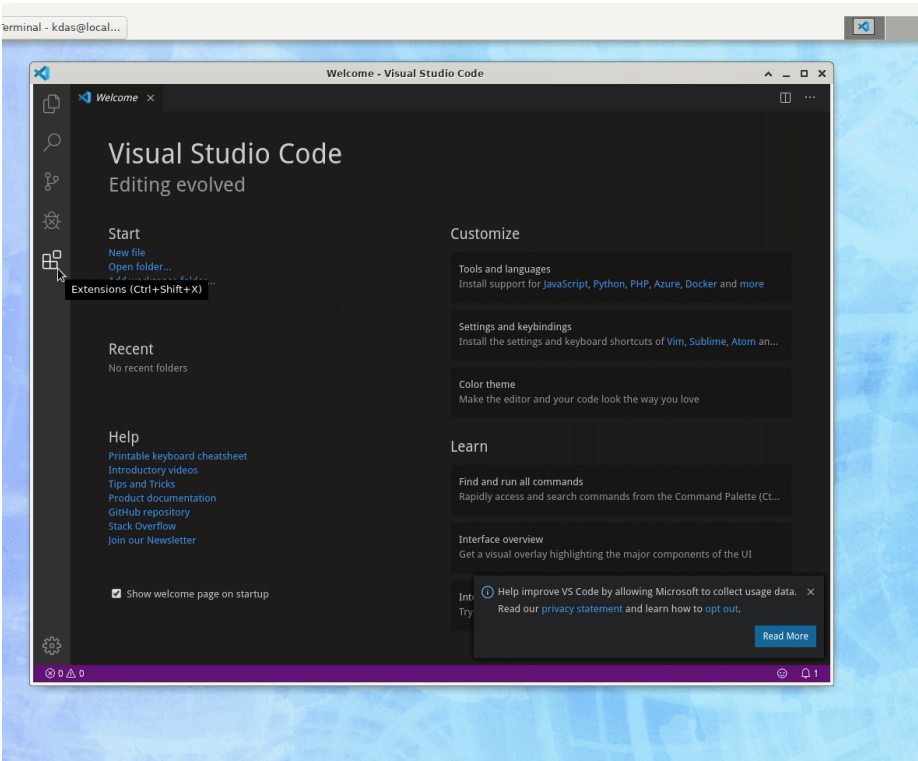


Figure 13:

Install the Device Simulator Express extension

Follow this page [<https://marketplace.visualstudio.com/items?itemName=ms-python.devicesimulatorexpress&ssr=false#overview>] to install the simulator, if you are on Linux, remember to remove the Windows dependency from the requirements.txt file as shown in the GIF file below.

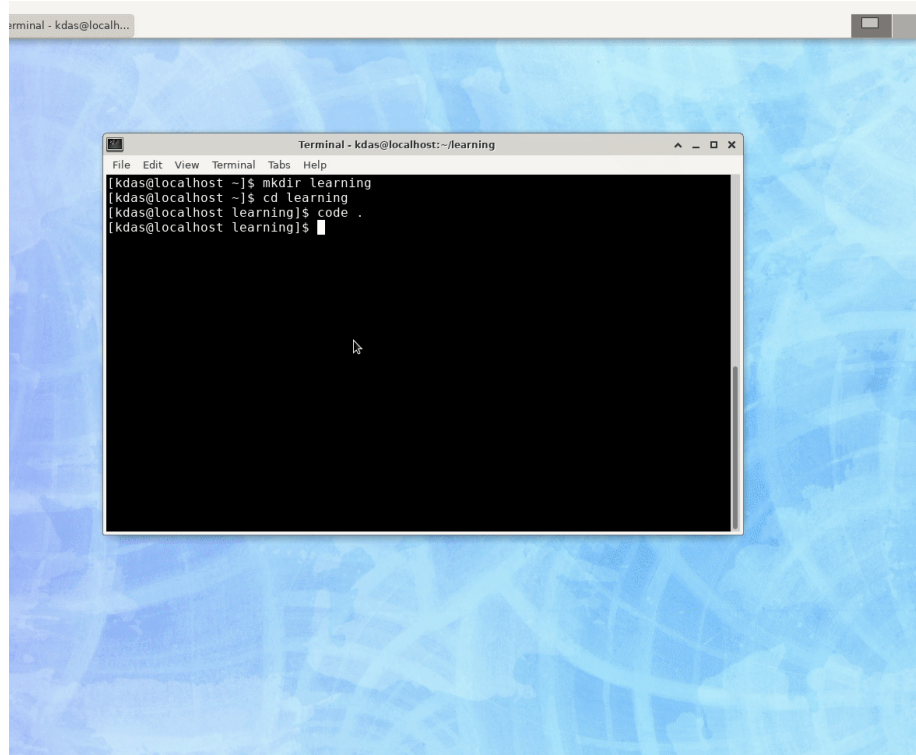


Figure 14:

PEP8 Guidelines

Introduction

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python [1].

This document and PEP 257 (Docstring Conventions) were adapted from Guido's original Python Style Guide essay, with some additions from Barry's style guide [2].

A Foolish Consistency is the Hobgoblin of Little Minds

One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 says, "Readability counts".

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.

But most importantly: know when to be inconsistent – sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

Two good reasons to break a particular rule:

1. When applying the rule would make the code less readable, even for someone who is used to reading code that follows the rules.
2. To be consistent with surrounding code that also breaks it (maybe for historic reasons) – although this is also an opportunity to clean up someone else's mess (in true XP style).

Code lay-out

Indentation

Use 4 spaces per indentation level.

For really old code that you don't want to mess up, you can continue to use 8-space tabs.

Continuation lines should align wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a hanging indent. When using a hanging indent the following considerations should be applied; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line.

Yes:

```
# Aligned with opening delimiter
foo = long_function_name(var_one, var_two,
                        var_three, var_four)

# More indentation included to distinguish this from the rest.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

No:

```
# Arguments on first line forbidden when not using vertical alignment
foo = long_function_name(var_one, var_two,
                        var_three, var_four)

# Further indentation required as indentation is not distinguishable
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Optional:

```
# Extra indentation is not necessary.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

The closing brace/bracket/parenthesis on multi-line constructs may either line up under the first non-whitespace character of the last line of list, as in:

```
my_list = [
    1, 2, 3,
```



```
    4, 5, 6,  
    ]  
result = some_function_that_takes_arguments(  
    'a', 'b', 'c',  
    'd', 'e', 'f',  
    )
```

or it may be lined up under the first character of the line that starts the multi-line construct, as in:

```
my_list = [  
    1, 2, 3,  
    4, 5, 6,  
]  
result = some_function_that_takes_arguments(  
    'a', 'b', 'c',  
    'd', 'e', 'f',  
)
```

Tabs or Spaces?

Never mix tabs and spaces.

The most popular way of indenting Python is with spaces only. The second-most popular way is with tabs only. Code indented with a mixture of tabs and spaces should be converted to using spaces exclusively. When invoking the Python command line interpreter with the `-t` option, it issues warnings about code that illegally mixes tabs and spaces. When using `-tt` these warnings become errors. These options are highly recommended!

For new projects, spaces-only are strongly recommended over tabs. Most editors have features that make this easy to do.

Maximum Line Length

Limit all lines to a maximum of 79 characters.

There are still many devices around that are limited to 80 character lines; plus, limiting windows to 80 characters makes it possible to have several windows side-by-side. The default wrapping on such devices disrupts the visual structure of the code, making it more difficult to understand. Therefore, please limit all lines to a maximum of 79 characters. For flowing long blocks of text (docstrings or comments), limiting the length to 72 characters is recommended.

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation.

Backslashes may still be appropriate at times. For example, long, multiple `with`-statements cannot use implicit continuation, so backslashes are acceptable:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
     open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

Another such case is with `assert` statements.

Make sure to indent the continued line appropriately. The preferred place to break around a binary operator is *after* the operator, not before it. Some examples:

```
class Rectangle(Blob):

    def __init__(self, width, height,
                  color='black', emphasis=None, highlight=0):
        if (width == 0 and height == 0 and
            color == 'red' and emphasis == 'strong' or
            highlight > 100):
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                             (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

Blank Lines

Separate top-level function and class definitions with two blank lines.

Method definitions inside a class are separated by a single blank line.

Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).

Use blank lines in functions, sparingly, to indicate logical sections.

Python accepts the control-L (i.e. `^L`) form feed character as whitespace; Many tools treat these characters as page separators, so you may use them to separate pages of related sections of your file. Note, some editors and web-based code viewers may not recognize control-L as a form feed and will show another glyph in its place.

Encodings (PEP 263)

Code in the core Python distribution should always use the ASCII or Latin-1 encoding (a.k.a. ISO-8859-1). For Python 3.0 and beyond, UTF-8 is preferred

over Latin-1, see PEP 3120.

Files using ASCII should not have a coding cookie. Latin-1 (or UTF-8) should only be used when a comment or docstring needs to mention an author name that requires Latin-1; otherwise, using `\x`, `\u` or `\U` escapes is the preferred way to include non-ASCII data in string literals.

For Python 3.0 and beyond, the following policy is prescribed for the standard library (see PEP 3131): All identifiers in the Python standard library **MUST** use ASCII-only identifiers, and **SHOULD** use English words wherever feasible (in many cases, abbreviations and technical terms are used which aren't English). In addition, string literals and comments must also be in ASCII. The only exceptions are (a) test cases testing the non-ASCII features, and (b) names of authors. Authors whose names are not based on the latin alphabet **MUST** provide a latin transliteration of their names.

Open source projects with a global audience are encouraged to adopt a similar policy.

Imports

- Imports should usually be on separate lines, e.g.:

```
Yes: import os
     import sys
```

```
No:  import sys, os
```

It's okay to say this though:

```
from subprocess import Popen, PIPE
```

- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

Imports should be grouped in the following order:

1. standard library imports
2. related third party imports
3. local application/library specific imports

You should put a blank line between each group of imports.

Put any relevant `__all__` specification after the imports.

- Relative imports for intra-package imports are highly discouraged. Always use the absolute package path for all imports. Even now that PEP 328 is fully implemented in Python 2.5, its style of explicit relative imports is actively discouraged; absolute imports are more portable and usually more readable.

- When importing a class from a class-containing module, it's usually okay to spell this:

```
from myclass import MyClass
from foo.bar.yourclass import YourClass
```

If this spelling causes local name clashes, then spell them

```
import myclass
import foo.bar.yourclass
```

and use “myclass.MyClass” and “foo.bar.yourclass.YourClass”.

Whitespace in Expressions and Statements

Pet Peeves

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces.

```
Yes: spam(ham[1], {eggs: 2})
No:  spam( ham[ 1 ], { eggs: 2 } )
```

- Immediately before a comma, semicolon, or colon:

```
Yes: if x == 4: print x, y; x, y = y, x
No:  if x == 4 : print x , y ; x , y = y , x
```

- Immediately before the open parenthesis that starts the argument list of a function call:

```
Yes: spam(1)
No:  spam (1)
```

- Immediately before the open parenthesis that starts an indexing or slicing:

```
Yes: dict['key'] = list[index]
No:  dict ['key'] = list [index]
```

- More than one space around an assignment (or other) operator to align it with another.

Yes:

```
x = 1
y = 2
long_variable = 3
```

No:

```
x           = 1
y           = 2
long_variable = 3
```

Other Recommendations

- Always surround these binary operators with a single space on either side: assignment (`=`), augmented assignment (`+=`, `-=` etc.), comparisons (`==`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, `in`, `not in`, `is`, `is not`), Booleans (`and`, `or`, `not`).
- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgement; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

Yes:

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

No:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

- Don't use spaces around the `=` sign when used to indicate a keyword argument or a default parameter value.

Yes:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

No:

```
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

- Compound statements (multiple statements on the same line) are generally discouraged.

Yes:

```
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

Rather not:

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

- While sometimes it's okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

Rather not:

```
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

Definitely not:

```
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                               list, like, this)

if foo == 'blah': one(); two(); three()
```

Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

If a comment is short, the period at the end can be omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period.

You should use two spaces after a sentence-ending period.

When writing English, Strunk and White apply.

Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

Block Comments

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a `#` and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single `#`.

Inline Comments

Use inline comments sparingly.

An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a `#` and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1                # Increment x
```

But sometimes, this is useful:

```
x = x + 1                # Compensate for border
```

Documentation Strings

Conventions for writing good documentation strings (a.k.a. “docstrings”) are immortalized in PEP 257.

- Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the `def` line.
- PEP 257 describes good docstring conventions. Note that most importantly, the `"""` that ends a multiline docstring should be on a line by itself, and preferably preceded by a blank line, e.g.:

```
"""Return a foobang
```

```
Optional plotz says to frobnicate the bizbaz first.
```

```
"""
```

- For one liner docstrings, it's okay to keep the closing `"""` on the same line.

Version Bookkeeping

If you have to have Subversion, CVS, or RCS crud in your source file, do it as follows.

```
__version__ = "$Revision$"  
# $Source$
```

These lines should be included after the module's docstring, before any other code, separated by a blank line above and below.

Naming Conventions

The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent – nevertheless, here are the currently recommended naming standards. New modules and packages (including third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

Descriptive: Naming Styles

There are a lot of different naming styles. It helps to be able to recognize what naming style is being used, independently from what they are used for.

The following naming styles are commonly distinguished:

- `b` (single lowercase letter)
- `B` (single uppercase letter)
- `lowercase`
- `lower_case_with_underscores`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (or `CapWords`, or `CamelCase` – so named because of the bumpy look of its letters [3]). This is also sometimes known as `StudlyCaps`.

Note: When using abbreviations in `CapWords`, capitalize all the letters of the abbreviation. Thus `HTTPServerError` is better than `HttpServerError`.

- `mixedCase` (differs from `CapitalizedWords` by initial lowercase character!)
- `Capitalized_Words_With_Underscores` (ugly!)

There's also the style of using a short unique prefix to group related names together. This is not used much in Python, but it is mentioned for completeness. For example, the `os.stat()` function returns a tuple whose items traditionally

have names like `st_mode`, `st_size`, `st_mtime` and so on. (This is done to emphasize the correspondence with the fields of the POSIX system call struct, which helps programmers familiar with that.)

The X11 library uses a leading X for all its public functions. In Python, this style is generally deemed unnecessary because attribute and method names are prefixed with an object, and function names are prefixed with a module name.

In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):

- `_single_leading_underscore`: weak “internal use” indicator. E.g. `from M import *` does not import objects whose name starts with an underscore.
- `_single_trailing_underscore_`: used by convention to avoid conflicts with Python keyword, e.g.

```
Tkinter.Toplevel(master, class_='ClassName')
```
- `__double_leading_underscore`: when naming a class attribute, invokes name mangling (inside class `FooBar`, `__boo` becomes `_FooBar__boo`; see below).
- `__double_leading_and_trailing_underscore__`: “magic” objects or attributes that live in user-controlled namespaces. E.g. `__init__`, `__import__` or `__file__`. Never invent such names; only use them as documented.

Prescriptive: Naming Conventions

Names to Avoid

Never use the characters ‘l’ (lowercase letter el), ‘O’ (uppercase letter oh), or ‘I’ (uppercase letter eye) as single character variable names.

In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use ‘l’, use ‘L’ instead.

Package and Module Names

Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

Since module names are mapped to file names, and some file systems are case insensitive and truncate long names, it is important that module names be chosen to be fairly short – this won’t be a problem on Unix, but it may be a problem when the code is transported to older Mac or Windows versions, or DOS.

When an extension module written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. `_socket`).

Class Names

Almost without exception, class names use the CapWords convention. Classes for internal use have a leading underscore in addition.

Exception Names

Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix “Error” on your exception names (if the exception actually is an error).

Global Variable Names

(Let’s hope that these variables are meant for use inside one module only.) The conventions are about the same as those for functions.

Modules that are designed for use via `from M import *` should use the `__all__` mechanism to prevent exporting globals, or use the older convention of prefixing such globals with an underscore (which you might want to do to indicate these globals are “module non-public”).

Function Names

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

mixedCase is allowed only in contexts where that’s already the prevailing style (e.g. `threading.py`), to retain backwards compatibility.

Function and method arguments

Always use `self` for the first argument to instance methods.

Always use `cls` for the first argument to class methods.

If a function argument’s name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus `class_` is better than `clss`. (Perhaps better is to avoid such clashes by using a synonym.)

Method Names and Instance Variables

Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.

Use one leading underscore only for non-public methods and instance variables.

To avoid name clashes with subclasses, use two leading underscores to invoke Python’s name mangling rules.

Python mangles these names with the class name: if class `Foo` has an attribute named `__a`, it cannot be accessed by `Foo.__a`. (An insistent user could still gain access by calling `Foo._Foo__a`.) Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

Note: there is some controversy about the use of `__`names (see below).

Constants

Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include `MAX_OVERFLOW` and `TOTAL`.

Designing for inheritance

Always decide whether a class’s methods and instance variables (collectively: “attributes”) should be public or non-public. If in doubt, choose non-public; it’s easier to make it public later than to make a public attribute non-public.

Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backward incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won’t change or even be removed.

We don’t use the term “private” here, since no attribute is really private in Python (without a generally unnecessary amount of work).

Another category of attributes are those that are part of the “subclass API” (often called “protected” in other languages). Some classes are designed to be inherited from, either to extend or modify aspects of the class’s behavior. When designing such a class, take care to make explicit decisions about which attributes are public, which are part of the subclass API, and which are truly only to be used by your base class.

With this in mind, here are the Pythonic guidelines:

- Public attributes should have no leading underscores.
- If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling. (However, not withstanding this rule, ‘cls’ is the preferred spelling for any variable or argument which is known to be a class, especially the first argument to a class method.)

Note 1: See the argument name recommendation above for class methods.

- For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods. Keep in mind

that Python provides an easy path to future enhancement, should you find that a simple data attribute needs to grow functional behavior. In that case, use properties to hide functional implementation behind simple data attribute access syntax.

Note 1: Properties only work on new-style classes.

Note 2: Try to keep the functional behavior side-effect free, although side-effects such as caching are generally fine.

Note 3: Avoid using properties for computationally expensive operations; the attribute notation makes the caller believe that access is (relatively) cheap.

- If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores. This invokes Python's name mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name.

Note 1: Note that only the simple class name is used in the mangled name, so if a subclass chooses both the same class name and attribute name, you can still get name collisions.

Note 2: Name mangling can make certain uses, such as debugging and `__getattr__()`, less convenient. However the name mangling algorithm is well documented and easy to perform manually.

Note 3: Not everyone likes name mangling. Try to balance the need to avoid accidental name clashes with potential use by advanced callers.

References

[1]

PEP 7, Style Guide for C Code, van Rossum

[2]

Barry's GNU Mailman style guide <http://barry.warsaw.us/software/STYLEGUIDE.txt>

[3]

<http://www.wikipedia.com/wiki/CamelCase>

Copyright

Author: Guido van Rossum <guido@python.org>, Barry Warsaw <barry@python.org>

Iterators, generators and decorators

In this chapter we will learn about iterators, generators and decorators.

Iterators

Python iterator objects are required to support two methods while following the iterator protocol.

`__iter__` returns the iterator object itself. This is used in *for* and *in* statements.

`__next__` method returns the next value from the iterator. If there is no more items to return then it should raise *StopIteration* exception.

```
class Counter(object):
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        'Returns itself as an iterator object'
        return self

    def __next__(self):
        'Returns the next value till current is lower than high'
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1
```

Now we can use this iterator in our code.

```
>>> c = Counter(5,10)
```

```
>>> for i in c:
...     print(i, end=' ')
...
5 6 7 8 9 10
```

Remember that an iterator object can be used only once. It means after it raises *StopIteration* once, it will keep raising the same exception.

```
>>> c = Counter(5,6)
>>> next(c)
5
>>> next(c)
6
>>> next(c)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 11, in next
StopIteration
>>> next(c)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 11, in next
StopIteration
```

Using the iterator in for loop example we saw, the following example tries to show the code behind the scenes.

```
>>> iterator = iter(c)
>>> while True:
...     try:
...         x = iterator.__next__()
...         print(x, end=' ')
...     except StopIteration as e:
...         break
...
5 6 7 8 9 10
```

Generators

In this section we learn about Python generators. They were introduced in Python 2.3. It is an easier way to create iterators using a keyword *yield* from a function.

```
>>> def my_generator():
...     print("Inside my generator")
...     yield 'a'
...     yield 'b'
```

```

...     yield 'c'
...
>>> my_generator()
<generator object my_generator at 0x7fbcfa0a6aa0>

```

In the above example we create a simple generator using the `yield` statements. We can use it in a `for` loop just like we use any other iterators.

```

>>> for char in my_generator():
...     print(char)
...
Inside my generator
a
b
c

```

In the next example we will create the same `Counter` class using a generator function and use it in a `for` loop.

```

def counter_generator(low, high):
    while low <= high:
        yield low
        low += 1

>>> for i in counter_generator(5,10):
...     print(i, end=' ')
...
5 6 7 8 9 10

```

Inside the `while` loop when it reaches to the *yield* statement, the value of `low` is returned and the generator state is suspended. During the second *next* call the generator resumed where it freeze-ed before and then the value of `low` is increased by one. It continues with the `while` loop and comes to the *yield* statement again.

When you call an generator function it returns a **generator** object. If you call **dir** on this object you will find that it contains `__iter__` and **__next__** methods among the other methods.

```

>>> c = counter_generator(5,10)
>>> dir(c)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__iter__',
 '__le__', '__lt__', '__name__', '__ne__', '__new__', '__next__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'close', 'gi_code', 'gi_frame', 'gi_running', 'send', 'throw']

```

We mostly use generators for laze evaluations. This way generators become a good approach to work with lots of data. If you don't want to load all the data

in the memory, you can use a generator which will pass you each piece of data at a time.

One of the biggest example of such example is *os.path.walk()* function which uses a callback function and current *os.walk* generator. Using the generator implementation saves memory.

We can have generators which produces infinite values. The following is a one such example.

```
>>> def infinite_generator(start=0):
...     while True:
...         yield start
...         start += 1
...
>>> for num in infinite_generator(4):
...     print(num, end=' ')
...     if num > 20:
...         break
...
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

If we go back to the example of *my_generator* we will find one feature of generators. They are not re-usable.

```
>>> g = my_generator()
>>> for c in g:
...     print(c)
...
Inside my generator
a
b
c
>>> for c in g:
...     print(c)
...
...
```

One way to create a reusable generator is Object based generators which do not hold any state. Any class with a `__iter__` method which yields data can be used as an object generator. In the following example we will recreate our counter generator.

```
>>> class Counter(object):
...     def __init__(self, low, high):
...         self.low = low
...         self.high = high
...     def __iter__(self):
...         counter = self.low
...         while self.high >= counter:
```



```

...             yield counter
...             counter += 1
...
>>> gobj = Counter(5, 10)
>>> for num in gobj:
...     print(num, end=' ')
...
5 6 7 8 9 10
>>> for num in gobj:
...     print(num, end=' ')
...
5 6 7 8 9 10

```

Generator expressions

In this section we will learn about generator expressions which is a high performance, memory efficient generalization of list comprehensions and generators.

For example we will try to sum the squares of all numbers from 1 to 9.

```
>>> sum([x*x for x in range(1,10)])
```

The example actually first creates a list of the square values in memory and then it iterates over it and finally after sum it frees the memory. You can understand the memory usage in case of a big list.

We can save memory usage by using a generator expression.

```
sum(x*x for x in range(1,10))
```

The syntax of generator expression says that always needs to be directly inside a set of parentheses and cannot have a comma on either side. Which basically means both the examples below are valid generator expression usage example.

```

>>> sum(x*x for x in range(1,10))
285
>>> g = (x*x for x in range(1,10))
>>> g
<generator object <genexpr> at 0x7fc559516b90>

```

We can have chaining of generators or generator expressions. In the following example we will read the file `*/var/log/cron*` and will find if any particular job (in the example we are searching for `anacron`) is running successfully or not.

We can do the same using a shell command `tail -f /var/log/cron |grep anacron`

```

>>> jobtext = 'anacron'
>>> all_lines = (line for line in open('/var/log/cron', 'r') )
>>> job = ( line for line in all_lines if line.find(jobtext) != -1)
>>> text = next(job)

```

```
>>> text
'May  6 12:17:15 dhcp193-104 anacron[23052]: Job `cron.daily' terminated\n"
>>> text = next(job)
>>> text
'May  6 12:17:15 dhcp193-104 anacron[23052]: Normal exit (1 job run)\n'
>>> text = next(job)
>>> text
'May  6 13:01:01 dhcp193-104 run-parts(/etc/cron.hourly)[25907]: starting 0anacron\n'
```

You can write a for loop to the lines.

Closures

Closures are nothing but functions that are returned by another function. We use closures to remove code duplication. In the following example we create a simple closure for adding numbers.

```
>>> def add_number(num):
...     def adder(number):
...         'adder is a closure'
...         return num + number
...     return adder
...
>>> a_10 = add_number(10)
>>> a_10(21)
31
>>> a_10(34)
44
>>> a_5 = add_number(5)
>>> a_5(3)
8
```

adder is a closure which adds a given number to a pre-defined one.

Decorators

Decorator is way to dynamically add some new behavior to some objects. We achieve the same in Python by using closures.

In the example we will create a simple example which will print some statement before and after the execution of a function.

```
>>> def my_decorator(func):
...     def wrapper(*args, **kwargs):
...         print("Before call")
...         result = func(*args, **kwargs)
...         print("After call")
```

```
...         return result
...     return wrapper
...
>>> @my_decorator
... def add(a, b):
...     "Our add function"
...     return a + b
...
>>> add(1, 3)
Before call
After call
4
```

Virtualenv

Virtual Python Environment or venv is a Python environment which will help you to install different versions of Python modules in a local directory using which you can develop and test your code without requiring to install everything systemwide.

Installation

In Python3 we can use the **venv** module to create virtual environments.

Usage

We will create a directory call *virtual* inside which we will have two different virtual environment.

The following commands will create an env called virt1.

```
$ cd virtual
$ python3 -m venv virt1
$
```

Now we can activate the virt1 environment.

```
$ source virt1/bin/activate
(virt1)[user@host]$
```

The first part of the prompt is now the name of the virtual environment, it will help you identify which environment you are in when you have multiple environments.

To deactivate the environment use *deactivate* command.

```
(virt1)$ deactivate
$
```

So, now we will install a Python module called redis.

```
(virt1)$ python3 -m pip install redis
Collecting redis
  Downloading redis-2.10.5-py2.py3-none-any.whl (60kB)
    100% || 61kB 607kB/s
Installing collected packages: redis
Successfully installed redis-2.10.5
```

See also

Read this blog post [<https://snarky.ca/why-you-should-use-python-m-pip/>] from Brett Cannon to understand why you should use `python3 -m pip` to install packages.

Now we will create another virtual environment *virt2* where we will install the same redis module but an old 2.4 version of it.

```
$ python3 -m venv virt2
$ source virt2/bin/activate
(virt2)$
(virt2)$ python3 -m pip install redis==2.4
Downloading/unpacking redis
Downloading redis-2.4.0.tar.gz
Running setup.py egg_info for package redis
Installing collected packages: redis
Running setup.py install for redis
Successfully installed redis
Cleaning up...
```

This way you can have many different environments for all of your development needs.

Note

Always remember to create virtualenvs while developing new applications. This will help you keep the system modules clean.

Pipenv

Pipenv [<https://docs.pipenv.org/>] is a tool created by Kenneth Reitz [<https://www.kennethreitz.org/>] which helps to create, manage the virtualenvs for your projects. It also helps to install/uninstall/update the dependencies of your project.

Installing pipenv

We can install pipenv by the following command.

```
$ python3 -m pip install --user pipenv
```

Using pipenv

You can go to your project directory, and then use the command **pipenv install** to create a new virtualenv for you. You can also pass any module name which *pipenv* will install on the environment.

```
$ mkdir myproject
$ cd myproject
$ pipenv install requests
Creating a virtualenv for this project...
Using /usr/bin/python3 (3.6.5) to create virtualenv...
Already using interpreter /usr/bin/python3
Using base prefix '/usr'
New python executable in /home/fedora/.local/share/virtualenvs/myproject-dbBcpQ4l/bin/python3
Also creating executable in /home/fedora/.local/share/virtualenvs/myproject-dbBcpQ4l/bin/python
Installing setuptools, pip, wheel...done.

Virtualenv location: /home/fedora/.local/share/virtualenvs/myproject-dbBcpQ4l
Creating a Pipfile for this project...
Installing requests...
Collecting requests
Downloading https://files.pythonhosted.org/packages/49/df/50aa1999ab9bde74656c2919d9c0c085fd2b377
Collecting chardet<3.1.0,>=3.0.2 (from requests)
Downloading https://files.pythonhosted.org/packages/bc/a9/01ffebfb562e4274b6487b4bb1ddec7ca55ec75
Collecting idna<2.7,>=2.5 (from requests)
Downloading https://files.pythonhosted.org/packages/27/cc/6dd9a3869f15c2edfab863b992838277279ce92
Collecting certifi>=2017.4.17 (from requests)
Using cached https://files.pythonhosted.org/packages/7c/e6/92ad559b7192d846975fc916b65f667c7b8c3a
Collecting urllib3<1.23,>=1.21.1 (from requests)
Downloading https://files.pythonhosted.org/packages/63/cb/6965947c13a94236f6d4b8223e21beb4d576dc7
Installing collected packages: chardet, idna, certifi, urllib3, requests
Successfully installed certifi-2018.4.16 chardet-3.0.4 idna-2.6 requests-2.18.4 urllib3-1.22

Adding requests to Pipfile's [packages]...
Pipfile.lock not found, creating..
Locking [dev-packages] dependencies...
Locking [packages] dependencies...
Updated Pipfile.lock (b14837)!
Installing dependencies from Pipfile.lock (b14837)...
 5/5 - 00:00:02
To activate this project's virtualenv, run the following:
$ pipenv shell
```

The above command will create a new virtualenv and then also install *requests* module in the environment. You can then use **pipenv shell** command to activate that environment. For our example, we will use the following Python code in a file named *main.py*.

```
import requests
response = requests.get('https://httpbin.org/ip')
print('Your IP is {0}'.format(response.json()['origin']))

$ pipenv shell
$ $ python main.py
Your IP is 192.168.1.2
```

Exiting from the virtualenv

You can exit from the virtualenv using **exit** command, or by pressing *Ctrl+d*.

Pipfile and Pipfile.lock

If you notice your project directory after you have used **pipenv**, you will find two new files inside, *Pipfile* and *Pipfile.lock*. These files have been created by the **pipenv** command. You should checkin these two files into your version control system (say: git), so that others can create the exact same environment of yours.

Pipfile

The following is the content of our *Pipfile*. It is using the TOML [<https://en.wikipedia.org/wiki/TOML>] file format.

```
[[source]]
verify_ssl = true
name = "pypi"
url = "https://pypi.python.org/simple"

[dev-packages]

[requires]
python_version = "3.6.5"

[packages]
requests = "*"
```

On the top it tells which source to use to get the packages. It also mentions the Python version required. The packages section tells us what all Python packages we need. The string “*” means install the latest version available on the package index. The exact version details of the packages are stored in the *Pipfile.lock* file, it is in machine readable JSON [<https://en.wikipedia.org/wiki/JSON>] format.

Remember to install any dependency for your project using **pipenv** comamnd, that will automatically update your *Pipfile* and *Pipfile.lock* file. If you have any dependency which is only required for the development, you can install

them marked as *dev-packages*. In the following example I am installing *flake8* as development dependency.

```
$ pipenv install --dev flake8
$ cat Pipfile
[[source]]
verify_ssl = true
name = "pypi"
url = "https://pypi.python.org/simple"

[dev-packages]
"flake8" = "*"

[requires]
python_version = "3.6.5"

[packages]
requests = "*"
```

You can watch this talk [<https://www.youtube.com/watch?v=GBQAKldqgZs>] by Kenneth from PyCon 2018 to know more about *Pipenv*.

Through out the rest of the book, we will use **pipenv** to create and manage virtualenvs for any code.

Type hinting and annotations

This is one of the new feature of the language. We can do the similar kind of work in Python2 also, but with different syntax. Please remember that Python will stay as a dynamically typed language, this type hinting does not effect your code anyway.

The major benefit of having type hints in your codebase is about future maintenance of the codebase. When a new developer will try to contribute to your project, having type hints will save a lot of time for that new person. It can also help to detect some of the runtime issues we see due to passing of wrong variable types in different function calls.

First example of type annotation

Let us start with a simple example, adding of two integers.

```
def add(a, b):  
    return a + b
```

Now, the above example will work for any object which supports `+` operator. But, we want to specify that it is expecting only Integers as parameters, and the function call will return another Integer.

```
def add(a: int, b: int) -> int:  
    return a + b
```

You can see that the return type of the function call is defined after `->`. We can do the same in Python2 using a comment (before any docstring).

```
def add(a, b):  
    # type: (int, int) -> int  
    return a + b
```

Using mypy and more examples

Mypy [<https://mypy.rtfd.io>] is a static type checker written for Python. If we use the type annotations as explained above, mypy can help by finding common problems in our code. You can use mypy in various ways in your development workflow, may be in CI as a proper test.

Installing mypy

We can install mypy inside of a virtual environment.

```
$ pipenv install mypy
Installing mypy...
Looking in indexes: https://pypi.python.org/simple
Collecting mypy
Downloading https://files.pythonhosted.org/packages/e2/3f/e20e2544b35e862fbed4e26a89e3...
Collecting typed-ast<1.2.0,>=1.1.0 (from mypy)
Downloading https://files.pythonhosted.org/packages/5b/4e/79e873aa89b8038ca6474c00afe9...
Installing collected packages: typed-ast, mypy
Successfully installed mypy-0.600 typed-ast-1.1.0

Adding mypy to Pipfile's [packages]...
Pipfile.lock (627f99) out of date, updating to (67e074)...
Locking [dev-packages] dependencies...
Locking [packages] dependencies...
Updated Pipfile.lock (67e074)!
Installing dependencies from Pipfile.lock (67e074)...
 7/7 - 00:00:01
To activate this project's virtualenv, run the following:
$ pipenv shell
```

Our example code

We will be working on the following example code. This isn't much useful, but we can use this to learn about type annotations and mypy.

```
class Student:

    def __init__(self, name, batch, branch, roll):
        self.name = name
        self.batch = batch
        self.branch = branch
        self.roll = roll
        self.semester = None
        self.papers = {}

    def is_passed(self):
```

```

        "To find if the student has pass the exam in the current semester"
        for k, v in self.papers.items():
            if v < 34:
                return False

        return True

    def total_score(self):
        "Returns the total score of the student"
        total = 0
        for k, v in self.papers.items():
            total += v

        return total

std1 = Student("Kushal", 2005, "cse", "123")
std2 = Student("Sayan", 2005, "cse", 121)
std3 = Student("Anwesha", 2005, "law", 122)

std1.papers = {"english": 78, "math": 82, "science": 77}
std2.papers = {"english": 80, "math": 92, "science": "78"}
std3.papers = {"english": 82, "math": 87, "science": 77}

for std in [std1, std2, std3]:
    print("Passed: {0}. The total score of {1} is {2}".format(std.is_passed(), std.name, std.total_score()))

```

You may find some errors in the code, but in case of a large codebase we can not detect the similar issues unless we see the runtime errors.

Using mypy

We can just call mypy on our source file, I named it as *students2.py*

```
$ mypy students2.py
```

Enabling the first few type annotations

We will add some type annotations to the `__init__` method. For reducing the code length, I am only showing the changed code below.

```

class Student:

    def __init__(self, name: str, batch: int, branch: str, roll: int) -> None:
        self.name = name
        self.batch = batch

```

```

        self.branch = branch
        self.roll = roll
        self.semester = None
        self.papers = {}

$ mpy students2.py
students2.py:11: error: Need type annotation for variable
students2.py:31: error: Argument 4 to "Student" has incompatible type "str"; expected

```

You can see mypy is complaining about variable which does not have type annotations, and also found that in line 31, as argument 4 we are passing *str*, where as we were supposed to send in an Integer for the roll number. Let us fix these.

```

from typing import Dict

class Student:

    def __init__(self, name: str, batch: int, branch: str, roll: int) -> None:
        self.name = name
        self.batch = batch
        self.branch = branch
        self.roll = roll
        self.semester: str = None
        self.papers: Dict[str, int] = {}

    def is_passed(self) -> bool:
        "To find if the student has pass the exam in the current semester"
        for k, v in self.papers.items():
            if v < 34:
                return False

        return True

    def total_score(self) -> int:
        "Returns the total score of the student"
        total = 0
        for k, v in self.papers.items():
            total += v

        return total

std1: Student = Student("Kushal", 2005, "cse", 123)
std2: Student = Student("Sayan", 2005, "cse", 121)
std3: Student = Student("Anwesha", 2005, "law", 122)

```

```
std1.papers = {"english": 78, "math": 82, "science": 77}
std2: Student.papers = {"english": 80, "math": 92, "science": 78}
std3.papers = {"english": 82, "math": 87, "science": 77}

for std in [std1, std2, std3]:
    print("Passed: {0}. The total score of {1} is {2}".format(std.is_passed(), std.name, std.total_score))

$ mpy students2.py
```

Now, it does not complain about any error. You can see that in line 1, we imported Dict from the typing module. And, then using the same we added the type annotation of the *self.paper* variable. We are saying that it is a dictionary which has string keys, and Integers as values. We also used our *Student* class as type of std1, std2, and std3 variables.

Now let us say we by mistake assign a new list to the papers variable.

```
std1.papers = ["English", "Math"]

Or maybe assigned a wrong kind of dictionary.

std2.papers = {1: "English", 2: "Math"}
```

We can see what mypy says in these cases

```
$ mpy students2.py
students2.py:35: error: Incompatible types in assignment (expression has type List[str], variable has type Dict[str, int])
students2.py:36: error: Dict entry 0 has incompatible type "int": "str"
students2.py:36: error: Dict entry 1 has incompatible type "int": "str"
```

More examples of type annotations

```
from typing import List, Tuple, Sequence, Optional

values: List[int] = []
city: int = 350 # The city code, not a name

# This function returns a Tuple of two values, a str and an int
def get_details() -> Tuple[str, int]:
    return "Python", 5

# The following is an example of Tuple unpacking
name: str
marks: int
name, marks = get_details()
```

```
def print_all(values: Sequence) -> None:
    for v in values:
        print(v)

print_all([1,2,3])
print_all({"name": "kushal", "class": 5})
# alltypes.py:23: error: Argument 1 to "print_all" has incompatible type Dict[str, obj
# But running the code will give us no error with wrong output

def add_ten(number: Optional[int] = None) -> int:
    if number:
        return number + 10
    else:
        return 42

print(add_ten())
print(add_ten(12))
```

You can learn more about types from PEP 484 [<https://www.python.org/dev/peps/pep-0484/>]. The typing module [<https://docs.python.org/3/library/typing.html>] has detailed explanation and more examples about how to add type annotations in your codebase.

You can also view the talk [<https://www.youtube.com/watch?v=pMgmKJyWKn8>] from Carl Meyer to learn about type checking in Python.

Simple testing in Python

What we should test ?

If possible everything in our codebase, each and every function. But it depends as a choice of the developers. You can skip it if it is not practical to write a robust test. As Nick Coghlan [<http://www.curious efficiency.org>] said in a guest session – ... *with a solid test suite, you can make big changes, confident that the externally visible behavior will remain the same*

To have effective tests, you should remember to write/split your code in smaller functions which can be tested separately. It is very easy to keep writing longer functions, which can do a lot of things at once. But, it will be increasingly difficult to test those functions. If you keep them short, and make sure that one function does one thing well, it will help to write better test cases.

Unit testing

A method by which individual units of source code. Wikipedia [http://en.wikipedia.org/wiki/Unit_testing] says *In computer programming, unit testing is a method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine if they are fit for use.*

unittest module

In Python we have `unittest` module to help us.

Factorial code

In this example we will write a file `factorial.py`.

```
import sys

def fact(n):
```

```

    """
    Factorial function

    :arg n: Number
    :returns: factorial of n

    """
    if n == 0:
        return 1
    return n * fact(n - 1)

def div(n):
    """
    Just divide
    """
    res = 10 / n
    return res

def main(n):
    res = fact(n)
    print(res)

if __name__ == '__main__':
    if len(sys.argv) > 1:
        main(int(sys.argv[1]))

```

Output

```
$ python factorial.py 5
```

Which function to test ?

As you can see `fact(n)` is function which is doing all calculations, so we should test that at least.

Our first test case

Now we will write our first test case.

```

import unittest
from factorial import fact

class TestFactorial(unittest.TestCase):
    """
    Our basic test class

```



```

"""

def test_fact(self):
    """
    The actual test.
    Any method which starts with ``test_`` will considered as a test case.
    """
    res = fact(5)
    self.assertEqual(res, 120)

if __name__ == '__main__':
    unittest.main()

```

Running the test:

```
$ python factorial_test.py
```

```
.
```

```
-----
Ran 1 test in 0.000s
```

```
OK
```

Description

We are importing `unittest` module first and then the required functions which we want to test.

A testcase is created by subclassing `unittest.TestCase`.

Now open the test file and change `120` to `121` and see what happens :)

Different assert statements

Method	Checks that	New in
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x) is True</code>	
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	
<code>assertIs(a, b)</code>	<code>a is b</code>	2.7
<code>assertIsNot(a, b)</code>	<code>a is not b</code>	2.7
<code>assertIsNone(x)</code>	<code>x is None</code>	2.7
<code>assertIsNotNone(x)</code>	<code>x is not None</code>	2.7
<code>assertIn(a, b)</code>	<code>a in b</code>	2.7
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	2.7

Method	Checks that	New in
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	2.7
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	2.7

Testing exceptions

If we call `div(0)` in `factorial.py`, we can see it raises an exception.

We can also test these exceptions, like:

```
self.assertRaises(ZeroDivisionError, div, 0)
```

Full code

```
import unittest
from factorial import fact, div

class TestFactorial(unittest.TestCase):
    """
    Our basic test class
    """

    def test_fact(self):
        """
        The actual test.
        Any method which starts with ``test_`` will be considered as a test case.
        """
        res = fact(5)
        self.assertEqual(res, 120)

    def test_error(self):
        """
        To test exception raised due to run time error
        """
        self.assertRaises(ZeroDivisionError, div, 0)

if __name__ == '__main__':
    unittest.main()
```

Note

The following example is for Linux only, you will have to modify the code so that it can find mount details in other operating systems properly.

mounttab.py

Here we have only one function *mount_details()* doing the parsing and printing mount details.

```
import os

def mount_details():
    """
    Prints the mount details
    """
    if os.path.exists('/proc/mounts'):
        fd = open('/proc/mounts')
        for line in fd:
            line = line.strip()
            words = line.split()
            print(f'{words[0]} on {words[1]} type {words[2]}', end=' ')
            if len(words) > 5:
                print('(%s)' % ' '.join(words[3:-2]))
            else:
                print('')

if __name__ == '__main__':
    mount_details()
```

After refactoring

Now we refactored the code and have one new function *parse_mounts* which we can test easily.

```
import os

def parse_mounts():
    """
    It parses /proc/mounts and returns a list of tuples
    """
    result = []
    if os.path.exists('/proc/mounts'):
        fd = open('/proc/mounts')
        for line in fd:
            line = line.strip()
            words = line.split()
            if len(words) > 5:
                res = (words[0], words[1], words[2], '(%s)' % ' '.join(words[3:-2]))
```

```

        else:
            res = (words[0], words[1], words[2])
            result.append(res)
    return result

def mount_details():
    """
    Prints the mount details
    """
    result = parse_mounts()
    for line in result:
        if len(line) == 4:
            print('%s on %s type %s %s' % line)
        else:
            print('%s on %s type %s' % line)

if __name__ == '__main__':
    mount_details()

and the test code for the same.

#!/usr/bin/env python
import unittest
from mounttab2 import parse_mounts

class TestMount(unittest.TestCase):
    """
    Our basic test class
    """

    def test_parsemount(self):
        """
        The actual test.
        Any method which starts with ``test_`` will considered as a test case.
        """
        result = parse_mounts()
        self.assertIsInstance(result, list)
        self.assertIsInstance(result[0], tuple)

    def test_rootext4(self):
        """
        Test to find root filesystem
        """
        result = parse_mounts()
        for line in result:
            if line[1] == '/' and line[2] != 'rootfs':

```

```

        self.assertEqual(line[2], 'ext4')

if __name__ == '__main__':
    unittest.main()

```

```
$ python mounttest.py
```

```
..
```

```
-----
Ran 2 tests in 0.001s
```

```
OK
```

Test coverage

Test coverage is a simple way to find untested parts of a codebase. It does not tell you how good your tests are.

In Python we already have a nice coverage tool to help us. You can install it in Fedora

```
# dnf install python3-coverage
```

Or using pip.

```
$ python3 -m pip install coverage
```

Coverage Example

```
$ coverage -x mounttest.py
```

```
<OUTPUT snipped>
```

```
$ coverage -rm
```

Name	Stmts	Miss	Cover	Missing
mounttab2	21	7	67%	16, 24-29, 33
mounttest	14	0	100%	
TOTAL	35	7	80%	

A project structure

This chapter explains a full Python project structure. What kind of directory layout you can use and how make release a software to the world.

We will call our example project *factorial*.

```
$ mkdir pymfactorial
$ cd pymfactorial/
```

Primary code

The name of the Python module will be *pymfactorial*, so we will create the directory next.

```
$ mkdir -p pymfactorial/pymfactorial
$ cd pymfactorial/
```

The primary code will be in a file called *fact.py*

```
import sys

def factorial(num):
    """
    Returns the factorial value of the given number.

    :arg num: Interger value of whose factorial we will calculate.

    :return: The value of the the factorial or -1 in case negative value passed.
    """
    if num >= 0:
        if num == 0:
            return 1
        return num * factorial(num -1)
    else:
        return -1
```

```
def cli():
    "The command line entry point"
    number = int(sys.argv[1])
    print(f"Factorial is {factorial(number)}")
```

We also have a `__init__.py` file for the module.

```
"pymfactorial module and command line tool"
```

```
__version__ = "0.1.0"
```

```
from fact import factorial
__all__ = [factorial, cli]
```

We also added a *README.md* file. So, the directory structure looks like

```
$ ls
pymfactorial  README.md
$ ls pymfactorial/
fact.py  __init__.py
```

LICENSE file

Always remember to add a proper license to your project. You can take help from this site [<https://choosealicense.com/>] if you are new to software licensing.

The following is the content of our project, which is licensed under **MIT**.

MIT License

Copyright (c) 2021 Kushal Das <kushaldas@gmail.com>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Installing flit package

You have to install *flit* package in your system. For this we are using a virtualenv. We will also install *wheel* package.

```
$ python3 -m venv .venv
$ source .venv/bin/activate
$ python3 -m pip install flit wheel
```

pyproject.toml

Finally we have to write a *pyproject.toml* which then can be used to create a source tarball or installing the software, or create a wheel to be uploaded to PyPI.

`flit init` command will help to create the initial file for us, provide the proper input as it asks.

```
$ flit init
Module name [pymfactorial]:
Author [Kushal Das]:
Author email [kushal@sunset.se]: mail@kushaldas.in
Home page [https://github.com/kushaldas/pymfactorial]: https://github.com/kushaldas/pym
Choose a license (see http://choosealicense.com/ for more info)
1. MIT - simple and permissive
2. Apache - explicitly grants patent rights
3. GPL - ensures that code based on this is shared with the same terms
4. Skip - choose a license later
Enter 1-4 [4]: 1
```

Written `pyproject.toml`; edit that file to add optional extra info.

This command created the `pyproject.toml` file in the same folder, we can check the content.

```
[build-system]
requires = ["flit_core >=3.2,<4"]
build-backend = "flit_core.buildapi"

[project]
name = "pymfactorial"
authors = [{name = "Kushal Das", email = "mail@kushaldas.in"}]
readme = "README.md"
license = {file = "LICENSE"}
```



```
classifiers = ["License :: OSI Approved :: MIT License"]
dynamic = ["version", "description"]
```

```
[project.urls]
Home = "https://github.com/kushaldas/pym"
```

Next we will update the file to make sure that we include the LICENSE and README.md file in the source tarball. We also mark the cli function as the starting point for the command line tool.

```
[build-system]
requires = ["flit_core >=3.2,<4"]
build-backend = "flit_core.buildapi"
```

```
[project]
name = "pymfactorial"
authors = [{name = "Kushal Das", email = "mail@kushaldas.in"}]
readme = "README.md"
license = {file = "LICENSE"}
classifiers = ["License :: OSI Approved :: MIT License"]
dynamic = ["version", "description"]
```

```
[project.urls]
Home = "https://github.com/kushaldas/pym"
```

```
[project.scripts]
myfact = "pymfactorial:cli"
```

```
[tool.flit.sdist]
include = ["LICENSE", "README.MD"]
```

Please read flit metadata documentation [https://flit.readthedocs.io/en/latest/pyproject_toml.html#new-style-metadata] for details of the various keys and their values mentioned above.

Building a package

To create a source release and also a binary wheel for distribution, use the following command.

```
$ flit build
```

One can see the output files under *dist* directory.

```
$ ls dist/
```

Warning

Remember to use a virtualenv while trying to install the code :)

Python Package Index (PyPI)

Do you remember the **pip** command we are using still now? Ever thought from where those packages are coming from? The answer is PyPI [<http://pypi.org/>]. It is a repository of software for the Python programming language.

For our example, we will use the test server of PyPI which is <https://test.pypi.org/>

Creating account

First register yourself in this link [<https://test.pypi.org/account/register/>]. You will receive an email with a link, go to that link and confirm your registration.

Note

Remember to change the name of the project to something else in the setup.py to test following instructions.

Uploading your project

Now finally we can upload our project to the PyPI server using **twine** command. Remember that this command needs to be invoked immediately after you build the source/binary distribution files.

First, we will have to install **twine** using **pip** (we are using a virtualenv).

```
$ python3 -m pip install twine
$ twine upload --repository-url https://test.pypi.org/legacy/ dist/*
Uploading distributions to https://test.pypi.org/legacy/
Enter your username: kushaldas
Enter your password:
Uploading pymfactorial-0.1-py3-none-any.whl
100%|| 4.29k/4.29k [00:01<00:00, 3.77kB/s]
Uploading pymfactorial-0.1.tar.gz
100%|| 3.83k/3.83k [00:00<00:00, 7.57kB/s]
```

Now if you visit the site [<https://test.pypi.org/pypi/pymfactorial/>], you will find your project is ready to be used by others.

Install from the test PyPI

You can use the following command to install from the test PyPI.

```
$ python3 -m pip install --index-url https://test.pypi.org/simple/ pymfactorial
```

More readings

Please visit <https://packaging.python.org> to learn more about Python packaging. There are many guides and tutorials available on that site. PEP-621 [<https://www.python.org/dev/peps/pep-0621/>] is also an important read.

Building command line applications with Click

I recommend click [<http://click.pocoo.org/>] module to build command line applications. Like any other project from Armin Ronacher [<http://lucumr.pocoo.org/about/>], it has great documentation. In this post, I am going to write a beginners tutorial, you should the read the documentation for any more details and examples.

Installation, and development tips

Using virtualenv is highly recommended for developing “click” applications. I am going to assume that we are in an empty directory and the continue from there. To start, we will have a simple hello.py file with the following content:

```
def cli():
    print("Hello World")
```

Now we will need a setup.py file. This will help us to use the python module we are writing as a command line tool. It is also the recommended way to write command line tools in python, then directly using shebang based scripts.

```
from setuptools import setup
```

```
setup(
    name="myhello",
    version='0.1',
    py_modules=['hello'],
    install_requires=[
        'Click',
    ],
    entry_points='''
        [console_scripts]
        myhello=hello:cli
    ''',
```

)

You can see that we mentioned the starting point of our tool in the entry_points, *hello:cli* points to the right function to start with. We can then install this on the virtualenv locally. I will also create the virtualenv below so that becomes easier others. To learn more, read this chapter [<http://click.pocoo.org/5/setuptools/#setuptools-integration>] later.

```
$ python3 -m venv env
$ source env/bin/activate
$ python3 -m pip install --editable .
Obtaining file:///home/kdas/code/practice/yoclick
Collecting Click (from myhello==0.1)
Using cached click-6.7-py2.py3-none-any.whl
Installing collected packages: Click, myhello
Running setup.py develop for myhello
Successfully installed Click-6.7 myhello
```

```
$ myhello
Hello World
```

Now to convert the same script into a click based tool, we will make the following modifications:

```
import click

@click.command()
def cli():
    print("Hello World")
```

Now when we execute the command again, we see nothing changed visually, but it magically has a *-help* command line argument (which is optional).

```
$ myhello
Hello World
$ myhello --help
Usage: myhello [OPTIONS]
```

```
Options:
--help  Show this message and exit.
```

Using echo for printing text

The click module suggests using *echo* function to print, rather than the standard print function. So, we will make the required change in our code.

```
import click
```

```
@click.command()
def cli():
    click.echo("Hello World")
```

Boolean flags

In a command line tool, we sometimes want to have a boolean option. If the option is provided then do something, if not provided, then do something else. In our example, we will call the flag as *-verbose*, if it is provided, then we will print some extra text.

```
import click

@click.command()
@click.option('--verbose', is_flag=True, help="Will print verbose messages.")
def cli(verbose):
    if verbose:
        click.echo("We are in the verbose mode.")
    click.echo("Hello World")
```

We added another decorator to the cli function. In *click.option()* decorator, first we passed the flag using *-verbose*, then marked this option as a boolean flag, and then finally added the help text.

```
$ myhello --help
Usage: myhello [OPTIONS]

Options:
  --verbose  Will print verbose messages.
  --help     Show this message and exit.
$ myhello --verbose
We are in the verbose mode.
Hello World
```

By default, any boolean flag is treated as false.

Standard options in the command line

We can now add more options to our tool. For example, we will have a *-name* option which will take a string as input.

```
import click

@click.command()
@click.option('--verbose', is_flag=True, help="Will print verbose messages.")
@click.option('--name', default='', help='Who are you?')
def cli(verbose, name):
```

```

    if verbose:
        click.echo("We are in the verbose mode.")
    click.echo("Hello World")
    click.echo('Bye {0}'.format(name))

$ myhello --help
Usage: myhello [OPTIONS]

```

```

Options:
  --verbose    Will print verbose messages.
  --name TEXT  Who are you?
  --help       Show this message and exit.

$ myhello
Hello World
Bye
$ myhello --name kushal
Hello World
Bye kushal

```

Same option multiple times

We may want to take the same option multiple times. Click has a very simple way to do so.

```

import click

@click.command()
@click.option('--verbose', is_flag=True, help="Will print verbose messages.")
@click.option('--name', '-n', multiple=True, default='', help='Who are you?')
def cli(verbose, name):
    if verbose:
        click.echo("We are in the verbose mode.")
    click.echo("Hello World")
    for n in name:
        click.echo('Bye {0}'.format(n))

```

In the above example, you can see that we specified the *-name* as a multiple options. It also means the name parameter in the *cli* function is now a tuple.

Help text for the script

We can add help text for the script using python docstrings. For example:

```

import click

@click.command()

```

*SUPER FAST WAY TO ACCEPT PASSWORD WITH CONFIRMATION*181

```
@click.option('--verbose', is_flag=True, help="Will print verbose messages.")
@click.option('--name', '-n', multiple=True, default='', help='Who are you?')
def cli(verbose,name):
    """This is an example script to learn Click."""
    if verbose:
        click.echo("We are in the verbose mode.")
    click.echo("Hello World")
    for n in name:
        click.echo('Bye {}'.format(n))
```

```
$ myhello --help
Usage: myhello [OPTIONS]
```

This is an example script to learn Click.

```
Options:
  --verbose      Will print verbose messages.
  -n, --name TEXT Who are you?
  --help        Show this message and exit.
```

Super fast way to accept password with confirmation

Click provides a *password_option()* decorator, which can be used to accept a password over hidden prompt and second confirmation prompt. By the way, I am printing the password here as an example, never print the password to stdout in any tool.

```
import click

@click.command()
@click.option('--verbose', is_flag=True, help="Will print verbose messages.")
@click.option('--name', '-n', multiple=True, default='', help='Who are you?')
@click.password_option()
def cli(verbose,name, password):
    """This is an example script to learn Click."""
    if verbose:
        click.echo("We are in the verbose mode.")
    click.echo("Hello World")
    for n in name:
        click.echo('Bye {}'.format(n))
    click.echo('We received {} as password.'.format(password))
```

The output looks like below:

```
$ myhello --help
```


Usage: myhello [OPTIONS]

This is an example script to learn Click.

Options:

```
--verbose      Will print verbose messages.
-n, --name TEXT Who are you?
--password TEXT
--help         Show this message and exit.
```

\$ myhello

Password:

Repeat for confirmation:

Hello World

We received hello as password.

To learn the full usage of password prompts, read this section [<http://click.pocoo.org/5/options/#password-prompts>].

Must have arguments

You can also add arguments to your tool. These are must haves, and if no default value is provided, they are assumed to be strings. In the below example, the script is expecting a county name to be specified.

```
import click

@click.command()
@click.option('--verbose', is_flag=True, help="Will print verbose messages.")
@click.option('--name', '-n', multiple=True, default='', help='Who are you?')
@click.argument('country')
def cli(verbose, name, country):
    """This is an example script to learn Click."""
    if verbose:
        click.echo("We are in the verbose mode.")
    click.echo("Hello {}".format(country))
    for n in name:
        click.echo('Bye {}'.format(n))
```

The output looks like:

\$ myhello

Usage: myhello [OPTIONS] COUNTRY

Error: Missing argument "country".

\$ myhello India

Hello India

Click has many other useful features, like *yes parameter*, *file path input*. I am not going to write about all of those here, but you can always read it from the upstream documentation [<http://click.pocoo.org/>].

Simple GUI application development using PyperCard

In this chapter we will learn about creating very simple GUI application using [PyperCard](<https://pypercard.readthedocs.io/en/latest/>). PyperCard is a HyperCard inspired Pythonic GUI framework for beginner programmers.

Installing PyperCard in a virtualenv

The first step would be installing PyperCard in a virtualenv.

```
python3 -m venv venv
source venv/bin/activate
python3 -m pip install pypercard
```

This may take some time, and specially while building Kivy, which is a dependency.

If you see any error in building *Kivy* on your distribution, you will have to install all dependencies on your operating system.

On Debian Buster

```
sudo apt install libsdl2-dev libsdl2-image-dev libsdl2-mixer-dev libsdl2-ttf-dev pkg-config libgl1
```

Hello World example

```
1from pypercard import Card, CardApp
2
3
4card = Card("hello", text="Hello, World!", text_color="yellow")
5
6app = CardApp(stack=[card, ])
```

```
7app.run()
```

In the first line, we are importing two classes from the module. **Card** is the class for every screen in our application, and **CardApp** is the primary application.

Note

Remember that each card in your application must have an unique name.

If you execute the code, `python3 hello.py`, you will see the following GUI window. It does not do much, it shows the string “Hello World!” with the text color we mentioned (Yellow).

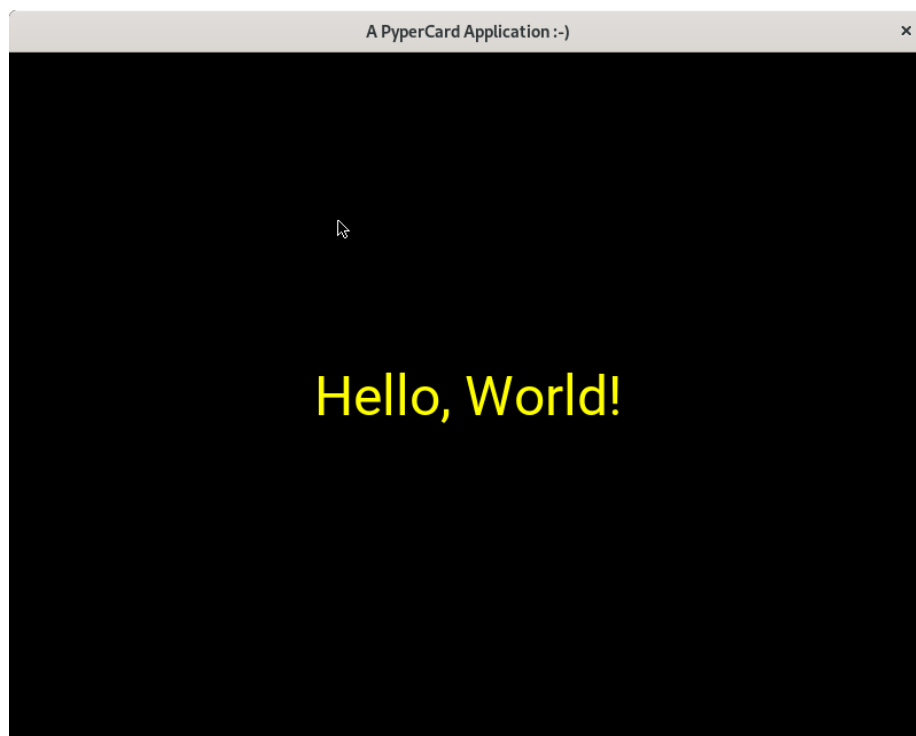


Figure 15:

Two Cards and a button

```
1from pypercard import Card, CardApp
2
3
4first_card = Card(
5    "first_card",
6    text="Hello, World!",
```

```
7     text_color="yellow",
8     buttons=[{"label": "Next", "target": "byescreen"}],
9)
10card2 = Card("byescreen", text="Hack the planet!", text_color="white")
11
12app = CardApp(stack=[first_card, card2])
13app.run()
```

In this code, we have two different cards. The `first_card` also got a button, with a text *Next*, and the *target* as the name of the next card to show. In this case, we are showing the card named *bye*

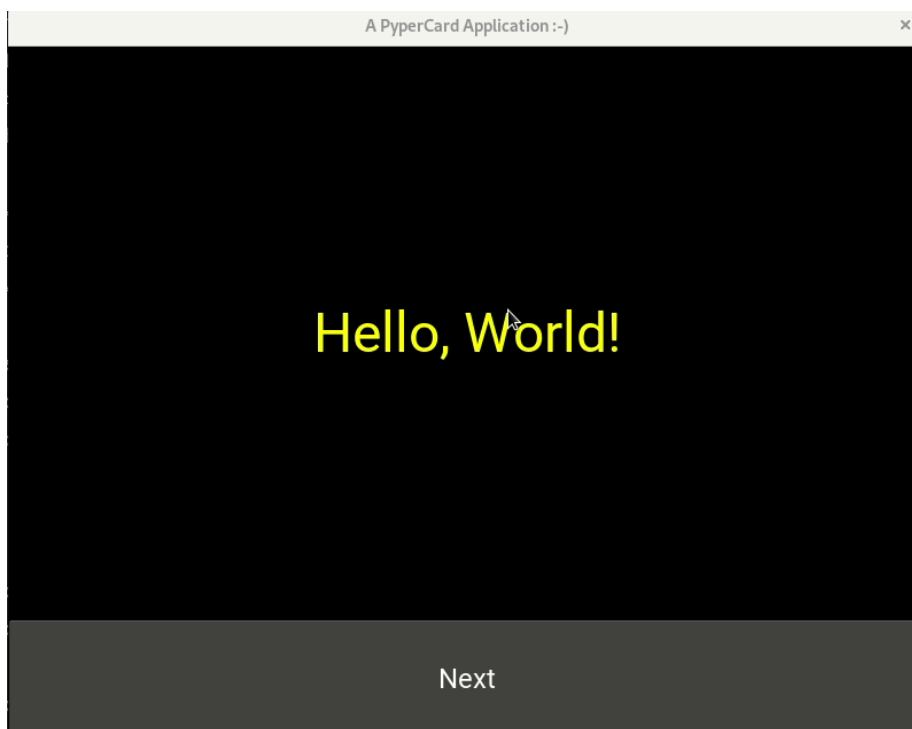


Figure 16:

Taking simple text input

In this example, we will learn how to take simple text input.

```
1from pypercard import Card, CardApp, Inputs
2
3
4def store_name(data_store, form_value):
```

```

5     """
6     Stores the user input in the data_store dictionary
7     """
8     if form_value:
9         data_store["name"] = form_value
10        return "showname"
11    else:
12        return "error"
13
14
15card = Card(
16    "start",
17    text="Enter your name",
18    text_color="yellow",
19    form=Inputs.TEXTBOX,
20    buttons=[{"label": "Next", "target": store_name}],
21)
22
23card2 = Card("showname", text="May the force be with you {name}.", text_color="white",
24
25errorcard = Card(
26    "error",
27    text="You forgot to enter a name",
28    text_color="RED",
29    buttons=[{"label": "Start again", "target": "start"}],
30)
31app = CardApp(stack=[card, card2, errorcard])
32app.run()

```

At line 1, we are importing **Inputs**, which has various graphical objects to take user input.

In line 19, you can see that we passed a new keyword argument to the `card` object called *form* and saying that it is a `Inputs.TEXTBOX`. Another big change is that as a target of the button, we are passing a function called `store_name`. This can be any function which takes two arguments, first one is called `data_store`, a dictionary passed from the application itself, and then `form_value` is the actual user input via the card. Here, we are checking if there is a proper user input, then we are storing the value in the dictionary with the key `name` and returning the next card name `showname`, otherwise we are showing the `error` card from the stack.

In line 23, we are creating the `showname` card, where you can see we can directly use any keyname of `data_store`.

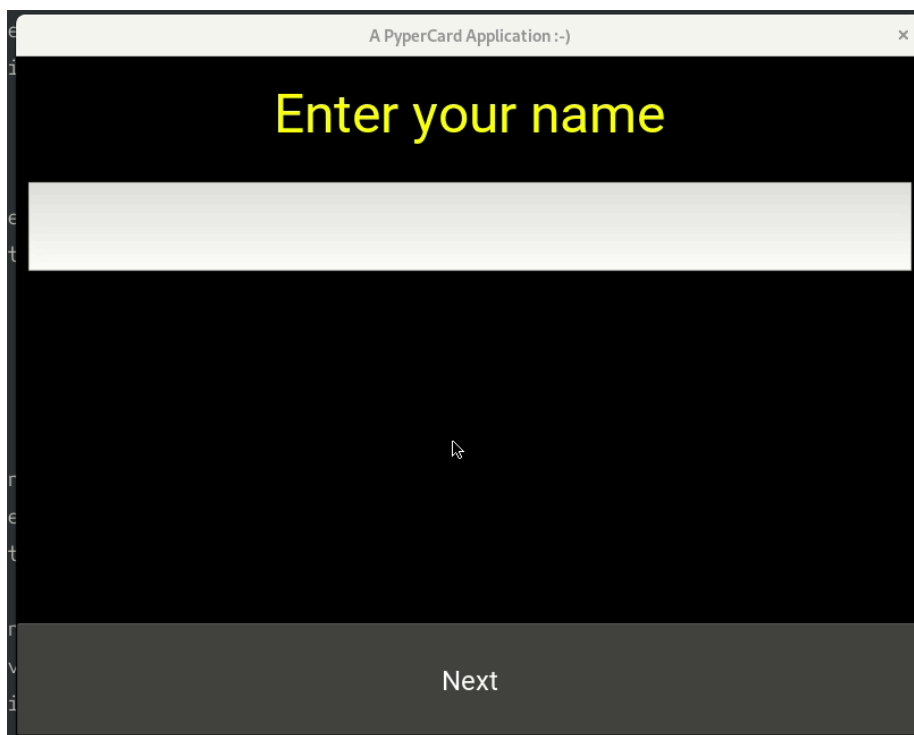


Figure 17:

Check name problem

Here is a small problem for you, change the code such a way, so that if the name starts with **Python**, then the next screen will tell us the version of Python we are running.



Figure 18:

Check name solution

```

1from pypercard import Card, CardApp, Inputs
2import sys
3
4
5def store_name(data_store, form_value):
6    """
7    Stores the user input in the data_store dictionary
8    """
9    if form_value:
10        if form_value.startswith("Python"):
11            data_store["msg"] = sys.version.replace("\n", "")

```



```
12         else:
13             data_store["msg"] = "May the force be with you {0}.".format(form_value)
14             return "showname"
15     else:
16         return "error"
17
18
19 card = Card(
20     "start",
21     text="Enter your name",
22     text_color="yellow",
23     form=Inputs.TEXTBOX,
24     buttons=[{"label": "Next", "target": store_name}],
25 )
26 card2 = Card(
27     "showname",
28     text="{msg}",
29     text_color="white",
30     buttons=[{"label": "Start again", "target": "start"}],
31 )
32
33 errorcard = Card(
34     "error",
35     text="You forgot to enter a name",
36     text_color="RED",
37     buttons=[{"label": "Start again", "target": "start"}],
38 )
39 app = CardApp(stack=[card, card2, errorcard])
40 app.run()
```

Introduction to Flask

What is flask?

Flask [<http://flask.pocoo.org/>] is a web framework. This means flask provides you with tools, libraries and technologies that allow you to build a web application. This web application can be some web pages, a blog, a wiki or go as big as a web-based calendar application or a commercial website.

Flask is part of the categories of the micro-framework. Micro-framework are normally framework with little to no dependencies to external libraries. This has pros and cons. Pros would be that the framework is light, there are little dependency to update and watch for security bugs, cons is that some time you will have to do more work by yourself or increase yourself the list of dependencies by adding plugins. In the case of Flask, its dependencies are:

- Werkzeug [<http://werkzeug.pocoo.org/>] a WSGI utility library
- jinja2 [<http://jinja.pocoo.org/>] which is its template engine

Note

WSGI is basically a protocol defined so that Python application can communicate with a web-server and thus be used as web-application outside of CGI.

What are template engines?

Have you ever built a website? Did you face the problem that to keep the style of the website consistent, you have had to write multiple times the same text? Did you ever tried to change the style of such website?

If your website contains only few pages, changing its style will take you some time but is doable. However, if you have a lot of pages (for example the list of items you sell in your store), this task become overwhelming.

Using templates you are able to set a basic layout for your pages and mention which element will change. This way you can define your header once and keep

it consistent over all the pages of your website, and if you need to change your header, you will only have to update it in one place.

Using a template engine will save you a lot of time when creating your application but also when updating and maintaining it.

A “Hello world” application in flask

We are going to perform a very basic application with flask.

- Create the structure of the project

```
mkdir -p hello_flask/{templates,static}
```

This is the basic structure of your web application:

```
$ tree hello_flask/
hello_flask/
|-- static
`-- templates
```

The `templates` folder is the place where the templates will be put. The `static` folder is the place where any files (images, css, javascript) needed by the web application will be put.

- Create the application file

```
cd hello_flask
vim hello_flask.py
```

Put the following code in this file:

```
#!/usr/bin/env python

import flask

# Create the application.
APP = flask.Flask(__name__)

@APP.route('/')
def index():
    """ Displays the index page accessible at '/'
    """
    return flask.render_template('index.html')

if __name__ == '__main__':
    APP.debug=True
```

```
APP.run()
```

- Create the template `index.html`

```
vim templates/index.html
```

Put the following code in this file

```
<!DOCTYPE html>
<html lang='en'>
<head>
  <meta charset="utf-8" />
  <title>Hello world!</title>
  <link type="text/css" rel="stylesheet"
        href="{{ url_for('static',
                          filename='hello.css')}}" />
</head>
<body>

It works!

</body>
</html>
```

- Run the flask application

```
python hello_flask.py
```

Access `http://127.0.0.1:5000/` this should simply show you in black on white the text “It works!” (see Figure below).

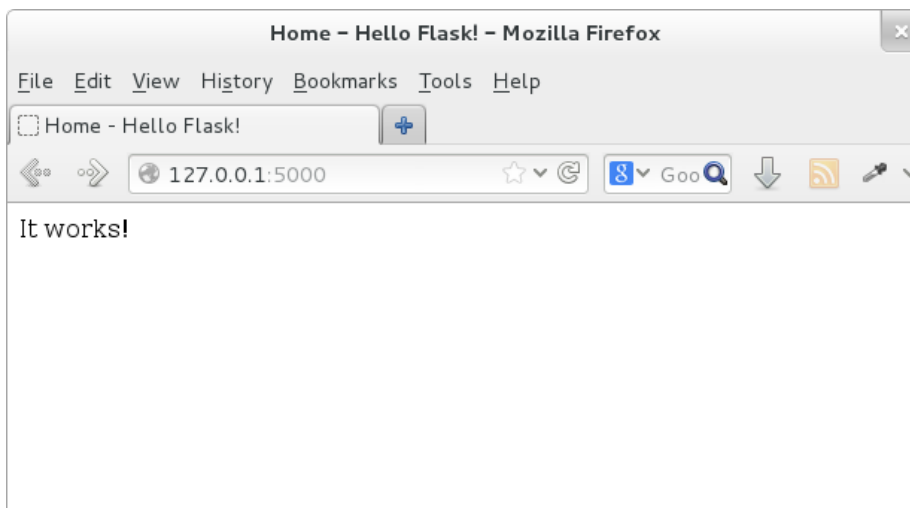


Figure 19:

Using arguments in Flask

In this section we are going to see how to use a page according to the URL used by the user.

For this we will update `hello_flask.py`.

- Add the following entry in `hello_flask.py`

```
@APP.route('/hello/<name>/')
def hello(name):
    """ Displays the page greets who ever comes to visit it.
    """
    return flask.render_template('hello.html', name=name)
```

- Create the following template `hello.html`

```
<!DOCTYPE html>
<html lang='en'>
<head>
    <meta charset="utf-8" />
    <title>Hello</title>
    <link type="text/css" rel="stylesheet"
        href="{{ url_for('static',
            filename='hello.css')}}" />
</head>
<body>

    Hello {{name}}

</body>
</html>
```

- Run the flask application

```
python hello_flask.py
```

Access `http://127.0.0.1:5000/` this should simply show you in black on white the text “It works!”.

Access `http://127.0.0.1:5000/hello/you` this should return you the text “Hello you” (see Figure below).

Whatever you put behind `/hello/` in the URL will be returned to you in the page.

This is your first use of the template, we set up a variable `name` in `hello_flask.py` (see the return line of the function `hello`). This variable is then displayed in the page itself using the syntax `{{name}}`.

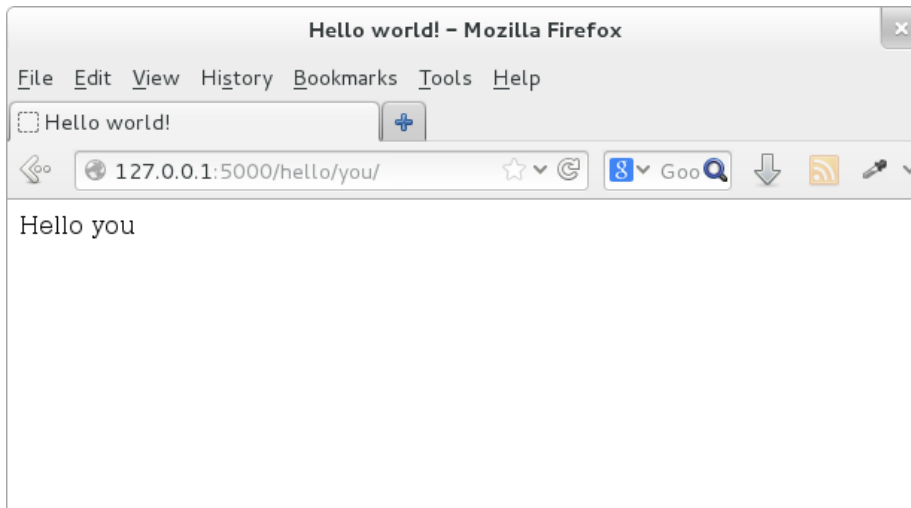


Figure 20:

Additional work

Make use of the templates

At the moment for each page we have created a template, this is actually bad practice, what we should do is create a **master** template and have each page use it.

- Create the template `master.html`

```
<!DOCTYPE html>
<html lang='en'>
<head>
  <meta charset="utf-8" />
  <title>{% block title %}{% endblock %} - Hello Flask!</title>
  <link type="text/css" rel="stylesheet"
        href="{{ url_for('static',
                          filename='hello.css')}}" />
</head>
<body>

{% block body %}{% endblock %}

</body>
</html>
```

- Adjust the template `index.html`

```
{% extends "master.html" %}
```

```
{% block title %}Home{% endblock %}

{% block body %}
It works!
{% endblock %}
```

As you can see, in the `master.html` template we have defined two sections, blocks which are named `title` and `body`.

In the template `index.html` we say that this template relies on the template `master.html`, then we define the content to put in these two sections (blocks). In the first block `title` we say to put the word Home, In the second block we define what we want to have in the body of the page.

- As an exercise, transform the other template `hello.html` to use the `master.html` template as well.
- Add link to the front page from the hello page

Flask uses a specific syntax to create links from a page to another. This is fact generates the link dynamically according to the decorator set to the function linked to. In addition it takes care of where the application is deployed.

For example, if you website is deployed at: `/myapp/` flask will automatically happend `/myapp/` to all links without the need for you to specify it.

To create a link in a template, flask relies on the function `url_for()`. This function takes as first argument the function you want to call (link to). The following arguments are the arguments of function itself (for example the argument `name` of the function `hello`).

Adjust the template `hello.html` to add a link to the front page

```
<a href="{{ url_for('index') }}"><button>Home</button></a>
```

- As an assignment add a link in the front page to the hello page for you.

Index

[_](#) | [A](#) | [C](#) | [D](#) | [E](#) | [F](#) | [I](#) | [L](#) | [N](#) | [P](#) | [R](#) | [S](#) | [T](#)

[_](#)

-
- [__init__](#)
-

A

-
- [append](#)
-

C

-
- [Class](#)
 - [count](#)
 - [Counter](#)
-

D

-
- [defaultdict](#)
 - [Dictionary](#)
 - [dir](#)
-

E

-
- [enumerate](#)
 - [Exception](#)
 - [extend](#)
-

F

-
- finally
-

I

-
- Inheritance
 - items
-

L

-
- List
 - List comprehension
-

N

-
- namedtuple
-

P

-
- Property
-

R

-
- remove
 - reverse
-

S

-
- Set
 - sort
-

T

-
- Tuple
 - TypeError
-