INTRODUCTION TO PYTHON

Compilation vs. interpretation - advantages and disadvantages

	COMPILATION	INTERPRETATION
ADVANTAGES	• the execution of the translated code is usually faster; • only the user has to have the compiler - the end-user may use the code without it; • the translated code is stored using machine language - as it is very hard to understand it, your own inventions and programming tricks are likely to remain your secret.	• you can run the code as soon as you complete it - there are no additional phases of translation; • the code is stored using programming language, not the machine one - this means that it can be run on computers using different machine languages; you don't compile your code separately for each different architecture.
DISADVANTAGES	• the compilation itself may be a very time-consuming process - you may not be able to run your code immediately after any amendment; • you have to have as many compilers as hardware platforms you want your code to be run on.	• don't expect that interpretation will ramp your code to high speed - your code will share the computer's power with the interpreter, so it can't be really fast; • both you and the end user have to have the interpreter to run your code.

- Python is an **interpreted language**. This means that it inherits all the described advantages and disadvantages. Of course, it adds some of its unique features to both sets.
- If you want to program in Python, you'll need the **Python interpreter**. You won't be able to run your code without it. Fortunately, **Python is free**. This is one of its most important advantages.
- Due to historical reasons, languages designed to be utilized in the interpretation manner are often called **scripting languages**, while the source programs encoded using them are called **scripts**.

What is Python?

- Python is a widely-used, interpreted, object-oriented, and high-level programming language with dynamic semantics, used for general-purpose programming.
- Python was created by **Guido van Rossum**, born in 1956 in Haarlem, the Netherlands.
- **Python goals:**

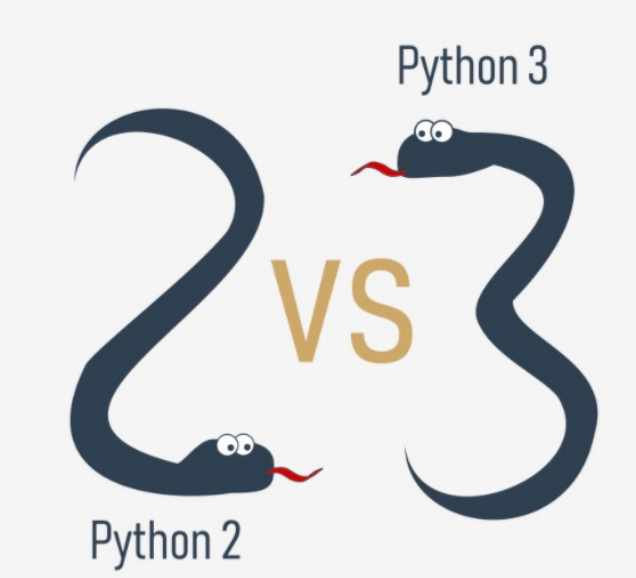
In 1999, Guido van Rossum defined his goals for Python:

- an **easy and intuitive** language just as powerful as those of the major competitors;
- **open source**, so anyone can contribute to its development;
- code that is as **understandable** as plain English;
- **suitable for everyday tasks**, allowing for short development times.

What makes Python special?

Python 2 vs. Python 3

- There are two main kinds of Python, called Python 2 and Python 3.
- These two versions of Python aren't compatible with each other. Python 2 scripts won't run in a Python 3 environment and vice versa, so if you want the old Python 2 code to be run by a Python 3 interpreter, the only possible solution is to rewrite it, not from scratch, of course, as large parts of the code may remain untouched, but you do have to revise all the code to find all possible incompatibilities. Unfortunately, this process cannot be fully automatized.



Installing Python 3

- Most of the Linux OS has Python pre-installed To check if your device is pre-installed with Python or not, just go to terminal and run the following command:

```
osboxes@osboxes:~$ python3 --version
Python 3.10.6
osboxes@osboxes:~$
```

- Installing PIP

```
$ sudo apt install python3-pip
```

Installing A Specific Version Of Python (Optional)

- Identify the Python targeted version

https://www.python.org/downloads/

- Install Python targeted version

```
$ sudo apt-get install python3.10
$ python3.10
```

Using your Python Interpreter

How to…	Command
Access the Python Interactive Shell	$ **python**
Running a Python script	$ **python** *script.py*

Python Hello World

- The print() function is a **built-in** function. It prints/outputs a specified message to the screen/console window.
- An "empty" print() function outputs an empty line to the screen.
- Python strings are delimited with **quotes**, e.g., "I am a string" (double quotes), or 'I am a string, too' (single quotes).
- The end and sep parameters can be used for formatting the output of the print() function. The sep parameter specifies the separator between the outputted arguments, whereas the end parameter specifies what to print at the end of the print statement.

```
print('Hello world')
print()
print("Hello world")
print()
print("Hello", "world!")
print("Hello", "world", sep="-->", end=";\n")
```

The Input() Function

- The print() function sends data to the console, while the input() function gets data from the console.
- The input() function comes with an optional parameter: the prompt string. It allows you to write a message before the user input

```
name = input("Enter your name: ")
print("Hello, " + name + ". Nice to meet you!")
```

Basic Data Types

Python type()	Values (examples)
int	-128, 0, 42
float	-1.12, 0, 3.14159
bool	True, False
str	“Hello” Can use ‘’, “”, and “””””””
bytes	b”Hello \xf0\x9f\x98\x8e”

```
type(3) #output: <class 'int'>
type(1.4) #output: <class 'float'>
type(True) #output: <class 'bool'>
type("Hello") #output <class 'str'>
type(b"Hello") #output <class 'bytes'>
```

Numerical Operators

Math Operations

Addition:	+
Subtraction:	-
Multiplication:	*
Division:	/
Floor Division:	//
Modulo:	%
Power:	**

```
5 + 2 #output 7
9 * 12 #output 108
13/4 #output 3.25
13 % 4 #output 1
2 ** 10 #output 1024
```

Variables

Names

- Cannot start with a number [0-9]
- Cannot conflict with a language keyword
- Can contain: [A-Za-z0-9_-]
- Recommendations for naming (variables, classes, functions, etc.) can be found in PEP8

Created with the = assignment operator

```
b = 7
c = 3
a = b + c
print(a) #output 10

a += 20
print(a) #output 30

a = "Hello Python"
print(a) #output Hello Python
```

Python is a **dynamically-typed** language, which means you don't need to *declare* variables in it. (2.1.4.3) To assign values to variables, you can use a simple assignment operator in the form of the equal (=) sign, i.e., var = 1.

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

Name convention

The PEP 8 -- Style Guide for Python Code recommends the following naming convention for variables and functions in Python:

- #### variable names should be lowercase, with words separated by underscores to improve readability (e.g., var, my_variable)

- #### function names follow the same convention as variable names (e.g., fun, my_function)

- #### it's also possible to use mixed case (e.g., myVariable), but only in contexts where that's already the prevailing style, to retain backwards compatibility with the adopted convention.

Working With Strings

Python strings are **immutable sequences** and can be indexed, sliced, and iterated like any other sequence, as well as being subject to the in and not in operators. There are two kinds of strings in Python:

Oneline String

```
oneline = "I Am A String"
print(oneline)
```

Multi-Line String

```
multiline = '''Line #1
Line #2'''

print(multiline)
```

Working With Strings

The len() function used for strings returns a number of characters contained by the arguments. The escape character (\) is not counted

```
# Example 1

word = 'by'
print(len(word)) #output 2


# Example 2

empty = ''
print(len(empty)) #output 0


# Example 3

i_am = 'I\'m'
print(len(i_am)) #output 3
```

Working With Strings

String Operations

Concatenation: **+**

Multiplication: *****

Some Useful String Methods

Composition: **“{}”.format()**

Splitting: **“”.split()**

Joining: **“”.join()**

```
print('one' + "two") #output: onetwo
print("Abc" * 3) #output: AbcAbcAbc

octets = "192.168.1.1".split(".")
print(octets) #output: ['192', '168', '1', '1']
ip_address=".".join(octets)
print(ip_address) #output 192.168.1.1

str = "Default Gateway IP Address Is {}".format(ip_address)
print(str) #output: Default Gateway IP Address Is 192.168.1.1

ip_address = "192.168.1.1"
str2 = f"Default Gateway IP Address Is {ip_address}"
print(str2) #output: Default Gateway IP Address Is 192.168.1.1
```

String Escape Characters

```
txt = "We are the so-called \"Vikings\" from the north."
print(txt) #output We are the so-called "Vikings" from the north.
print("\"I\'m\"\n\"\"learning\"\"\n\"\"\"Python\"\"\"")
#output
#   "I'm"
#   ""learning""
#   """Python"""
```

Conditionals

Syntax:

```
if expression1:
   statements…
elif expression2:
   statements…
else:
   statements…
```

- ✓ Indentation is important!
- ✓ 4 spaces indent recommended
- ✓ You can nest if statements

Comparison Operators:

Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
Equal	==
Not Equal	!=
Contains element	in

Combine expressions with: **and**, **or**

Negate with: **not**

Loops

There are two types of loops in Python: while and for

```
# Example 1
word = "Python"
for letter in word:
    print(letter, end="*")

# Example 2
for i in range(1, 10):
    if i % 2 == 0:
        print(i)
```

```
# Example 1
while True:
    print("Stuck in an infinite loop.")

# Example 2
counter = 5
while counter > 2:
    print(counter)
    counter -= 1
```

The for loop executes a set of statements many times; it's used to iterate over a sequence (e.g., a list, a dictionary, a tuple, or a set - you will learn about them soon) or other objects that are iterable (e.g., strings). You can use the for loop to iterate over a sequence of numbers using the built-in range function.

The while loop executes a statement or a set of statements as long as a specified boolean condition is true

Range() Function

The range() function generates a sequence of numbers. It accepts integers and returns range objects. The syntax of range() looks as follows: range(start, stop, step), where:

- start is an optional parameter specifying the starting number of the sequence (0 by default)
- stop is an optional parameter specifying the end of the sequence generated (it is not included),
- and step is an optional parameter specifying the difference between the numbers in the sequence (1 by default.)

```
for i in range(3):
    print(i, end=" ")   # Outputs: 0 1 2

for i in range(6, 1, -2):
    print(i, end=" ")   # Outputs: 6, 4, 2
```

Lists

The list is a type of data in Python used to store multiple objects. It is an ordered and mutable collection of comma-separated items between square brackets

```
my_list = [1, None, True, "I am a string", 256, 0]
```

Lists can be **nested**

```
my_list = [1, 'a', ["list", 64, [0, 1], False]]
```

Lists

Lists can be **indexed and updated**

```
my_list = [1, None, True, 'I am a string', 256, 0]
print(my_list[3])  # outputs: I am a string
print(my_list[-1])  # outputs: 0

my_list[1] = '?'
print(my_list)  # outputs: [1, '?', True, 'I am a string', 256, 0]

my_list.insert(0, "first")
my_list.append("last")
print(my_list)  # outputs: ['first', 1, '?', True, 'I am a string', 256, 0
```

List elements and lists can be **deleted**

```
my_list = [1, 2, 3, 4]
del my_list[2]
print(my_list)  # outputs: [1, 2, 4]

del my_list  # deletes the whole list
```

Lists

Lists can be **iterated** through using the for loop,

```
my_list = ["white", "purple", "blue", "yellow", "green"]

for color in my_list:
    print(color)
```

The len() function may be used to **check the list's length**

```
my_list = ["white", "purple", "blue", "yellow", "green"]
print(len(my_list))  # outputs 5

del my_list[2]
print(len(my_list))  # outputs 4
```

List Slicing

If you want to copy a list or part of the list, you can do it by performing **slicing**:

```
colors = ['red', 'green', 'orange']

copy_whole_colors = colors[:]  # copy the entire list
copy_part_colors = colors[0:2]  # copy part of the list
```

You can use **negative indices** to perform slices, too.

```
sample_list = ["A", "B", "C", "D", "E"]
new_list = sample_list[2:-1]
print(new_list)  # outputs: ['C', 'D']
```

List Slicing

The start and end parameters are **optional** when performing a slice: list[start:end]

```
my_list = [1, 2, 3, 4, 5]
slice_one = my_list[2: ]
slice_two = my_list[ :2]
slice_three = my_list[-2: ]

print(slice_one)   # outputs: [3, 4, 5]
print(slice_two)   # outputs: [1, 2]
print(slice_three)   # outputs: [4, 5]
```

You can **delete slices** using the del instruction

```
my_list = [1, 2, 3, 4, 5]
del my_list[0:2]
print(my_list)   # outputs: [3, 4, 5]

del my_list[:]
print(my_list)   # deletes the list content, outputs: []
```

You can test if some items **exist in a list or not** using the keywords in and not in

```
my_list = ["A", "B", 1, 2]

print("A" in my_list)   # outputs: True
print("C" not in my_list)   # outputs: True
print(2 not in my_list)   # outputs: False
```

The Inner Life Of Lists

If you have a list l1, then the following assignment: l2 = l1 does not make a copy of the l1 list, but makes the variables l1 and l2 **point to one and the same list in memory**

```
vehicles_one = ['car', 'bicycle', 'motor']
print(vehicles_one) # outputs: ['car', 'bicycle', 'motor']

vehicles_two = vehicles_one
del vehicles_one[0] # deletes 'car'
print(vehicles_two) # outputs: ['bicycle', 'motor']
```

List Comprehension

List comprehension allows you to create new lists from existing ones in a concise and elegant way.

The syntax of a list comprehension looks as follows:

```
[expression for element in list if conditional]
```

```
lst = [i ** i for i in range (1,10)]
print(lst) #Output: [1, 4, 27, 256, 3125, 46656, 823543, 16777216, 387420489]
lst = [i for i in range(101) if i % 2 == 0]
print(lst) #Output: [0, 2, 4, 6, 8, 10.. 100]
```

Dictionaries

Dictionaries are unordered, changeable (mutable), and indexed collections of data

Each dictionary is a set of *key: value* pairs. You can create it by using the following syntax

```
my_dictionary = {
    key1: value1,
    key2: value2,
    key3: value3
}
```

Accessing A Dictionary Items

If you want to access a dictionary item, you can do so by making a reference to its key inside a pair of square brackets (ex. 1) or by using the get() method (ex. 2):

```
pol_eng_dictionary = {
    "kwiat": "flower",
    "woda": "water",
    "gleba": "soil"
    }

item_1 = pol_eng_dictionary["gleba"]    # ex. 1
print(item_1)    # outputs: soil

item_2 = pol_eng_dictionary.get("woda")
print(item_2)    # outputs: water
```

Updating A Dictionary Items

If you want to change the value associated with a specific key, you can do so by referring to the item's key name in the following way:

```
pol_eng_dictionary = {
    "zamek": "castle",
    "woda": "water",
    "gleba": "soil"
    }

pol_eng_dictionary["zamek"] = "lock"
item = pol_eng_dictionary["zamek"]
print(item)  # outputs: lock
```

To add or remove a key (and the associated value), use the following syntax:

```
phonebook = {}     # an empty dictionary

phonebook["Adam"] = 3456783958    # create/add a key-value pair
print(phonebook)    # outputs: {'Adam': 3456783958}

del phonebook["Adam"]
print(phonebook)    # outputs: {}
```

Updating A Dictionary

You can also insert an item to a dictionary by using the update() method, and remove the last element by using the popitem() method, e.g:

```
pol_eng_dictionary = {"kwiat": "flower"}

pol_eng_dictionary.update({"gleba": "soil"})
print(pol_eng_dictionary)    # outputs: {'kwiat': 'flower', 'gleba': 'soil'}

pol_eng_dictionary.popitem()
print(pol_eng_dictionary)    # outputs: {'kwiat': 'flower'}
```

You can use the del keyword to remove a specific item, or delete a dictionary. To remove all the dictionary's items, you need to use the clear() method:

```
pol_eng_dictionary = {
    "zamek": "castle",
    "woda": "water",
    "gleba": "soil"
    }

print(len(pol_eng_dictionary))    # outputs: 3
del pol_eng_dictionary["zamek"]    # remove an item
print(len(pol_eng_dictionary))    # outputs: 2

pol_eng_dictionary.clear()   # removes all the items
print(len(pol_eng_dictionary))    # outputs: 0

del pol_eng_dictionary    # removes the dictionary
```

Looping Through A Dictionary

You can use the for loop to loop through a dictionary

```
pol_eng_dictionary = {
    "zamek": "castle",
    "woda": "water",
    "gleba": "soil"
    }

for item in pol_eng_dictionary:
    print(item)

# outputs: zamek
#          woda
#          gleba
```

If you want to loop through a dictionary's keys and values, you can use the items() method, e.g.:

```
pol_eng_dictionary = {
    "zamek": "castle",
    "woda": "water",
    "gleba": "soil"
    }

for key, value in pol_eng_dictionary.items():
    print("Pol/Eng ->", key, ":", value)
```

Looping Through A Dictionary

To check if a given key exists in a dictionary, you can use the in keyword

```
pol_eng_dictionary = {
    "zamek": "castle",
    "woda": "water",
    "gleba": "soil"
    }

if "zamek" in pol_eng_dictionary:
    print("Yes")
else:
    print("No")
```

Copying A Dictionary

To copy a dictionary, use the copy() method:

```
pol_eng_dictionary = {
    "zamek": "castle",
    "woda": "water",
    "gleba": "soil"
    }

copy_dictionary = pol_eng_dictionary.copy()
```

Functions

A **function** is a block of code that performs a specific task when the function is called (invoked). You can use functions to make your code reusable, better organized, and more readable. Functions can have parameters and return values.

You can define your own function using the def keyword and the following syntax:

```
def your_function(optional_parameters):
    # the body of the function
```

You can define a function which doesn't take any arguments

```
def message():    # defining a function
    print("Hello")    # body of the function

message()    # calling the function
```

You can define a function which takes arguments

```
def hello(name):    # defining a function
    print("Hello,", name)    # body of the function


name = input("Enter your name: ")

hello(name)    # calling the function
```

Passing Arguments To A Function

You can pass information to functions by using parameters

```
def hi(name):
    print("Hi,", name)

hi("Greg") #output Hi, Greg
```

You can use the keyword argument passing technique to **pre-define** a value for a given argument

```
def name(first_name, last_name="Smith"):
    print(first_name, last_name)

name("Andy")    # outputs: Andy Smith
name("Betty", "Johnson")
# outputs: Betty Johnson (the keyword argument replaced by "Johnson")
```

Passing Arguments To A Function

You can pass arguments to a function using the following techniques:

- **positional argument passing** in which the order of arguments passed matters (Ex. 1),
- **keyword (named) argument passing** in which the order of arguments passed doesn't matter (Ex. 2),
- a mix of positional and keyword argument passing (Ex. 3).

```
#Ex. 1
def subtra(a, b):
    print(a - b)

subtra(5, 2)    # outputs: 3
subtra(2, 5)    # outputs: -3
```

```
#Ex. 2
def subtra(a, b):
    print(a - b)

subtra(a=5, b=2)    # outputs: 3
subtra(b=2, a=5)    # outputs: 3
```

```
#Ex. 3
def subtra(a, b):
    print(a - b)

subtra(5, b=2)    # outputs: 3
subtra(5, 2)    # outputs: 3
```

Returning A Result From A Function

Return without an expression

- It causes the immediate termination of the function's execution

Return with an expression

- it causes the immediate termination of the function's execution
- moreover, the function will evaluate the expression's value and will return (hence the name once again) it as the function's result.

```
def square(a):
    print('Before return instruction')
    return a * a
    print('After return instruction')

x = square(10) #output: Before return instruction
print(x) #output 100
```

Importing and Using Packages & Modules

- Module is a file containing Python definitions and statements, which can be later imported and used when necessary.
- Syntax

Import module
From module import thing
Import module as alias_name

```
In [2]: import math, sys

In [3]: print(math.pi)
        3.141592653589793

In [4]: print(math.sin(math.pi/2))
        1.0

In [5]: from math import pi
        print(pi)
        3.141592653589793

In [6]: from math import sin, pi
        print(sin(pi/2))
        1.0

In [20]: import math as m
         print(m.sin(m.pi/2))
         1.0

In [9]: from math import pi as PI, sin as sine

        print(sine(PI/2))
        1.0
```

Importing and Using Packages & Modules

- Import user module in same directory

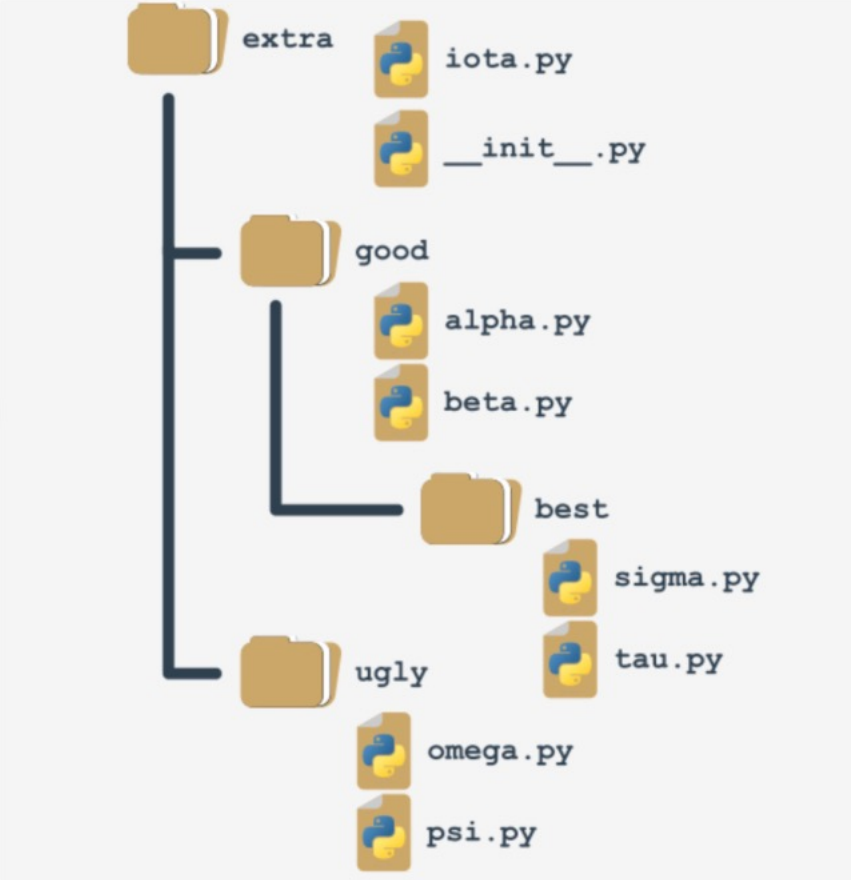
```
import module1 as m1
```

- Import user module from different directories

```
import os
from sys import path
module_path = "path to module directory"
path.append(module_path)
import module2 as m2
```

Packages

The presence of the init.py file makes up the package




```
import os
from sys import path

package_path = os.getcwd() + "/extra"
#or can use .zip as module
#package_path = os.getcwd() + "extrapack.zip"
path.append(package_path)
import extra.good.best.sigma as sig
import extra.good.alpha as alp
```

Python Package Installer (PIP)

- Uses the open PyPI repository
- Installs packages and their dependencies
- You can post your packages to PyPI

```
$ sudo apt install python3-pip

$ pip --version
pip 20.3.4 from /usr/lib/python3/dist-packages/pip (python 3.9)

$ pip3 --version
pip 20.3.4 from /usr/lib/python3/dist-packages/pip (python 3.9)
```

```
$ pip3 help
$ pip3 list
$ pip3 show package_name
$ pip3 search anystring
$ pip3 install package_name
$ pip3 uninstall package_name
```

Processing files

A file needs to open before it can be processed by a program, and it should be closed when the processing is finished.

Open a file

```
f = open("demofile.txt", "r")
```

Close a file

```
f.close()
```

Open & close files by with...as

```
with open("demofile.txt", "r") as data:
    print(data.read())
```

Read Files

```
print(f.read())
print(f.readline())
data = f.readlines();
for line in data:
    print(line)
```

Processing files

Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

```
f = open("demofile.txt", "a")
f.write("\nFourth line added by Python")
f.close()
```

```
f = open("demofile.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()
```

Create a New File

To create a new file in Python, use the open() method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

```
f = open("myfile.txt", "x")
```

```
f = open("myfile.txt", "w")
```

NETMIKO

Netmiko is a multi-vendor library to simplify CLI connections to network device

https://github.com/ktbyers/netmiko

Installation

```
pip3 install netmiko
```

NETMIKO

Establish an SSH connection to the device by passing in the device dictionary.

```
from netmiko import ConnectHandler

iosxe = {
    'device_type': 'cisco_ios',
    'ip':   'sandbox-iosxe-latest-1.cisco.com',
    'username': 'developer',
    'password': 'C1sco12345',
    'port' : 22,          # optional, defaults to 22
}

connection = ConnectHandler(**iosxe)
```

NETMIKO

Execute show commands.

```
output = connection.send_command('show ip int brief')
print(output)

"""output
Interface              IP-Address      OK? Method Status                Protocol
GigabitEthernet1       10.10.20.48     YES NVRAM  up                    up
GigabitEthernet2       unassigned      YES NVRAM  administratively down down
GigabitEthernet3       unassigned      YES NVRAM  administratively down down
"""
```

NETMIKO

Execute configuration change commands (will automatically enter into config mode

```
config2 = ['router ospf 1001', 'router-id 1.1.1.1']
result = connection.send_config_set(config2)
print(result)

"""output
configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
ROUTER-1(config)#router ospf 1001
ROUTER-1(config-router)#router-id 1.1.1.1
ROUTER-1(config-router)#end
ROUTER-1#
"""
```


TRAINOCATE