Complete Python Boot camp — Go from zero to hero in Python 3

David Corzo

2020 May 18

Contents

Chapter 1

Python Object and Data Structure Basics

1.1 What beginners don't know

- Command line basics:
  - Windows, macOS, Linux
- Installing python and making it a path environment variable.
- Difference between distributions.
- How to google search for answering one's own questions.
- How to run python code.
- Difference between an IDE and python.
- What is syntax highlighting.
- Variables and their purpose.

1.2 Basic data types

Name	Type
Integers	int
Floating point	float
Strings	str
Lists	list
Dictionaries	dict
Tuples	tup
Sets	set
Booleans	bool

1.3 Numbers

Modulo	20 % 2	$20 \times \pmod 2$
Powers	2 ** 2	2^2
Floor	10 // 3	$\left\lfloor \frac{10}{3} \right\rfloor$
Grouping	(10 + 3) / 3	$\frac{10+3}{3}$

1.4 Variable names

Rules:

- No spaces
- Can't use these: `;"',<>\()!@#$%^&*~-+|`
- Don't use keywords.

1.4.1 Dynamic typing

- Python is a dynamically typed language.
- This means you can reassign variables to different data types.
- This makes Python very flexible in assigning data types, this is different to other languages that are "Statically-Typed".

Pros	Cons
• Easy to work with • Faster development process	• May result in bugs for unexpected data types. • You need to be aware of `type()` function to check this.

1.5 Strings

- In Python ' and " are the same in strings.
- String indexing: used to extract a single character in a string. a[index]
- String slicing: grabs an interval of indexes. a[start:stop:step]
  - start: numerical index for the slice start.
  - stop index you will go up to but not include.

  - step: the size of the jump you take.

- Special characters: `\n`, `\t`, `\d`
- the lenght function: len(a) -> how many elements are in the string.

1.5.1 Normal and reverse indexes

H	E	L	L	O		W	O	R	L	D	!
0	1	2	3	4	5	6	7	8	9	10	11
0	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

1.5.2 Examples

```
# STRING INDEXES AND REVERSE INDEXES
a = "123456"
a[1:2]
# output: "2" -> Index 1 up to but not including 2
a[1:3]
# output: "23" -> Index 1 up to but not including 3
a[:3]
# output: "123" -> Index 0 (default) up to but not including 3
a[::]
# output: "123456" -> Normal string
```

1.6 String properties

- Strings are inmutable:

```
name = "david"
name[0] = "b"
# OUTPUT:
# Traceback (most recent call last):
#   File "<stdin>" line 1, in <module>
# TypeError: 'str' object does not support item assignment
```

- You can concactenate using the + and * signs

```
2 + 3
# output: 5
"2" + "3"
# output: "23"
"h" * 10
# output: "hhhhhhhhhh"
```

- Methods:
  - .upper()
  - .lower()
  - .replace()
  - .split()

1.7 Print formating with strings

- .format() for string interpolation:

```
print("this is a string {}".format("inserted"))
# output: This is a string inserted

print("The {} {} {}".format("fox","brown","quick"))
# output:  The fox brown quick

print("The {2} {1} {0}".format("fox","brown","quick"))
# output: The quick brown fox

print("The {a} {b} {c}".format(a="fox",b="brown",c="quick"))
# output: The quick brown fox

result = 100/777
print(result)
# output: 0.1287001287001287
# Float formatting formula "{value:width.precision f}"
print("The result was {r:1.3f}".format(r=result))
# output: The result was 0.129
```

- f-strings for string interpolation:

```
# F-STRING FORMATING
var = "World"
print(f"Hello {var}")
# output: Hello World
```

1.8 Lists

- They suport indexing and slicing.

```
my_list = [1,2,3]
my_list[1:]
# output: [2,3]
another_list = [4,5]
my_list + another_list
# output: [1,2,3,4,5]
```

- You can also concactenate lists together.
- Lists are mutable.

1.8.1 Methods

- .append(⟨something⟩) -> Appends to the list.
- .sort() -> sorts the iterable alphabetically or numericaly, this is a void, it doen't return anything, don't assign it to anything because it returns None.

```
new_list = ['a','e','x','b','c']
new_list.sort()
new_list
# output: ['a','b','c','e','x']
```

- .pop() → eliminates last element and prints it.

```
my_list.pop(0)
# output: pops element 0.
```

- .reverse() → returns the reverse of the list.

1.9 Dictionaries

- Doesn't need indexes. Just keys and values.
- Dictionaries are unordered and can't be sorted.
- Methods:

```
d = {"key1":1,"key2":2}
```

1.9.1 Methods

- d.keys() → returns all the keys in a dictionary.

```
d.keys()
# output:  dict_keys(['key1','key2'])
```

- d.values() → returns all the values in a dictionary.

```
d.values()
# output: dict_values([1,2])
```

1.10 Tuples

- They are immutable lists essentially.

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

- Useful for immutable applications and memory efficiency.

1.10.1 Methods

- t.count('a') → counts the occurrences of 'a' in the tuple.
- t.index('a') → returns the index in which the first time 'a' appears.

1.11 Sets

- Unordered collection of unique elements.

```
my_set = set()
```

- There can only be one of each element, no duplicates.
- Useful for duplicate deletion.

1.11.1 Methods

- .add() → inserts 1 to the set.

```
my_set.add(1) # output: {1}
my_set.add(2) # output: {1,2}
my_set.add(2) # output: {1,2} -> doesn't add because that is a duplicate.
```

1.12 Boolans

- Allows only True, False.

1.13 Files

- Opening syntax:

```
with open('myfile.txt') as f:
    contents = f.read()
# or
contents = f.read('myfile.txt')
```

1.13.1 Methods

- .read() → returns a string with all the contents of the file.
- .readlines() → returns a list with the lines in a file.
- .seek(⟨index⟩) → sets the cursor to the specified index (usually 0).
- .truncate() → deletes all information in a file.
- .close() → closes the file, it's a best practice.

1.13.2 Permissions on the with open('myfile.txt') statement

- `mode='r'` → permissions are enabled to read a file.
- `mode='w'` → permissions are enabled to write and overwrite to a file, or create a new file.
- `mode='r+'` → reading and writing permissions.
- `mode='w+'` → writing and reading, overwriting or creating new files.
- `mode='a'` → appends data to the end of a file, overwriting and writing to a file permissions are enabled.

Chapter 2

Python comparison operators

2.1 Comparison operators

a == b	a is b
a != b	a is not b
a > b	a is greater than b
a < b	a is less than b
a >= b	a is grater or equals b
a <= b	a is less than or equal to b

- In python, 2.0 == 2 is True.

2.2 Chaining comparison operators in Python with logical operators

2.2.1 Logical operators

- and
- or
- not

```
1 < 2 < 3
# output: True -> Chaining
1 < 2 and 2 < 3
# output: True -> Comparison operators you can also use parenthesis.
(1 < 2) and (2 < 3)
# output: True -> with parenthesis comparison operators.
not(1 == 1)
# output: False -> returns the oposite.
not 1 == 1
# output: False -> the same but with no parenthesis.
```

Chapter 3

Python statements

3.1 Control flow

```
if (True): # output: it this isn't true you excecute the elif or else, these don't have a limit
    pass
elif (True): # output: elif statements don't have a limit. Exclusive
    pass
else: # output: These have a limit, only one.
    pass
```

3.2 For loops

- Iterable: something that is iterable.

```
mylist = [(1,2),(3,4),(5,6),(7,8)]
for (a,b) in mylist: # output: This is variable unpacking, this can be done without the parenth
    print(a)
    print(b)
mylist = [(1,2,3),(4,5,6),(7,8,9)]
for a,b,c in mylist: # output: tuple unpacking of three variables.
    print(a,b,c)
```

3.3 While loops

- pass → do nothing at all. Useful to for placeholders.

  ```
  while True:
      pass
  else: # output: combine the else with the while.
      pass
  ```

- continue → used to omit an action, goes to the top of the closest enclosing loop.

  ```
  for item in x:
      pass
  ```

- break → breaks out of the closest enclosing loops.

```
    mystring = "Sammy"
    for letter in mystring:
        if letter == "a":  # this letter will be ommited.
            continue
        print(letter)
    # output: S
    # output: m
    # output: m
    # output: y

x = 5
while x < 5:
    if x == 2: # whenever x = 2 the loop will stop.
        break
    print(x)
    x += 1
```

3.4 Useful operators

- range(start, stop, step)
  - start: start index
  - stop: all the way up but not including the stop.
  - step: the step size
- enumerate(iterable) → keeps index count automatically.

```
index = 0
word = 'abcde'
for item in word:
    print(index,item)
    index += 1
# (0,'a')
# (1,'b')
# (2,'c')
# (3,'d')
# (4,'e')
# The same can be achived with enumerate
for item in enumerate(word):
    print(item)
# (0,'a')
# (1,'b')
# (2,'c')
# (3,'d')
# (4,'e')
```

- zip(iterable1, iterable2, iterablen) → zips together iterables.

```
mylist1 = [1,2,3]
mylist2 = ['a','b','c']
```

```
mylist3 = [100,200,300]
for item in zip(mylist1,mylist2,mylist3):
    print(item)
# output: (1,'a',100)
# output: (2,'b',200)
# output: (3,'c',300)
```

- in operator → returns a boolean value:

  ```
  'x' in ['x','y']
  # output: True
  'mykey' in {'mykey':345} # works for the keys
  # output: True
  ```

- min & max → returns minimum of maximum of an iterable.

  ```
  mylist = [1,2,3]
  min(mylist)
  # output: 1
  max(mylist)
  # output: 3
  mylist = "abcd"
  min(mylist)
  # output: a
  max(mylist)
  # output: d
  ```

- random library functions:
  - shuffle → shuffles elements of a list, this is a void function.

    ```
    from random import shuffle
    mylist = [1,2,3,4,5,6,7,8,9,10]
    shuffle(mylist)
    # output: [3, 9, 5, 1, 2, 10, 8, 6, 7, 4] -> it shuffles it
    ```

  - randint → selects random int in range.

    ```
    from random import randint
    mynum = randint(0,10)
    # output: 3 -> random number in range 0 to 9
    ```

- input function:
  - Input always returns a string.

```
result = input("Enter a number: ")
# output: Enter a number: -> you will be required to enter input.
```

3.5 Lists comprehensions

- This is used for generating lists.
- Syntax is: ⟨what will be appended⟩ for ⟨element⟩ in ⟨iterable⟩

```
mylist = [letter for letter in 'string']
# output: ['s', 't', 'r', 'i', 'n', 'g']

mylist = [num for num in range(0,11)]
# grab the square of every number in that list.
mylist = [num**2 for num in range(0,11)]
# output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- You can also add conditional statements.
- Syntax is: [⟨what will be appended⟩ for ⟨element⟩ in ⟨iterable⟩ if (⟨condition⟩)]

```
mylist = [x for x in range(0,11) if x % 2 == 0]
# output: [0, 2, 4, 6, 8, 10]
```

3.5.1 Example

Celsius to Fahrenheit:

```
fahrenheit = []
celsius = [0,10,20,34.5]
for temp in celsius:
    fahrenheit.append(((9/5)*temp + 32))
# output: [32.0, 50.0, 68.0, 94.1]

# Alternatively on a one liner
fahrenheit = [((9/5)*temp + 32) for temp in celsius]
# output: [32.0, 50.0, 68.0, 94.1]
```

3.5.2 List comprehension using if and else

```
results = [x if x % 2 == 0 else 'ODD' for x in range(0,11)]
# output: [0, 'ODD', 2, 'ODD', 4, 'ODD', 6, 'ODD', 8, 'ODD', 10]
```

- The syntax is: [⟨element⟩ if ⟨condition is wanted⟩ else ⟨if condition isn't wanted return⟩ ⟨what to return in for ⟨element⟩ in ⟨iterable⟩]

3.5.3 Nested loops in list comprehension

Syntax is: [⟨what will be appended⟩ for ⟨element⟩ in ⟨iterable⟩ for ⟨second element⟩ in ⟨second iterable⟩]

```
mylist = []
for x in [2,4,6]:
    for y in [1,10,1_000]:
        mylist.append(x*y)
# output: [2, 20, 2000, 4, 40, 4000, 6, 60, 6000]

# Alternative one liner
myresult = [x*y for x in [2,4,6] for y in [1,10,1_000]]
# output: [2, 20, 2000, 4, 40, 4000, 6, 60, 6000]
```

Chapter 4

Methods and functions

4.1 Functions on python

Syntax is:

```
def name_of_function(args):
    """ Doc strings """
    pass
```

4.2 *args and **kwargs

Example of *args:

```
def myfunc(*args):
    return sum(args) * 0.05
myfunc(1,2,3,4,5,6,7,8,9,10)
# output: 2.75
```

Example of **kwargs:

```
def myfunckwarg(**kwargs):
    if 'fruit' in kwargs:
        print(f"My fruit of chice is {kwargs['fruit']}")
    else:
        print('I didn\'t find anything')
myfunc(fruit='watermelon')
# output: My fruit of chice is watermelon
```

Using **kwargs like this creates a dictionary:

```
myfunc(fruit='apple',veggie='lettuce')
# output: {'fruit':'apple','veggie':'lettuce'}
```

Use them in combination:

```
def myfunc(*args, **kwargs):
    print('I would like {} {}'.format(args[0],kwargs['food']))
myfunc(1,2,3,4,5,6,7,8,9,'Hello','None')
# output: I would like 1 None
```

4.3 Lambda expressions Map and Filter

- map(func, iterable): aplies a function to an iterable.

  ```
  def square(num):
      return num**2
  my_nums = [1,2,3,4,5]
  for item in map(square,my_nums):
      print(item)
  # output: 1
  # output: 4
  # output: 9
  # output: 16
  # output: 25
  list(map(square,my_nums))
  # output: [1, 4, 9, 16, 25]
  ```

- filter(func,iterable): keeps element if given function returns true.

  ```
  def check_even(num):
      return num%2 == 0
  mynums = [1,2,3,4,5,6]
  list(filter(check_even,mynums))
  # output: [2, 4, 6]
  for n in filter(check_even,mynums):
      print(n)
  # output: 2
  # output: 4
  # output: 6
  ```

- lambda: are one use functions.

  - Syntax is: lambda ⟨function arguments⟩ : ⟨what will be returned⟩

  ```
  square = lambda num: num ** 2
  square(5)
  # output: 25
  list(map(lambda num: num**2,mynums))
  # output: [1, 4, 9, 16, 25, 36]
  list(map(lambda num: num%2 == 0,mynums))
  # output: [2, 4, 6]
  names = ['Hello','World','!']
  list(map(lambda string:string[0],names))
  # output: ['H', 'W', '!']
  ```

4.4 Nested statements and scope

4.4.1 LEGB Rule

- L: Local - all names assigned inside a function (def or lambda).

```
lambda num: num**2 # num is local
```

- E: Enclosing function locals - all names assigned to functions inside functions. (def or lambda), from inner to outer.

```
name = "Global string"
def greet():
    name = 'Sammy'
    def hello():
        print('Hello ' + name)
    hello()
greet()
# output: Hello Sammy
```

- G: Global (module) - names assigned at the top level of a modile file, or declared global in def within the file.

```
# GLOBAL
name = "Global string"
def greet():
    # ENCLOSING
    name = 'Sammy'
    def hello():
        # LOCAL
        name = 'Im a local'
        print('Hello ' + name)
    hello()
greet()
# output: Hello Im a local
```

- B: Built-in (Python) - names preassigned in the buit-in names module: open,range,SyntaxErrore,etc.

4.4.2 global keyword

- You must remove the global variable as a parameter, then you declare it as: global ⟨var name⟩

```
x = 50
def func():
    global x # go to the global space and grab the variable x
    # Anything that happens with x will be equivalent as it happening on a global scope.
    print(f"x is {x}")
    #LOCAL REASSIGNMENT ON A GLOBAL VARIABLE
    x = 'new value'
    print(f"I JUST LOCALLY CHANGED X TO {x}")
print(x)
# output: 50
func()
# output: x is 50
# output: I JUST LOCALLY CHANGED X TO new value
print(x)
# output: new value
```

- The global keyword is hard to predict, if you are working with lots of scripts you might make it more dificult to debug or you might overwrite variables in the global space unintentionally. So instead do this:

```python
# EQUIVALENT OF ALTERNATIVE
x = 50
def func(x):
    print(f"x is {x}")
    #LOCAL REASSIGNMENT ON A GLOBAL VARIABLE
    x = 'new value'
    print(f"I JUST LOCALLY CHANGED X TO {x}")
    return x
x = func(x)
# output: x is 50
# output: I JUST LOCALLY CHANGED X TO new value
print(x)
# output: new value
```

Chapter 5

Object-oriented programming

5.1 Object-oriented programming

- Methods: act as functions that use information about the object, as well as the object itself to return results, or change the current object.

5.2 Attributes and class keyword

- By convention class names are written in upper camel case convention.

```
class Sample():
    pass
my_sample = Sample()
type(my_sample)
# output: __main__Sample

class Dog():
    # This is the constructor method.
    def __init__(self,breed,name,spots):
        # By convention the attribute name and the parameter name are the same.
        # Attributes
        # We take in the argument
        # Assign it using self.attribute_name
        self.breed = breed
        self.name = name
        self.spots = spots

my_dog = Dog(breed="Lab",name="Sammy",spots=False)
type(my_dog)
# output: __main__.Dog
my_dog.breed
# output: 'Lab'
my_dog.name
# output: 'Sammy'
my_dog.spots
# output: False
```

5.3 OOP Class object attributes

- Class object attributes are defined before the constructor.
- They are the same for any instance of the class.
- Class object attributes can be referenced in two ways:
    1. ⟨Class name⟩ . ⟨class object attribute⟩
    2. self. ⟨class object attribute⟩

```
class Circle():
    # CLASS OBJECT ATTRIBUTE
    pi = 3.14

    def __init__(self, radious=1):
        self.radious = radious
        self.area = (radious ** 2) * Circle.pi

    def get_circumference(self):
        return self.radious * self.pi * 2

my_circle = Circle(30)
my_circle.pi
# output: 3.14
my_circle.radious
# output: 30
my_circle.get_circumference()
# output: 188.4
my_circle.area
# output: 2826.0
```

5.4 OOP Inheritance and polymorphism

- Inheritance: form new classes using classes that have already been defined, it's goal is to reuse code and to reduce complexity.
    - All methods from the base class can be overwritten and used in the inheritance class.

```
# BASE CLASS
class Animal():
    def __init(self):
        print("Animal created")

    def who_im_i(self):
        print("I'm an animal")

    def eat(self):
        print("I'm eating")
```

```python
# INHERITANCE OF DOG FROM THE ANIMAL CLASS
class Dog(Animal):
    def __init__(self):
        Animal.__init__(self) # Copy Animal class constructor
        print("Dog created")
    def who_im_i(self):
        print("I'm a dog!")
    def bark(self):
        print("I'm a dog eating")

mydog = Dog()
# output: Animal created
# output: Dog created
mydog.who_im_i()
# output: I'm a dog!
mydog.bark()
# output: I'm a dog eating
```

- Polymorphism: diferent object classes with identical method names.

```python
class Dog():
    def __init__(self,name):
        self.name = name
    def speak(self):
        print(self.name + " says woof!")

class Cat():
    def __init__(self,name):
        self.name = name
    def speak(self):
        print(self.name + " says meow!")

niko = Dog("niko")
felix = Cat("felix")
print(niko.speak())
# output: niko says woof!
print(felix.speak())
# output: felix says meow!

for pet in [niko,felix]:
    print(type(pet))
    print(pet.speak())
# output: <class '__main__.Dog'>
# output: niko says woof!
# output: <class '__main__.Dog'>
# output: felix says meow!
```

- An abstract class is a class which is not intended to be instanciated, it's only used as a base class.

```python
# ABSTRACT CLASSES
class Animal():
    def __init__(self,name):
        self.name = name
    def speak(self):
        raise NotImplementedError("Subclass must implement this abstract method.")
class Dog(Animal):
    def __init__(self,name):
        self.name = name
    def speak(self):
        print(self.name + " says woof!")

class Cat(Animal):
    def __init__(self,name):
        self.name = name
    def speak(self):
        print(self.name + " says meow!")

fido = Dog("Fido")
isis = Cat("Isis")
print(fido.speak())
print(isis.speak())
```

5.5 Special (Magic/Dunder) methods

- Make python built-in methods interact with the user defined objects and classes.

```
class Book():
    def __init__(self,title,author,pages):
        self.title = title
        self.author = author
        self.pages = pages

b = Book("Python","David",200)
print(b)
# output: <__main__.Book object at 0x000001B7E92102B0>
# I want to be able to print this in a string form so...

class Book():
    def __init__(self,title,author,pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return f"{self.title} by {self.author}"

    def __len__(self):
        return self.pages

    def __del__(self):
```

```python
        print(f"The book object '{self.__str__()}'' has been deleted")

b = Book("Python","David",200)
print(b)
# output: Python by David
print(len(b))
# output: 200
del b
# output: The book object 'Python by David' has been deleted
```

Chapter 6

Modules and packages

6.1 pip install and pypi

6.1.1 PyPI

- PyPI is a repository for open source third party python packages.
- Similar to NPM in node.js

6.2 pip

- Used to Python standard libraries of Python.

```
C:\Users\DAVIDCORZO>pip install requests
```

6.3 Modules and packages

6.3.1 Modules

- Modules: importing .py scripts into others.

6.3.2 Packages

Follow a directory structure:

```
C:SomeDir\mypackage
# Create file inside "mypackage" dir
C:SomeDir\mypackage\__init__.py
# Add modules inside this directory
```

In python import it as:

```
from mypackage import some_module
```

Subpackages:

```
# Add subpackage inside mypackage
C:SomeDir\mypackage\mysubpackage
# Add the __init__.py
C:SomeDir\mypackage\mysubpackage\__init__.py
```

In python do:

```
from mypackage.mysubpackage import some_module
```

import specific functions inside a subpackage or a package

6.4 The __name__ == "__main__" statement

- Used to distinguish between modules and main files.
- In Python all the code in the first level gets runned, the built-in variable `__name__` is also always equals to `__main__`.
- This is why the main function is runned in this famous if statement.

```
def main():
    pass
if __name__ == "__main__": # This is always going yo be the case.
    main()
else:
    print("Not excecuted")
```

Chapter 7

Errors and exception handling

7.1 Errors and exception handling

- try: code that will be attempted.
- except: in case of error this will be executed.
- finally: this will always be executed regardless of the error.

```
def add(n1,n2):
    return n1 + n2

add(1,input("Num2:"))
# output: error

try:
    add(1,input("Num2:"))
except:
    print("Error")
else:
    print("excecute if except is not excecuted")
finally:
    print("Done")
```

7.2 Pylint overview

- Unit testing: test your code, makes sure your code still works.
- pylint & unittest
- PEP-8 unit testing convention.

pylint: run on command line:

```
pylint myfile.py
```

7.3 unittest: run on command line:

in the cap.py file:

```python
def cap_text(text):
    return text.title()
```

on test.py:

```python
import unittest
import cap
class TestCap(unittest.TestCase):
    def test_one_word(self):
        text = "python"
        result = cap.cap_text(text)
        self.assertEqual(result,"Python")
    def test_multiple_words(self):
        text = "monty python"
        result = cap.cap_text(text)
        self.assertEqual(result,"Monty Python")
if __name__ == "__main__":
    unittest.main()
```

Chapter 8

Python decorators

8.1 Decorators in Python

- It's an on/off switch that lets a function activate extra functionality meanwhile maintaining the original (w/o the new functionality).
- It consists of an anidated function which wraps the parameter function with a before and after code.
- They are used in web frameworks such as Django or Flask.
- In the below examples there are two ways to decorate a function one with the @ operator and the other without the @ operator.

8.1.1 Example without the @ operator

```
def new_decorator(original_func):
    def wraper_func():
        print("Some code actions before your original function")
        original_func()
        print("Some code actions after your original function")
    return wraper_func
def func_deco():
    print("This is the code intended to be decorated")
decorated_func = new_decorator(func_deco)
decorated_func()
# output: Some code actions before your original function
# output: This is the code intended to be decorated
# output: Some code actions after your original function
```

8.1.2 Example with the @ operator

```
# Using a decorator syntax
@new_decorator
def func_deco():
    print("This is the code intended to be decorated")
func_deco()
# output: Some code actions before your original function
# output: This is the code intended to be decorated
# output: Some code actions after your original function
```

Chapter 9

Generators with Python

9.1 Generators

- Allows algorithms to only do what is necessary at a time.
- They don't actually return a value and then exit, it will complete the task periodically.
- It's more memory efficient.

```python
def create_cubes(n):
    """
    This function will create an entire list in memory before doing anything else.
    """
    result = []
    for x in range(n):
        result.append(x**3)
    return result

for x in create_cubes(10):
    print(x)
# output: 0
# output: 1
# output: 8
# output: 27
# output: 64
# output: 125
# output: 216
# output: 343
# output: 512
# output: 729

def create_cubes(n):
    """
    This function will produce the cubes according to when they are needed.
    """
    for x in range(n):
        yield x**3
for i in create_cubes(10):
    print(x)
```

```
# output: 0
# output: 1
# output: 8
# output: 27
# output: 64
# output: 125
# output: 216
# output: 343
# output: 512
# output: 729

def gen_fibon(n):
    """ Memory ineficient fib sequence """
    a,b = 1,1
    output = []
    for i in range(n):
        output.append(a)
        a,b = b,a+b
    return output
for num in gen_fibon(10):
    print(num)
# output: 1
# output: 1
# output: 2
# output: 3
# output: 5
# output: 8
# output: 13
# output: 21
# output: 34
# output: 55

def gen_fibon(n):
    """ Memory eficient fib sequence """
    a,b = 1,1
    for i in range(n):
        yield a
        a,b = b,a+b
for num in gen_fibon(10):
    print(num)
# output: 1
# output: 1
# output: 2
# output: 3
# output: 5
# output: 8
# output: 13
# output: 21
# output: 34
# output: 55
```

- The range function is a generator.
- next(⟨iterable⟩): returns the next iteration of the generator object.

```
def simple_gen():
    for x in range(3):
        yield x
for num in simple_gen():
    print(num)
# output: 0
# output: 1
# output: 2

g = simple_gen()
g
# output: <generator object simple_gen at 0x000001BB5D22D840>

print(next(g))
# output: 0
print(next(g))
# output: 1
print(next(g))
# output: 2
print(next(g))
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# StopIteration

s = "hello"
for letter in s:
    print(letter)
next(s)
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# TypeError: 'str' object is not an iterator
```

- iter(⟨iterable⟩): is a function that turns an iterable object in to a generator.

```
s_iter = iter(s)
next(s_iter)
# output: 'h'
next(s_iter)
# output: 'e'
next(s_iter)
# output: 'l'
```

9.1.1 Generator comprehension

You can use comprehension to make generators just like list comprehension just with the diferent syntax, just change the `[]` to `()`.

```python
square = (i**2 for i in range(1,11))
print(square)
# output: <generator object <genexpr> at 0x0000014A1427D840>
for x in square:
    print(x)
# output: 1
# output: 4
# output: 9
# output: 16
# output: 25
# output: 36
# output: 49
# output: 64
# output: 81
# output: 100
```

Chapter 10

Advanced Python Modules

10.1 Advanced Python modules

10.1.1 Collections module

Count occurances:

```
from collections import Counter
l = [1,1,1,1,12,2,2,2,2,3,3,3,3,3,4,5,5,5,6]
Counter(l)
# output: Counter({3: 5, 1: 4, 2: 4, 5: 3, 12: 1, 4: 1, 6: 1})
s = "aaaasssssssvvvvsssaasassjjjdjjd"
Counter(s)
# output: Counter({'s': 12, 'a': 7, 'j': 5, 'v': 4, 'd': 2})
s = "Hello this is a sentence containing lots of hellos and hello statments hello hello hello"
Counter(s.split())
# output: Counter({'hello': 4, 'Hello': 1, 'this': 1, 'is': 1, 'a': 1, 'sentence': 1, 'containi
c = Counter(s.split())
c.most_common(2) # Two most common
# output: [('hello', 4), ('Hello', 1)]
```

`sum(c.values())`	total of all counts
`c.clear()`	reset all counts
`list(c)`	list unique elements
`set(c)`	convert to a set
`dict(c)`	convert to dict
`c.items()`	convert to a list of (elem,cnt) pairs
`Counter(dict(list_of_pairs))`	convert from a list of (elem,cnt) to pairs
`c.most_common()[:-n-1:-]`	n least comon elements
`c += Counter()`	remove zero and negative counts

10.2 defaultdict

- defaultdict will never raise a KeyError, any key that isn't found returns the value of the default factory.

```
from collections import defaultdict
d = defaultdict(object)
d['one']
```

```python
# output: <object object at 0x0000019728D8E1B0>
d = defaultdict(lambda: 0) # lambda that returns 0
d['one']
# output: 0
d['two'] = 2
d
# output: defaultdict(<function <lambda> at 0x0000019728DDC1E0>, {'one': 0, 'two': 2})
```

10.3 OrderedDict

- This data structure considers the order in which elements were added to a list.

```python
d = {}
d['a'] = 1
d['b'] = 2
d['c'] = 3
d['d'] = 4
d['e'] = 5
d
# output: {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

for k,v in d.items():
    print(k,v)
# output: a 1
# output: c 3
# output: e 5
# output: b 2
# output: d 4

from collections import OrderedDict
d = OrderedDict()
d['a'] = 1
d['b'] = 2
d['c'] = 3
d['d'] = 4
d['e'] = 5
d
# output: a 1
# output: b 2
# output: c 3
# output: d 4
# output: e 5

d1 = {}
d1['a'] = 1
d1['b'] = 2

d2 = {}
d2['a'] = 2
```

```
d2['b'] = 1

print(d1 == d2)
# output: True

d1 = OrderedDict()
d1['a'] = 1
d1['b'] = 2

d2 = OrderedDict()
d2['b'] = 2
d2['a'] = 1

print(d1 == d2)
# output: False
```

10.4 namedtuple

- This allows to create a class on one line.

```
from collections import namedtuple
Dog = namedtuple('Dog','age breed name')
sam = Dog(age=2,breed='Lab',name='Sammy')
sam.age
# output: 2
sam[0]
# output: 2

Cat = namedtuple('Cat','fur claws name')
c = Cat(fur='Fuzzy',claws=False,name='Kitty')
c.name
# output: 'Kitty'
c[2]
# output: 'Kitty'
```

10.5 Datetime

```
import datetime
t = datetime.time(5,25,1)
print(t)
t.hour
# output: 5
t.minute
# output: 25
t.second
# output: 1
print(datetime.time.min)
# output: 00:00:00
print(datetime.time.max)
```

```
# output: 23:59:59.999999
print(datetime.time.resolution)
# output: 0:00:00.000001

today = datetime.date.today()
print(today)
# output: 2020-05-25
today.timetuple()
# output: time.struct_time(tm_year=2020, tm_mon=5, tm_mday=25, tm_hour=0, tm_min=0, tm_sec=0, t
today.year
# output: 2020
today.month
# output: 4
today.day
# output: 25
print(datetime.date.min)
# output: 0001-01-01
print(datetime.date.max)
# output: 9999-12-31
print(datetime.date.resolution)
# output: 1 day, 0:00:00

d1 = datetime.date(2015,3,11)
print(d1)
# output: 2015-03-11
d2 = d1.replace(year=1900)
print(d2)
# output: datetime.date(1900, 3, 11)
d1-d2
# output: datetime.timedelta(days=42003)
```

10.6 Python debugger

- python debugger is called pdb
- The pbd module allows for an interactive debugging environment after the excecution of `pbd.set_trace()`

```
import pdb
x = [1,3,4]
y = 2
z = 3
result = y + z
print(result)

pdb.set_trace()

result2 = y + x
print(result2)
# output: import pdb
x = [1,3,4]
```

```
y = 2
z = 3
result = y + z
print(result)

pdb.set_trace()

result2 = y + x
print(result2)
# output: -> result2 = y + x
# output: (Pdb) y
# output: 2
# output: (Pdb) x
# output: [1, 3, 4]
# output: (Pdb) y+x
# output: *** TypeError: unsupported operand type(s) for +: 'int' # output: and 'list'
# output: (Pdb) x+z
# output: *** TypeError: can only concatenate list (not "int") to # output: list
# output: (Pdb) y+x
# output: *** TypeError: unsupported operand type(s) for +: 'int' # output: and 'list'
# output: (Pdb) y+z
# output: 5
# output: (Pdb) q
```

10.7 Timing your code

- timeit

```
import timeit
timeit.timeit('"-".join(str(n) for n in range(100))',number=1_000)
# output: 0.018360799999999955
timeit.timeit('"-".join([str(n) for n in range(100)])',number=10_000)
# output: 0.16509299999999993
timeit.timeit("-".join(map(str,range(100))),number=10_000)
# output: 0.00012020000000001474
```

10.8 Regular expressions

```
import re
patterns = ["term1","term2"]
text = 'This is a string with term1, but not the other term'
re.search('hello','hello world!')
# output: <re.Match object; span=(0, 5), match='hello'>
match = re.search(patterns[0],text)
type(match)
# output: <class 're.Match'>
match.start()
# output: 22
match.end()
```

```
# output: 27
split_term = '@'
phrase = 'What is your email, is it hello@gmail.com'
re.split(split_term,phrase)
# output: ['What is your email, is it hello', 'gmail.com']
'hello world'.split()
# output: ['hello', 'world']
re.findall(pattern="match",string="Here is one match, here is another match")
# output: ['match', 'match']
```

- Repetition syntax:

  ```
  def multi_re_find(patterns,phrase):
      for pattern in patterns:
          print(f"Searching the phrase using the re check: {pattern}")
          print(re.findall(pattern,phrase))

  test_phrase = 'sdsd..sssddd...sdddsddd...dsds...dsssss...sdddd'
  test_patterns = [
      'sd*', # s followed by zero or more d's
      'sd+', # s followed by one or more d's
      'sd?', # s followed by zero or one d's
      'sd{3}', # s followed by three d's
      'sd{2,3}' # s followed by two to three d's
  ]
  multi_re_find(test_patterns,test_phrase)

  # output: Searching the phrase using the re check: sd*
  # output: ['sd', 'sd', 's', 's', 'sddd', 'sddd', 'sddd', 'sd', 's',  's', 's', 's', 's', '
  # output: Searching the phrase using the re check: sd+
  # output: ['sd', 'sd', 'sddd', 'sddd', 'sddd', 'sd', 'sdddd']
  # output: Searching the phrase using the re check: sd?
  # output: ['sd', 'sd', 's', 's', 'sd', 'sd', 'sd', 'sd', 's', 's',  's', 's', 's', 's', 's
  # output: Searching the phrase using the re check: sd{3}
  # output: ['sddd', 'sddd', 'sddd', 'sddd']
  # output: Searching the phrase using the re check: sd{2,3}
  # output: ['sddd', 'sddd', 'sddd', 'sddd']
  ```

- Character sets:

  ```
  test_phrase = 'sdsd..sssddd...sdddsddd...dsds...dsssss...sdddd'
  test_patterns = [
      '[sd]', # either s or d
      's[sd]+' # s followed by one or more s or d
  ]
  multi_re_find(test_patterns,test_phrase)
  # output: Searching the phrase using the re check: [sd]
  # output: ['s', 'd', 's', 'd', 's', 's', 's', 'd', 'd', 'd', 's', 'd', 'd', 'd', 's', 'd',
  # output: Searching the phrase using the re check: s[sd]+
  # output: ['sdsd', 'sssddd', 'sdddsddd', 'sds', 'sssss', 'sdddd']
  ```

- Exclusion:

```
test_phrase = 'This is a string! But it has punctuation. How can we remove it?'
re.findall('[^!.? ]+',test_phrase)
# output: ['This', 'is', 'a', 'string', 'But', 'it', 'has', 'punctuation', 'How', 'can', '
```

- Character ranges:

```
test_phrase = 'This is an example sentence. Lets see if we can find some letters.'
test_patterns = [
    '[a-z]+', # sequences of lower case letters
    '[A-Z]+', # sequences of upper case letters
    '[a-zA-Z]+', # sequences of lower or upper case letters
    '[A-Z][a-z]+' # one upper case letter followed by lower case letters.
]
multi_re_find(test_patterns,test_phrase)
# output: Searching the phrase using the re check: [a-z]+
# output: ['his', 'is', 'an', 'example', 'sentence', 'ets', 'see', 'if', 'we', 'can', 'fin
# output: Searching the phrase using the re check: [A-Z]+
# output: ['T', 'L']
# output: Searching the phrase using the re check: [a-zA-Z]+
# output: ['This', 'is', 'an', 'example', 'sentence', 'Lets', 'see', 'if', 'we', 'can', 'f
# output: Searching the phrase using the re check: [A-Z][a-z]+
# output: ['This', 'Lets']
```

- Escape Codes:

\d	a digit
\D	a non-digit
\s	whitespace (tab,space,newline,etc.)
\S	non-whitespace
\w	alphanumeric
\W	non-alphanumeric

```
test_phrase = 'This is a string with some numbers 1233 and a symbol #hashtag'
test_patterns = [
    r'\d+', # sequense of digits
    r'\D+', # sequense of non-digits
    r'\s+', # sequense of whitespace
    r'\S+', # sequense of characters
    r'\w+', # alphanumeric characters
    r'\W+' # non-alphanumeric
]
multi_re_find(test_patterns,test_phrase)
# output: Searching the phrase using the re check: \d+
# output: ['1233']
# output: Searching the phrase using the re check: \D+
# output: ['This is a string with some numbers ', ' and a symbol #hashtag']
# output: Searching the phrase using the re check: \s+
# output: [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
# output: Searching the phrase using the re check: \S+
# output: ['This', 'is', 'a', 'string', 'with', 'some', 'numbers', '1233', 'and', 'a', 'sy
# output: Searching the phrase using the re check: \w+
# output: ['This', 'is', 'a', 'string', 'with', 'some', 'numbers', '1233', 'and', 'a', 'sy
# output: Searching the phrase using the re check: \W+
# output: [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' #']
```

10.9 StringIO

- Allows us to treat a string as a file.

```
import StringIO
message = 'This is just a normal string'
f = StringIO.StringIO(message)
f.read()
f.write(' Second')
f.seek(0)
```

Chapter 11

Advanced Python Objects and Data Structures

11.1 Advanced Numbers

11.1.1 Hexadecimal numbers

```
hex(12)
# output: '0xc'
hex(512)
# output: '0x200'
```

11.1.2 Binary numbers

```
bin(1234)
# output: '0b10011010010'
bin(128)
# output: '0b10000000'
bin(512)
# output: '0b1000000000'
```

11.1.3 Pow function

```
pow(2,4) # With three args the third one is mod.
# output: 16
```

11.1.4 Abs function

```
abs(-2)
# output: 2
```

11.1.5 Round function

```
round(3)
# output: 3.0
round(3.9)
# output: 4
```

```
round(3.141592,2)
# output: 3.14
```

11.2 Advanced strings

```
s = 'hello world'
s.capitalize()
# output: 'Hello world'
s.upper()
# output: 'HELLO WORLD'
s.lower()
# output: 'hello world'
s.count('o')
# output: 2
s.find('o')
# output: 4
s.center(20,'z')
# output: 'zzzzhello worldzzzzz'
'hello\thi'.expandtabs()
# output: 'hello    hi'
s.isalnum()
# output: True
s.isalpha()
# output: True
s.islower()
# output: True
s.isspace()
# output: False
s.istitle()
# output: False
s.isupper()
# output: False
s.endswith('o')
# output: True
s.split('e')
# output: ['h','llo']
s = 'hihihiiiihihii'
s.partition('i') # splits only the first instance
# output: ('h', 'i', 'hihiiiihihii')
```

11.3 Advanced sets

- ⟨set1⟩ .add(⟨element⟩): adds elements to a set, if they are not already in there.

  ```
  s = set()
  s.add(1)
  s.add(2)
  s
  # output: {1,2}
  ```

```
s.add(2)
s
# output: {1,2}
```

- ⟨set1⟩ .clear(): clears the set of all its elements.

```
s.clear()
```

- ⟨set1⟩ .copy(): copies all elements in set to another set.

```
s = {1,2,3}
sc = s.copy()
sc
s.add(4)
# output: {1,2,3}
```

- ⟨set1⟩ .difference(⟨set2⟩): returns a set containing the differences between set1 and set2.

```
s.difference(sc)
# output: {4}
```

- ⟨set1⟩ .difference_update(⟨set2⟩): it's the same as the .difference() only that the returning set will be assigned to set1.

```
s1 = {1,2,3}
s2 = {1,4,5}
s1.difference_update(s2) # does the same as diference() just that the return from the dife
s1
# output: {2,3}
```

- ⟨set1⟩ .discard(⟨element⟩): eliminates an element from the set.

```
s
# output: {1,2,3,4}
s.discard(2)
# output: {1,3,4}
```

- ⟨set1⟩ .intersection(⟨set2⟩): returns the elements that are common in both sets, or the intersection of the two sets.

```
s1 = {1,2,3}
s2 = {1,2,4}
s1.intersection(s2) # elements that are common to both of the sets
# output: {1, 2}
```

- ⟨set1⟩ .isdisjoint(⟨set2⟩): returns True if the sets are disjoint, which means that the sets don't have anything in common.

```
s1 = {1,2}
s2 = {1,2,4}
s3 = {5}
s1.isdisjoint(s2) # returns True if they don't have anything in common.
# output: False
```

- ⟨set1⟩ .issubset(⟨set2⟩): returns True if set2 is a super set of set1, or set1 is contained in set2.

```
s1.issubset(s2) # {1,2} is a subset of {1,2,4} returns true
# output: True
```

- ⟨set1⟩ .issuperset(⟨set2⟩): the complete inverse of .issubset, here if set2 is a super set of set1 returns true.

```
s2.issuperset(s1) # is the inverse of the issubset()
# output: True
```

- ⟨set1⟩ .symmetric_difference(⟨set2⟩): returns the elements that only show up in one of the sets.

```
s1.symmetric_difference(s2) # the elements that are only in one of the sets.
# output: {4}
```

- ⟨set1⟩ .union(⟨set2⟩): returns the union or the elements that are in either set, all of the ven diagram.

```
s1.union(s2) # elements that are in either set.
# output: {1,2,4}
```

- ⟨set1⟩ .update(⟨set2⟩): it's the same as .union() but the returning set is stored in s1.

```
s1.update(s2) # it's the same as the union, only that the result is stored in s1.
```

11.4 Advanced Dictionaries

Dictionary comprehension:

```
d = {x:x**2 for x in range(10)}
d
# output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
d = {k:v**2 for k,v in zip(['a','b'],range(10))}
d
# output: {'a': 0, 'b': 1}
```

11.5 Advanced lists

```
x = [1,2,3]
x.extend([4,5])
x
# output: [1, 2, 3, 4, 5]

x.index(2) # you will get an error if the index isn't in list
# output: 3
x.insert(2,'inserted')
x
# output: [1, 2, 'inserted', 3, 4, 5]

x.remove('inserted') # removes the forst occurance of a value
x
# output: [1, 2, 3, 4, 5]
```

```
x.reverse()
x
# output: [5, 4, 3, 2, 1]

x.sort()
x
# output: [1, 2, 3, 4, 5]
```