Complete Python Developer in 2023: Zero to Mastery

Status	In progress
Assign	

https://www.udemy.com/course/complete-python-developer-zero-to-mastery/

▼ Links

Python cheat sheet

https://www.udemy.com/course/complete-python-developer-zero-to-mastery/

All python codes

https://www.udemy.com/course/complete-python-developer-zero-to-mastery/

The ultimate guide to coding interview

https://www.udemy.com/course/complete-python-developer-zero-to-mastery/

▼ Section 2 - Python Introduction

- What is a programming language
    - High and low-level languages
- Interpreter vs compiler
- Python official package → **cpython** (written in C)
    - Python code → interpreter → cpython VM → machine code

- Python 2 vs. Python 3
- Why we need different programming language
    - Application dependent

▼ Section 3 - Python Basics I

- Data types → what type of data/values a program can hold
- Programming goal = perform actions
- Types
    - Python *fundamental data types*
        - int
        - float
        - bool
        - str
        - list
        - tuple
        - set
        - dict
    - classes → custom types
    - We also have *specialized data types*
        - packages and modules
    - None
- Actions in programming are called *functions*
    - math functions
        - Whenever you need, go to python official **math** documentation and check available functions.

- **Developer Fundamentals**
    - Don’t try to learn every single thing.
    - Understand what exists and what you can use.
    - You can always google things.
- Operators precedence
- We also have `complex` data type for complex numbers
- `bin()` return binary version of an integer
- Variables → store information
    - snake_case
    - Case sensitive
    - Start with lowercase or underscore
    - Don’t overwrite *keywords*
    - Constants → never change → global variables
        - PI = 3.14
    - Dunder variables → __
        - Don’t make custom dunder variables
- Expressions vs. Statements
    - Expression → a piece of code that produces a value
        - user_age = var / 5 → `var / 5` is an expression
    - Statement → an entire line of code
- Augmented assignment operator
    - +=, -=
- Type conversion
- Escape sequence
    - `\`
        - Whatever comes immediately after it is considered a string

    - can also use `‘””’` or `“’’”`
    - To write `\` use `\\`
    - `\t` -> Tab
- Formatted strings
    - f-string `f”...”`
- String slicing
    - `[start:stop:step]`
    - `[::-1]` → reverse the string
- String in python are *immutable*.
- Built-in functions vs. methods
    - `len()`
    - Methods are owned by something
        - e.g. type specific methods like *string methods*
        - `.format()`
        - `islower()`
        - `upper()`
        - `find()`
        - `replace()`
- Add comments to your code
- *List* can contain different objects
    - mutable
    - *items are next to each other in the memory*
    - If we assign list to a variable without `[:]` or `.copy()`, it will modify the original as well
        - memory location
- Data structure → container around your data

    - access data, remove data, write etc.
- Copying vs. modifying the list
- Matrix is a multidimensional list/array
    - pixels in CV
- List methods
    - `append()` changed the list inplace. It does not create a new list.
    - `insert()` can append at specified location
    - `extend()` takes an *iterable* i.e a list.
    - `pop()` takes index, `remove()` takes the value
        - `pop()` also return the value which is removed
    - `clear()` clears the list
    - `index()` return index, also provides option for start and end index
    - `.sort()` vs `sorted()`
        - `sorted()` creates. new copy of the list
    - `.join()` → creates new item, so we have to assign
- `None` is a special data type in python
    - `Null` in other values
    - It represents absence of value
- Dictionaries → unordered key: value pair
    - *unordered → not next to each other in the memory*
    - *We can have lists inside dicts and vice versa*
- **Develop Fundamentals**
    - Understanding data structures

> 💡 A good programmer knows what data structure to use when

- List has order, Dict has no order
- Dict keys themselves cannot be changed
    - We can use list as a key because list is mutable
- Dict methods
    - `get()` can be used to return *default value* if key is not found
    - `key()`, `values()`, `items()`
    - `popitem()` → removes random pair
    - `update()`
        - updates a k,v pair
        - if pair does not exist, it appends the new pair
- Tuple
    - immutable lists
    - cannot sort or reverse etc.
    - can access items
    - used when we do not want to change
    - *faster than lists*
    - `count()` and `index()` main tuple methods
- Set → unordered collection of *unique* objects
    - *For example if we want to avoid duplicate email in a database*
    - does not support indexing
    - `difference()`, `difference_update()`
    - `intersection()`
        - can also use “&” for it
    - `isdisjoint()` → nothing in common
    - `union()` also removes the duplicates
        - can also use “|” for it

    - `issubset()` and `issuperset()`

▼ Section 4 - Python Basics II

- Conditional logic
    - `if-else` blocks
    - `elif`
- Indentation is important in all languages
    - other languages use `{}` for blocks
        - spacing is only for styling
    - In python, spacing is not only for styling. It is to help the interpreter.
- `if` automatically performs type conversion
- Python *Truthy and Falsy* values
- Ternary operator → do something after evaluation expression
    - Short and ncie
- Short circuiting
    - `or` operator
    - `and` operator
    - evaluates one or few condition instead of all
    - ignores rest → efficient
- Logical Operators
    - `<, >, and, or, ==`
    - `>=, <=, !=`
    - `not`
        - also works as a function
    - comparison on characters “a”, “b” used ASCII

        - `ord()` → get ASCII code
        - `chr()` convert ASCII code to character
- is vs. ==
    - == → checks quality in value
    - is → check if memory location of both values is same
    - data structures are created at different memory spaces
    - but constants and values are in similar locations so `is` can return `True`
- for loops, nested for loops
- Iterable → can be iterated over
    - list, dict, set, tuple, string etc.
    - *iterate* → the action of iterating over an iterable
- `range()` is used hand in hand with loops

```
for i in range(0, 10, 2):
    print(i)
for i in range(10, 0, -1):
    print(i)
```

- `enumerate()` → also return index along with the values

```
for idx, val in enumerate(["a", "b", "c"]):
    print(idx, val)
for idx, val in enumerate(list(range(10))):
    print(idx, val)
for idx, val in enumerate(list(range(10))):
    if val == 5:
        print(f"Index of 5 is {idx}.")
```

- While loop
    - `else` works with while

        - execute only if there isn't a `break` and while condition turns to `false`

- When to use *for loop* and when to use *while loop*?
    - `for` loops are simples
    - `while` are more flexible
    - for simple and iterables, `for` loops are good
    - if you don't know how many times you need to loop, use `while`
    - `while True:`
- Break, continue and pass
    - break → moves out of the loop
    - continue → sends to start of loop, starts next iteration and ignores remaining commands
    - pass → just passes to next line
        - *Good way to have `placeholders` in your code*
        - *I will implement something here later*
- `print("hello", end="")` → no newline added at the end
- GUI exercise

```
# GUI exercise
picture = [
    [0, 0, 0, 1, 0, 0, 0],
    [0, 0, 1, 1, 1, 0, 0],
    [0, 1, 1, 1, 1, 1, 0],
    [1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0],
]

for row in picture:
    for pixel in row:
        print(" ", end="") if pixel == 0 else print("*", end="")
    print("")  # prints a new line by default
```

- **Developer Fundamentals**
    - What is a good code?
        - clean, readable, comments
        - predictability
        - DRY → do not repeat yourself → functions
        - use variables and placeholders
- Functions
    - define and call/invoke
    - Intrepreter goes line by line
        - Define function first
    - parameter and arguments
    - **positional arguments** → position matters
        - argument should be passed in same order as the parameters
        - easy to read
    - **keyword arguments**
        - Don't worry about position
    - Default parameters → safety feature
    - If no `return()`, then it returns `None`
    - Function
        - should do one thing well
        - should return something
    - We can also have nested function
        - Be careful with return statements
        - Outer function returns inner function
            - Inner function is not run, it is just defined
                - It runs later when it is called indirectly

  - `return` exits the function
  - Methods vs. functions
  - Dock strings
  - Clean code

```
# Clean code
# def is_even(num):
#     if num % 2 == 0:
#         return True
#     elif num % 2 != 0:
#         return False


# def is_even(num):
#     if num % 2 == 0:
#         return True
#     else:
#         return False


# def is_even(num):
#     if num % 2 == 0:
#         return True
#     return False


def is_even(num):
    return num % 2 == 0


print(is_even(50))
```

  - Rule → params, *args, default parameters, **kwargs
- Walrus operator → assigns expressions value to a variable
  - New feature in python 3.8

```
a = "helloooooooo"
# if len(a) > 10:
#     print(f"Too long {len(a)} elements")
if (n := len(a)) > 10:
    print(f"Too long {n} elements")
```

```
while (n := len(a)) > 1:
    print(n)
    a = a[:-1]
print(a)
```

- Scope
  - *What variables do I have access to?*
  - global scope, function scope
  - New scope only comes with functions
    - not with `if` statements
  - Starts with local scope, then checks parent local scope, global scope in the end, then built in python functions
- `global` keyword → to access global variables outside the function scope
- Dependency injection → no `global`, we detached dependency
- `nonlocal` keyword
  - New keyword
  - makes code more complicated tho
- Try to avoid using `global` and `nonlocal` if you can

> 💡 Python *garbage collector* automatically removes variables from memory after the function has been execute
>
> - That is why `scope` is useful.

▼ Section 5 - Development Environment

- Terminal
- Code Editors → lightweight
    - VS Code
    - Sublime Text
- IDE
    - Pycharm
    - Spyder
- Jupyter Notebook

▼ Section 6 - Advanced Python: Object Oriented Programming

- Everything in python is an object
- Procedural code to OOP
- `Class` is a blueprint
    - `Objects` are built using a `Class` blueprint
    - *Objects = Instances*
    - Instantiate the class
- Constructor or init method
    - Instantiate the class
        - Takes arguments
    - `self` refers to that instance of the class
- Class Object Attribute → static variables
    - Defined on same level as method

    - Class-based
    - *Does not change across instanced*
- Attributes → change, dynamic
    - Instance-based
- We can also directly pass arguments to class methods
- classmethod → `cls` is required in parameters → first parameters is `cls` just like `self`
    - work without instances
    - not used often
    - We use it when we want to modify and change the attributes of class
    - A class method is a method that is bound to the **class** and not the object of the class.
    - They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance.
    - It can modify a class state that would apply across all the instances of the class.
- staticmethod → perform some kind of task
    - We don't have access to `cls` parameters
    - We use this when we don't care about the class state
        - like attributes of the class
- *We generally use the class method to create factory methods. Factory methods return class objects (similar to a constructor) for different use cases.*
- *We generally use static methods to create utility functions.*
- **Developer Fundamental** → *Test your assumptions*
    - What do you expect the outcome to be
    - Test and compare the actual output with your assumption
- **Four pillars of OOP**

- Encapsulation → Binding of data and functions
  - attributes and methods
  - Defining methods and attributes in an object instead of outside is called encapsulation. We package everything in one object.
  - We need both attributes and methods to do something useful.
- Abstraction → hiding information that user does not need to see
  - When use functions like `len()` `count()` etc., we don't want to know the details
    - Things are abstracted from us
  - Public and private variables
    - No true privacy in python
    - But we have convention to start private variable names with `_`
- Inheritance
  - Parent and children classes (derived / sub classes)
  - `isinstance` → if object is an instance of another class → return bool
  - Python comes with base `object` class
    - All classes inherit from this class by default
      - That is why we already have many methods available when we create a new class
- Polymorphism → Many forms
  - Object classes can share same method names
    - But these methods can act in differently, based on what object calls them
  - We can customize function according to our needs

- `super()` → get access to attributes of parent class
- Obejct Introspection
- Dunder methods → special methods

    - We can customize them (sometimes)
        - They are modified only for that specific objects

```
class Toy:
    def __init__(self, color, age):
        self.color = color
        self.age = age
        self.my_dict = {"name": "yoyo", "has_pets": False}

    def __str__(self):
        return f"{self.color}"

    def __len__(self):
        return 5

    def __del__(self):
        print("Deleted")

    def __call__(self):
        return "yes??"

    def __getitem__(self, i):
        return self.my_dict[i]


action_figure = Toy("red", 0)
print(action_figure.__str__())
print(str(action_figure))
print(len(action_figure))
print(action_figure())  # call method
# del action_figure
print(action_figure["name"])
```

- `issubclass()` → Inheritence → return bool
- Multiple Inheritance
    - Can get complicated
    - Have to initialize each parent separately
- MRO Method Resolution Order
    - Which method to run
    - `print(class.mro())` or `class.__mro__`

  - DFS (Depth First Search) is used for it

▼ Section 7 - Advanced Python: Functional Programming

- Another paradigm, just like OOP
- Separation of concerns
- Separate data and functions (unlike OOP)
  - *attributes and functions are not encapsulated*
- Clear and understandable, easy to extend, easy to maintain, memory efficient, DRY
- **Pure functions**
  - Given same input, always returns same output
  - Function should not produce any side effects → affects the outside world
  - A function with `print` state interacts with outside display → not pure function
  - Modifying a list outside the function → not pure
  - *Less buggy code, easy to understand, part separation*
  - More like a guideline
- `map()` → simplifies list operations → applies function on data
  - pure function
- `filter()` → also removes items, applies filter
- `zip()` → zips two iterables together

  ```
  my_list = [1, 2, 3]
  your_list = [4, 5, 6]
  print(list(zip(my_list, your_list)))
  ```

- `reduce` is not builtin python function → reduce list to few values or one

```
from functools import reduce

my_list = [1, 2, 3]


def accumulator(acc, item):
    print(acc, item)
    return acc + item  # result becomes the new acc


print(reduce(accumulator, my_list, 10))  # param3 is initial value for acc
```

  - `map` and `filter` are using `reduce` under the hood

- **lamda expressions**
  - For functions only used once
  - Anonymous functions
  - used together with `map()`

```
print(list(map(lambda item: item * 2, my_list)))
```

  - Code can be confusing

- Sort with different keys

```
# sorting based on second item
a = [(0, 2), (4, 3), (9, 9), (10, -1)]
# a.sort()  # sorts based on first item
# print(a)
a.sort(key=lambda x: x[1])  # we cans et key
print(a)
```

- List comprehensions

```
even_list = [num for num in double_list if num % 2 == 0]
double_list = [num**2 for num in num_list]
```

- Set comprehension → just change brackets

```
my_set = {char for char in "hello"}
```

- Dict comprehension

```
# dict
simple_dict = {"a": 1, "b": 2}
my_dict = {key: value**2 for key, value in simple_dict.items() if value % 2 == 0}
print(my_dict)
```

▼ Section 8 - Advanced Python: Decorators

- Higher Order Function
    - Accepts a function as a parameter
    - Or returns a function
    - `map()`, `reduce()` etc.

```
# Higher Order Function HOC
def greet(func):
    func()


def greet2():
    def func():
        return 5
    return func
```

- Decorators
    - We already saw `classmethod` and `staticmethod`
    - They supercharge our functions
        - Enhances another function
    - Decorator pattern → `*args`, `**kwargs` make decorators useful

```
def my_decorator(func):
    def wrap_func(*args, **kwargs):
        func(*args, **kwargs)
    return wrap_func
```

    - Create `performance` decorator → calculate function runtime
        - If you want to check time for a step like opencv, make a function of that step and wrap it with your wrapper

```
from time import time


def performance(fn):
    def wrapper(*args, **kwargs):
        t1 = time()
        result = fn(*args, **kwargs)
        t2 = time()
        print(f"Took {t2-t1} sec.")
        return result

    return wrapper


@performance
def long_time():
    for i in range(10000000):
        i * 5
```

```
long_time()
```

- Authentication example

```
user1 = {"name": "Sorna", "valid": True}
user2 = {"name": "Sorna", "valid": False}


def authenticated(fn):
    def is_valid(*args, **kwargs):
        if args[0]["valid"] == True:
            fn(*args, **kwargs)

    return is_valid


@authenticated
def message_friends(user):
    print("message has been sent.")


message_friends(user1)
```

▼ Section 9 - Advanced Python: Error Handling

- Error that crashes our program is called an *exception*
- If you are a big tech company, error can cause a lot of problems
- Error handling
    - Let python program to keep running despite there errors
    - `try-except-else`
    - Using `except` with no exception does not give us proper information

- Always add exceptions like `ValueError`, `TypeError` etc.

```
while True:
    try:
        age = int(input("What is your age? "))
        print(age)
    except:
        print("please enter a number.")
    else:
        print("Thank you!")
        break
```

```
def sum(num1, num2):
    try:
        return num1 + num2
    except TypeError as err:
        print(err)


print(sum("1", 2))
```

```
def sum(num1, num2):
    try:
        return num1 / num2
    except (TypeError, ZeroDivisionError) as err:
        print(err)


print(sum("1", 0))
```

```
while True:
    try:
        age = int(input("What is your age? "))
        10 / age  # 0 division
    except ValueError:
        print("please enter a number.")
    except ZeroDivisionError:
        print("please enter a number higher than 0.")
    else:
        print("Thank you!")
        break
```

- Syntax error, name error, index error, KeyError, zero division error
- `try-except-else-finally`
    - `finally` runs at the end of everything (try, except, else)

```
while True:
    try:
        age = int(input("What is your age? "))
        10 / age  # 0 division
    except ValueError:
        print("please enter a number.")
        continue
    except ZeroDivisionError:
        print("please enter a number higher than 0.")
        break
    else:
        print("Thank you!")
    finally:
        print("ok, finally done!")
    print("can you hear me?")  # doesn't run if we break out of the loop
```

- Sometimes we want to stop program execution with if an error occurs
    - raise errors or exceptions

```
# What if we want to stop the program with error??
while True:
    try:
        age = int(input("What is your age? "))
        10 / age  # 0 division
        raise ValueError("Hey, cut it out!")
        #raise Exception("Hey, cut it out!")
    except ZeroDivisionError:
        print("please enter a number higher than 0.")
        break
    else:
        print("Thank you!")
    finally:
        print("ok, finally done!")
    print("can you hear me?")  # doesn't run if we break out of the loop
```

- As a programmer, we should be able to anticipate errors and bugs, and handle them properly
    - Most error occur when we take input from the users or other programs

▼ Section 10 - Advanced Python: Generators

- Generators → generate a sequence of values over time
    - `yield`
        - pauses the function
        - `next()` restarts function from next iteration
            - `StopIteration` error when you reach the limit of the `range`.
        - It remembers last state
    - `range()` is also a generator
        - generates numbers one by one
        - does not create a list in the memory
    - Every generator is an iterable, but every iterable is not a geenrator
    - Instead of creating whole list/object in memory, it goes one by one
        - We hold only one item in memory

```
# Generator


# def generator_function(num):
#     for i in range(num):
#         yield i * 2


# g = generator_function(100)
# print(g)
# print(next(g))
# next(g)
# next(g)
```

```python
# print(next(g))

# for item in generator_function(1000):
#     print(item)

# def make_list(num):
#     result = []
#     for i in range(num):
#         result.append(i*2)
#     return result

from time import time


def performance(fn):
    def wrapper(*args, **kwargs):
        t1 = time()
        result = fn(*args, **kwargs)
        t2 = time()
        print(f"Took {t2-t1} sec.")
        return result

    return wrapper


@performance
def long_time():
    print("1")
    for i in range(100000000):
        i * 5


@performance
def long_time2():
    print("2")
    for i in list(range(100000000)):
        i * 5


long_time()
long_time2(
```

```
1
Took 4.702873945236206 sec.
2
Took 8.832290887832642 sec.
```

- Really useful and efficient
- `for loop` use generator

```
def special_for(iterable):
    iterator = iter(iterable)
    while True:
        try:
            print(iterator)
            print(next(iterator))
        except StopIteration:
            break


special_for([1, 2, 3])
```

```
<list_iterator object at 0x10892ef40>
1
<list_iterator object at 0x10892ef40>
2
<list_iterator object at 0x10892ef40>
3
<list_iterator object at 0x10892ef40>
```

- `range()`

```
class MyGen:
    current = 0

    def __init__(self, first, last):
        self.first = first
        self.last = last

    def __iter__(self):
        return self

    def __next__(self):
        if MyGen.current < self.last:
            num = MyGen.current
            MyGen.current += 1
            return num
        raise StopIteration  # no more things to iterate over
```

```
gen = MyGen(0, 100)
for i in gen:
    print(i)
```

- fibonacci with gen
  - Common interview question

```
def fib(num):
    a = 0
    b = 1
    for i in range(num + 1):
        yield a
        temp = a
        a = b
        b = temp + b


for x in fib(20):
    print(x)
```

▼ Section 11 - Modules in Python

- Up till now, every thing is in one file
- Organize in modules
  - `import`
  - `pycache` → compile and cache imported modules (c-compiler **cython**)
    - Next time, module is not loaded. Instead compiled version is loaded.
    - runs faster
    - recompiled if something changes
- **Package** → a folder containing modules

    - `__init__.py` is used to initialize the package
        - It can be empty
        - we need to have it
    - You can use `__init__.py` to configure the package when it is imported. For example, you can set variables or perform other initialization tasks that are needed for your package to work correctly.
- Different ways to import packages
    - Try to be explicit, don't just say `from utilities import *`
- Name collisions
- if name is main then run
    - Run only if it is main file
- Built-in modules
- import with alias → `import numpy as np`
- `sys.argv` → to pass CLI arguments

```
import sys

name = sys.argv[1]
print(f"Hello {name}")
```

- Placing of user input in `try-except` block
    - Careful
        - Can cause infinite loop
- pip and conda install
- pypi → python package index
- Check from builtin modules first
- Versioning

    - 0.5.1
        - 1 → bug fixes
        - 5 → feature releases
        - 0 → major version changes
- Useful modules
    - collections
        - defaultdict
        - Counter
        - OrderedDict
            - Recently, the Python has made Dictionaries ordered by default! So unless you need to maintain older version of Python (older than 3.7), you no longer need to use ordered dict, you can just use regular dictionaries!
    - datetime
    - array
        - Takes less memory and performs faster
        - sets memory in advance
        - code optimization

```python
from collections import Counter, defaultdict, OrderedDict

li = [1, 2, 3, 4, 5, 6, 7, 7]
sentence = "hello, how are you?"


print(Counter(li))
print(Counter(sentence))

dictionary = {"a": 1, "b": 2}
print(dictionary["a"])
dictionary = defaultdict(lambda: 0, {"a": 1, "b": 2})
print(dictionary["c"])

d1 = OrderedDict()
d1["a"] = 1
```

```
d1["b"] = 2

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

print(d1 == d2)

import datetime

print(datetime.time(5, 42, 2))
print(datetime.date.today())

import time

print(time.perf_counter())
print(time.time())

from array import array

arr = array("i", [1, 2, 3])
print(arr)
```

- **Developer fundamentals**
- Pros and Cons of libraries
  - Avoid bad packages
  - More space required
    - Projects gets bigger

▼ Section 12 - Debugging in Python

- Linting
  - allows us to find errors
- Always use editor or an IDE
- Learn to read errors
  - Check on documentation
- `pdb` → python debugger (builtin)

- help → shows documentation
    - help list
- step → go to next line
- a → fives all the arguments
- continue → continues and finishes execution
- w → shows context of current line
- exit → to stop Pdb

```
import pdb


def add(num1, num2):
    # print(num1, num2)
    pdb.set_trace()
    return num1 + num2


add(4, "abc")
```

```
-> return num1 + num2
(Pdb) num1
4
(Pdb) num2
'abc'
(Pdb) help
(Pdb) step
```

- We can also change variable values and test code working

    ```
    (Pdb) a
    num1 = 4
    num2 = 5
    (Pdb) num2 = "abc"
    (Pdb) next
    TypeError: unsupported operand type(s) for +: 'int' and 'str'
    ```

▼ Section 13 - File I/O

- `open()` and `close()`
- `read()`, `readline()` and `readlines()`
- `r+` → read and write mode
    - overwrites
- When you reopen a file, cursor resets
- `w` → removes and writes like a new file
    - Also can create new files
- `a` → append to the end
- file paths
    - absolute and relative
    - `./` and `../`
- **pathlib**
    - works for both paths i.e. windows and unix

▼ Section 14 - Regular Expressions

- RegEx
- Validation checking, piece of string exists etc.
- import re
    - `re.search()`
    - Returns an object
    - Return `None` if not found
    - `re.findall` → all occurences
    - `re.fullmatch` → exact match

    - `re.match` → match and doesn't care what comes after
    - `re.compile` → compile a pattern to search for later
    - `.group()` → re pattern pattern groups
- Useful for advanced patterns
    - Develop as the need arises
    - No one remembers all
    - We can group patterns
        - *Capturing groups*
- Use regex101
- Collecting emails
    - We can use regex to verify
        - Throw away bad input
            - Save database
- `.` → anything
- `+` between and unlimited times before
- `$` end of line

```
"""
email
password:
    1. Can be 8 characters long
    2. Can contain letters, numbers and $@%#
    3. Ends with a number
"""

import re

email_pattern = re.compile(r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$")
pass_pattern = re.compile(r"[a-zA-Z0-9@#$%]{8,}\d$")

email = "b@b.com"
email2 = "asas"
password = "ahgshag121#$5"
password2 = "ahgs12"
```

```
a = email_pattern.search(email)
b = pass_pattern.fullmatch(password)
c = email_pattern.search(email2)
d = pass_pattern.fullmatch(password2)
print(a)
print(b)
print(c)
print(d)
```

▼ Section 15 - Testing in Python

- After debugging, linting, pep-8 etc., testing is an additional layer to check bugs
- `unittest`
  - import functions and run tests
  - `self.assertEqual()`
- We try to break our function to improve it

```
# main.py
def do_stuff(num):
    return int(num) + 5
```

```
# test.py
import unittest
import script


class Testscript(unittest.TestCase):
    def test_do_stuff(self):
        test_param = 10
        result = script.do_stuff(test_param)
        self.assertEqual(result, 15)

    # def test_do_stuff2(self):
    #     test_param = "abc"
    #     result = script.do_stuff(test_param)
    #     self.assertEqual(result, ValueError)

    def test_do_stuff2(self):
        test_param = "abc"
        result = script.do_stuff(test_param)
        # self.assertTrue(isinstance(result, ValueError))
        self.assertIsInstance(result, ValueError)

    def test_do_stuff3(self):
        test_param = None
        result = script.do_stuff(test_param)
        self.assertEqual(result, "please enter number")

    def test_do_stuff4(self):
        test_param = ""
        result = script.do_stuff(test_param)
        self.assertEqual(result, "please enter number")


if __name__ == "__main__":
    unittest.main()
```

- We evolve our main function based on our testing results

```
def do_stuff(num=0):
    try:
        if num:
            return int(num) + 5
        else:
            result = "please enter number"
            return result
    except ValueError as e:
        return e
```

- We use `unittest.main()`

💡 Try to keep you script name other than `main.py`. I tried using `main.py` but intellisense was not working properly because `unittest` has its own main as well. When I renamed it to `script.py` then everything was working smoothly.

- Create `test.py`

    - Then in CLI `python3 -m unittest`
        - It will run all the tests

```
python3 -m unittest
python3 -m unittest -v. # more details
```

- `setUp()` → Set things up
    - runs before each function
        - default variables
- `tearDown()`-> cleanup and reset variables etc.
    - runs after each function

```
# sript.py
def do_stuff(num=0):
    try:
        if num:
            return int(num) + 5
        else:
            result = "please enter number"
            return result
    except ValueError as e:
        return e
```

```
# test.py
import unittest
import script


class Testscript(unittest.TestCase):
    def setUp(self) -> None:
        print("about to test a function")

    def test_do_stuff(self):
        """
        *** Add comments here! ***
        """
        test_param = 10
        result = script.do_stuff(test_param)
        self.assertEqual(result, 15)
    # def test_do_stuff2(self):
    #     test_param = "abc"
    #     result = script.do_stuff(test_param)
    #     self.assertEqual(result, ValueError)

    def test_do_stuff2(self):
        test_param = "abc"
        result = script.do_stuff(test_param)
        # self.assertTrue(isinstance(result, ValueError))
        self.assertIsInstance(result, ValueError)

    def test_do_stuff3(self):
        test_param = None
        result = script.do_stuff(test_param)
        self.assertEqual(result, "please enter number")

    def test_do_stuff4(self):
        test_param = ""
        result = script.do_stuff(test_param)
        self.assertEqual(result, "please enter number")

    def tearDown(self) -> None:
        print("Cleaning up!")


if __name__ == "__main__":
    unittest.main()
```

- Game example

```
# script.py
"""
1. generate a number from 1~10
2. get input from user
3. check input is 1~10
4. check if number is right guess, else guess again
"""

import sys
from random import randint


def run_guess(guess, answer):
    if 0 < guess < 11:
        if guess == answer:
            print("Correct.")
            return True
    else:
        print("I said 1 to 10!")
        return False

if __name__ == "__main__":
    answer = randint(1, 10)

    while True:
        try:
            guess = int(input("Guess a number between 1 to 10:  "))
            if run_guess(guess, answer):
                break
        except ValueError:
            print("Please enter a number.")
            break
```

```
# test.py
import unittest
import game


class TestGame(unittest.TestCase):
    def test_input(self):
        result = game.run_guess(5, 5)
        self.assertTrue(result)

    def test_input_wrong_guess(self):
        result = game.run_guess(5, 10)
        self.assertFalse(result)

    def test_input_wrong_number(self):
        result = game.run_guess(5, 11)
        self.assertFalse(result)

    def test_input_wrong_type(self):
        result = game.run_guess(5, "abc")
        self.assertFalse(result)


if __name__ == "__main__":
    unittest.main()
```

▼ Section 16 - Career of a python developer

- Different directions
    - Data scientist, ML, Backend, automation, python developer etc.

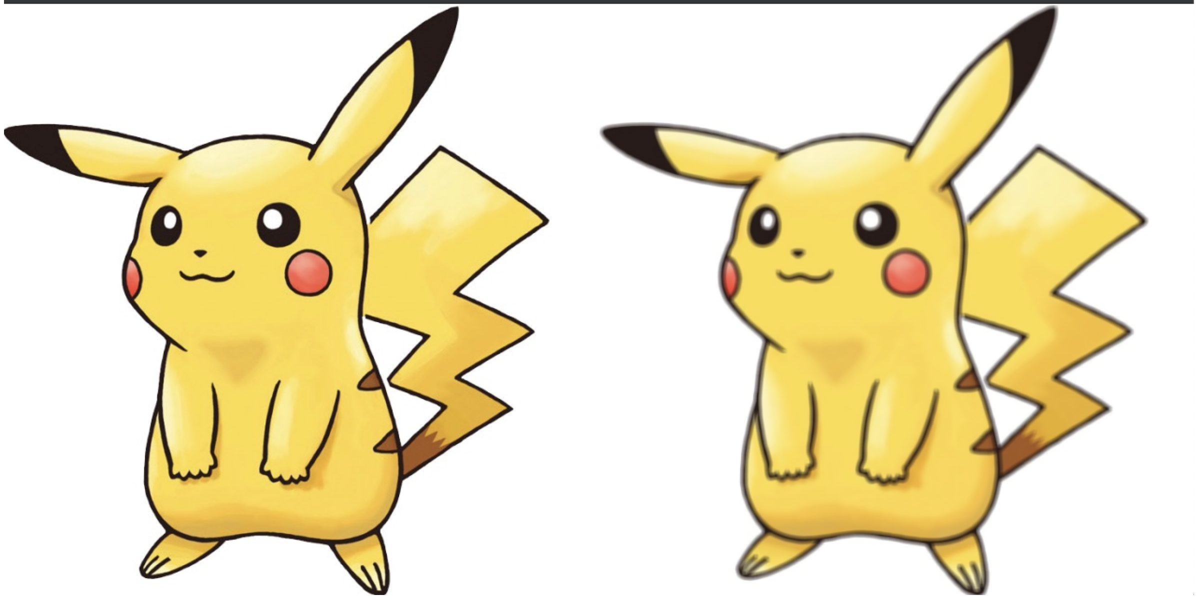
    - Apply for jobs that feel you are under qualified for
        - Opportunity to learn new things
        - If you stay in comfort zone, you don’t learn new things
        - What is this job gonna teach me
            - New things or not?
        - Add things in resume
            - Better opportunities in the future
    - Every application should be a long shot
    - Github, website, blog posts, big projects

Section 17 - Scripting with Python

- Image compression on social websites
- We will use Pillow
- **Developer Fundamentals** → Pick the right library
    - Latest
    - Most people use
- **PIL**
    - Converts images to objects
    - Crop, resize, filter etc.
    - `img.thumbnail` keeps the aspect ratio
        - But it modifies the original image
        - Useful for social media pics
        - max size is `400x400`
            - Use `copy()`

```
import numpy as np
from PIL import Image, ImageFilter
```

```
img = Image.open("./Pokedex/pikachu.jpg")

# print(img)
# print(img.format)
# print(img.size)
# print(img.mode)
# print(dir(img))


filtered_img = img.filter(ImageFilter.BLUR)
# filtered_img = img.filter(ImageFilter.SMOOTH)
# filtered_img = img.filter(ImageFilter.SHARPEN)
# filtered_img = img.convert("L")
# filtered_img = filtered_img.convert("RGB")  # back to 3 channed for np cpncat

# We save image
# filtered_img.save("blur.png", "png")

# Show result
filtered_img.show()

# Show side-by-side
# img = np.asarray(img)
# filtered_img = np.asarray(filtered_img)
# concatenate = np.concatenate((img, filtered_img), axis=1)

# result = Image.fromarray(np.uint8(concatenate))
# result.show()
```



- JPG to PNG converter
    - PNG images are better for websites
- `os.path.splittext(filename)[0]` → to remove file extensions