

Python From Scratch



Python Mauritius UserGroup

More info: mscc.mu/python-mauritius-usergroup-pymug/

Follow Us:

github.com/pymug

twitter.com/pymugdotcom

linkedin.com/company/pymug/

pymug.com

[[mailing list: pymug.python.org](https://pymug.python.org)]

Python From Scratch


- optimised for hands-on sessions
- suitable for trainings

Abdur-Rahmaan Janhangeer
(Just a Python programmer)

twitter: [@osdotsystem](#)

github: <https://github.com/abdur-rahmaanj>

www.pythonmembers.club

 *Informations concerning the setting up of an interactive prompt (REPL) has not been included*

Why Repls Are Useful

Shells are used to test out commands. You get the evaluated result without compiling, without typing and run instructions. The enter key is enough

Python version?

Python 3.7 (3.8 is already out)

Numbers and (Numerical) Operators ✂

Python understands numerical operators, just as you would guess

```
>>> 1 + 1
2
>>> 5 - 4
1
```

but also multiplication `*` and division `/`.

```
>>> 2 * 5
10
>>> 10 / 5
2.0
```

Python also has the double divide symbol `//` (called floor division)

```
>>> 5 / 2 # normal division
2.5
>>> 5 // 2
2
```

5/2 gives 2.5 which is between 2 and 3. `//` gives the lower value

`**` is the symbol for exponentiation (to the power of)

```
>>> 3 ** 2 # same as 3 x 3
9
```

`%` called modulo gives the remainder

```
>>> 9 % 5
4
```

Why are there spaces between operators?

- `2 + 2` as opposed to `2+2`

It's because Python's style guide (PEP8) proposes this

Integers and floats

1 is called an integer

1.0 is called a float

Text

Also called string. Why? in other languages (like C++), a piece of text is literally characters added together. Text was actually a string of characters.

strings are enclosed between quotes

```
>>> 'a'  
>>> 'abcd'
```

double quotes also no problem

```
>>> "abcd"
```

You can use one inside the other

```
>>> "i didn't want"  
"i didn't want"
```

as this `'i didn't want'` produces an error

you could also use " inside '

```
>>> '""''
```

if you still want to use ' inside ', escape it with \

```
>>> 'i didn't' # error
```

this one works

```
>>> 'i did\'t'
```

Which is better, " or '?

' as if you type `>>> "abcd"` it gives back `'abcd'`

Just an indication that when Python shows internal messages, it prefers '

Adding Strings Together

You can use `+` to add strings together

```
>>> 'get' + '-' + 'together'  
'get-together'
```

Bonus 📁

Strings can be added without +

```
>>> 'a' 'b'  
'ab'
```

Strings can be multiplied

```
>>> 'z' * 5  
'zzzzz'
```

Coverting 

String to integers

```
>>> 1 + '1' # error
>>> 1 + int('1')
2
```

Python needs similar data types to work with. You can convert to string via `str()`

```
>>> 'a' + str('1')
a1
```

Printing

or showing to the screen

or showing the text where you want (by default it's the console)

You can print one thing at a time

```
>>> print(1)  
1
```

You can print many things

```
>>> print(1, 2, 3)  
1 2 3
```

And you can print different things

```
>>> print(1, 2.0, 'a')  
1 2.0 a
```


Bonus 📁

You can say what to separate the elements by using `sep=' '`.

```
>>> print(1, 2, 3, sep='â€') # Default is sep=' '  
1â€2â€3
```

You can say how to end the printing by using `end=' '`.

```
>>> print(1, 2, 3, end='@') # # Default is end='\n'  
1 2 3@
```

Print also takes `file=` and `flush=` parameters. Try

```
print(1, 2, 3, 4, file=open('myfile.txt', 'w+'))
```

Inspection

Python has some cool functions for inspecting objects

help()

`help()` is used to provide more info about something. try

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

type()

type gives the type of the value

```
>>> type(2.0)  
<class 'float'>
```

dir()

dir is used to find associated modules and info. try

```
>>> dir('abc')
```

What more?

What can we use more / what's included right of the bat?

What more?

The builtins module provides info about what's included by default

```
>>> import builtins  
>>> dir(builtins)
```

String functions

You can do a lot of manipulations with strings

```
>>> dir('abc')
[__ **not interested** __,
'__len__', ...,
'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isascii',
'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```


Here are some interesting functions

```
>>> len('abc')
3
>>> 'abc'.replace('a', 'z')
'zbc'
>>> 'abc'.upper()
'ABC'
>>> '1'.isdigit()
True
>>> '__abcd__'.strip('_')
'abcd'
>>> '^'.join('abcd')
'a^b^c^d'
>>> 'abcdefabcdef'.count('a')
2
```

Indexing

You can get specific elements of a string

```
>>> 'abcd'[0]
'a'
>>> 'abcd'[1]
'b'
>>> 'abcdefgh'[2:4]
'cd'
>>> 'abcdefgh'[0:4:2] # [start:stop:step]
'ac'
>>> 'abcdefgh'[0::2]
'aceg'
>>> 'abcdefgh'[::-2]
'aceg'
>>> 'abcdefgh'[::-1] # reverses the string
'hgfedcba'
```

String Formatting 🌽

normally if you want to add string and int, you have to convert to string

```
for i in range(5):  
    print('now passing over: ' + str(i))
```

but .format makes it easier

```
for i in range(5):  
    print('now passing over: {}'.format(i))
```

even easier with f strings (py3.7)

```
for i in range(5):  
    print(f'now passing over: {i}')
```

Variables

You can assign values to names using the `=` symbol.

```
>>> x = 1
>>> x = 'abcd'
>>> x[0]
'a'
>>> x = 'abcd'.upper()
>>> x
'ABCD'
>>> x = 2 + 2
>>> x
4
>>> y = 1
>>> x + y
5
```

more operators

You can use the `+=` operator

```
x = 1
x = x + 1
print(x) # 2
```

same as

```
x = 1
x += 1
print(x) # 2
```

we also have `-=`, `*=`, `/=`, `%=`

Bools

Booleans can be True or False. You can test by `bool()` . `bool('a')` .

Conditionals ?

We can compare values.

```
if 1 == 1:  
    print('1 equals one')
```

but it's more useful to compare variables

```
x = 1  
if x == 1:  
    print('x equals one')
```

But you also have more operators

```
x = 2  
if x > 1:  
    print('x greater than 1')
```


you also have

Operator	Description
>	greater than
<	less than
==	equal to
!=	not equal to
>=	greater equal to
<=	less equal to

but if we have more conditions, we use elif

```
x = 2
if x == 1:
    print('x equals one')
elif x == 2:
    print('x equals two')
```

we can add an else to catch whatever remains

```
x = 2
if x == 1:
    print('x equals one')
elif x == 2:
    print('x equals two')
else:
    print('x is neither one or two')
```

And and Or

Here is an example that uses and and or

```
x = 5
y = 6

if x == 5 and y == 6:
    print('ok') # prints

if x == 5 or y == 6:
    print('ok') # prints

if x == 5 or y == 3:
    print('ok') # prints
```

More Data Types:

lists and dictionaries

You have lists for storing many items

```
fruits = ['apple', 'orange', 'pear']
```

to access the elements, we do:

```
fruits[0] # apple  
fruits[1] # orange  
fruits[2] # pear
```

indexing works same as string

```
>>> fruits[::-1]  
['pear', 'orange', 'apple']
```

lists can also hold values of different types

```
[1, 'a']
```

more operations:

```
>>> fruits.append('mango')
>>> fruits
['apple', 'orange', 'pear', 'mango']
>>> [2] + [2]
[2, 2]
>>> x = [[1, 2], [3, 4], 1] # list within list
>>> x[0]
[1, 2]
```

You have dictionaries for storing value by pairs

```
x = {  
    0: 'zero',  
    1: 'one'  
}
```

Then to access elements we do:

```
x[0] # 'zero'  
x[1] # 'one'
```

Loops

Loops are used to cycle over values

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

...

0

1

2

3

4

...

we can add a starting point

```
for i in range(2, 5):  
    print(i)
```

...

2

3

4

...

we can skip

```
for i in range(0, 10, 2): # start, end, step
    print(i)
'''
0
2
4
6
8
'''
```

to go over lists we do

```
fruits = ['apple', 'orange', 'pear']
for f in fruits:
    print(f)
```

:

```
'apple'
'orange'
'pear'
```

You also have while loops:

```
x = True

while x == True: # while True or while 1
    print('ok')
```

since x is always true, it will print ok infinitely. we can add conditions to change x to false or we break upon certain conditions

```
ongoing = True
x = 0 # counter
while ongoing:
    print(x)
    if x == 6:
        ongoing = False
    x = x + 1
# prints 0 ... 6
```

Bonus 📁

In python, True is equal to one (Try `True + 1` to verify). That's why in some scripts you'll find

```
while 1:  
    ...
```

which is the same as `while True:`

list comprehensions use the concept of loop

```
x = [i for i in range(5)]
```

is a shorthand of getting

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

Breaking Out of Loops

```
for i in range(0, 10, 2):  
    if i == 6:  
        break  
    print(i)
```

it will print 0 2 4 and will stop at 6

```
for i in range(0, 10, 2):  
    if i == 6:  
        continue  
    print(i)
```

continue will skip that pass / what comes after it

it will print 0 2 4 8

Imports

try those:

```
import this
```

and

```
import antigravity
```

those are fun stuffs lying around, but, python has many useful imports


```
import random
```

```
x = random.random()  
print(x)
```

random integer

```
import random
```

```
x = random.randint(4, 10)  
print(x)
```

you can shuffle a list

```
random.shuffle(fruits)
```

you can also randomly choose an element

```
import random

fruits = ['apple', 'orange', 'pear']
x = random.choice(fruits)
print(x)
```

you could also write

```
from random import shuffle

fruits = ['apple', 'orange', 'pear']
x = choice(fruits)
print(x)
```

or

```
from random import *

fruits = ['apple', 'orange', 'pear']
x = choice(fruits)
print(x)
```

but it's not recommended as it can collide with your own shuffle functions or for efficiency reasons as your program will load everything

Functions

Functions were intended to be a way not to repeat your code

Let's say you have

```
print(1)  
print(2)  
print(3)  
print(4)
```

if you want to print those 4 times, you'll do:

```
print(1)
print(2)
print(3)
print(4)
print(1)
print(2)
print(3)
print(4)
print(1)
print(2)
print(3)
print(4)
```

which is tedious

but putting those in a function

```
def print_nums():
    print(1)
    print(2)
    print(3)
    print(4)
```

then calling it four times

```
print_nums()  
print_nums()  
print_nums()  
print_nums()
```

is much easier

You can also return values

```
def return_3():  
    return 3  
  
x = return_3() # x now equals 3  
print(return_3()) # same as printing 3
```

You can also specify what to print in variables called parameters (here x)

```
def print_thrice(x)  
    print(x)  
    print(x)  
    print(x)  
  
print_thrice(2)  
# 2  
# 2  
# 2
```


you can return a value

```
def area_of_rect(width, height):  
    return width * height  
  
area = area_of_rect(10, 5) # now equals 50
```

You can also specify by name (keyword arguments)

```
def print_fruit(fruit='apple'):  
    print(fruit)  
  
print_fruit() # apple  
print_fruit(fruit='orange') # orange
```

Classes

Classes

classes are a way to hold variables and functions together under a common name

```
class Car:
    def __init__(self): # ignore that word 'self'
        self.wheels = 4
        self.make = 'mazda'

    def move(self):
        print('moving ...')

    def change_make(self, mymake):
        self.make = mymake
```

self holds the variables and functions to Car

you use it like

```
car1 = Car() # assigning to a class is called instantiation
print(car1.wheels) # 4
print(car1.make) # 'mazda'
car1.move() # 'moving ...'
car1.change_make('toyota')
print(car1.make) # 'toyota'
```

Exceptions

if you try:

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

you'll get

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

notice: we have a **TypeError**

we can catch the error by

```
try:  
    x = 1 + '1'  
except TypeError:  
    print('please add integers with integers')
```

we can also write

```
try:  
    x = 1 + '1'  
except:  
    print('please add integers with integers')
```

but specifying the error we want to catch gives a better idea of what went wrong

you can also print the error

```
try:  
    x = 1 + '1'  
except Exception as e:  
    print('exception' + str(e) + 'occured')
```

or let go

```
try:  
    x = 1 + '1'  
except:  
    pass
```

Can we use exceptions to control program flow in Python?

It is normal in Python to use exceptions to control flow as well as catching exceptions. It is not an anti-pattern as in other languages

Fun Corner 🍭

Files

To read file, you do

```
f = open('file.txt', 'r')  
print(f.read())
```

To write to a file you do:

```
f = open('file.txt', 'w+')  
f.write('pymug')  
f.flush()  
f.close()
```

but this syntax is also allowed:

```
with open('file.txt', 'r') as f:  
    print(f.read())
```

and writing

```
with open('file.txt', 'w+') as f:  
    f.write('abc')
```

no need to flush and close

Bonus

When reading files you can omit the 'r' as files are opened by default in the read mode

```
f = open('file.txt')  
print(f.read())
```

Commands

you can pass command-line strings in `os.system`

```
import os  
os.system('commands here')
```

⚠ not safe at all, allows RCE

Create folder

```
import os  
os.mkdir('folder path here') # 'testing'
```



🌸 Please Subscribe To Always Catch Updates: [\[Click here\]](#)