



**SECURITY FLAME**

# Python For Beginners

Learn Python Programming

With Yousuf Alhajri

**Cyber Security  
IT Services**

# Whoami

- ❖ Yousuf Alhajri ([@D4rkness\\_14](https://twitter.com/D4rkness_14))
  - A cyber security enthusiast and a programmer
  - Certifications
    - ✓ **OSCP** (*Offensive Security Certified Professional*)
    - ✓ **OSWE** (*Offensive Security Web Expert*)
    - ✓ **OSEP** (*Offensive Security Experienced Penetration Tester*)
    - ✓ **OSED** (*Offensive Security Exploit Developer*)
    - ✓ **OSCE3** (*Offensive Security Certified Expert 3*)



SECURITY FLAME

Before we start ...

Slides can be found on following Gitlab repository

<https://gitlab.com/omr00t/python-for-beginners/>



SECURITY FLAME

# Let's dive in!



SECURITY FLAME

# What is Python?

- Python is an interpreted high-level programming language.
- It is written in C.
- It's designed with code readability in mind.



SECURITY FLAME

# A bit of history

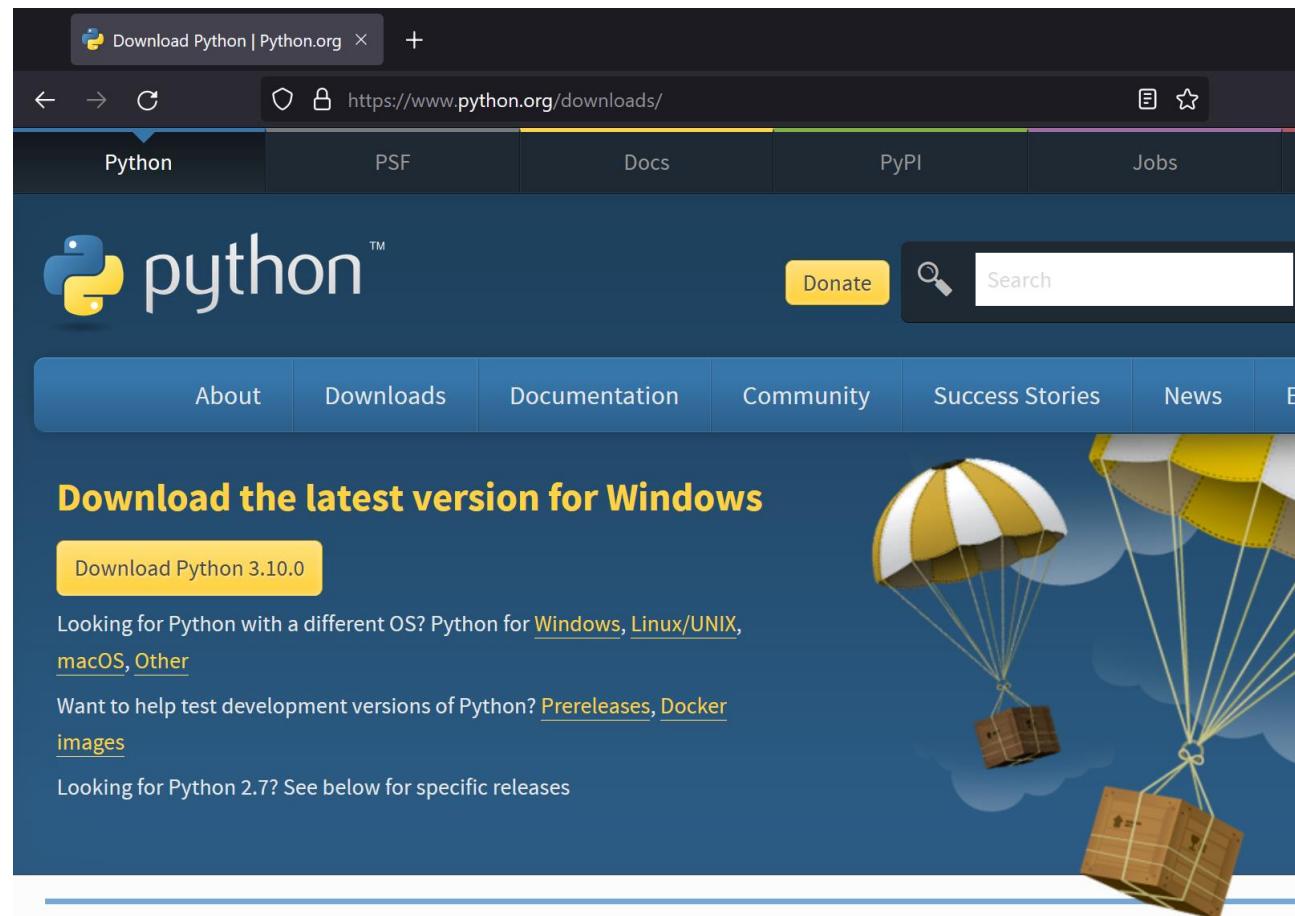
- First released in 1991 as Python 0.9.0.
- Designed by a guy named Guido Van Rossum in Netherlands.
- It was supposed to be the successor of a programming language called ABC.
- Named after BBC show "Monty Python's Flying Circus".
- Python 2.0 was released in 2000.
- Python 3.0 was released in 2008 due to fundamental flaws in previous versions.
- Python 2 was discontinued as of 2020 with latest version being 2.7.18.



SECURITY FLAME

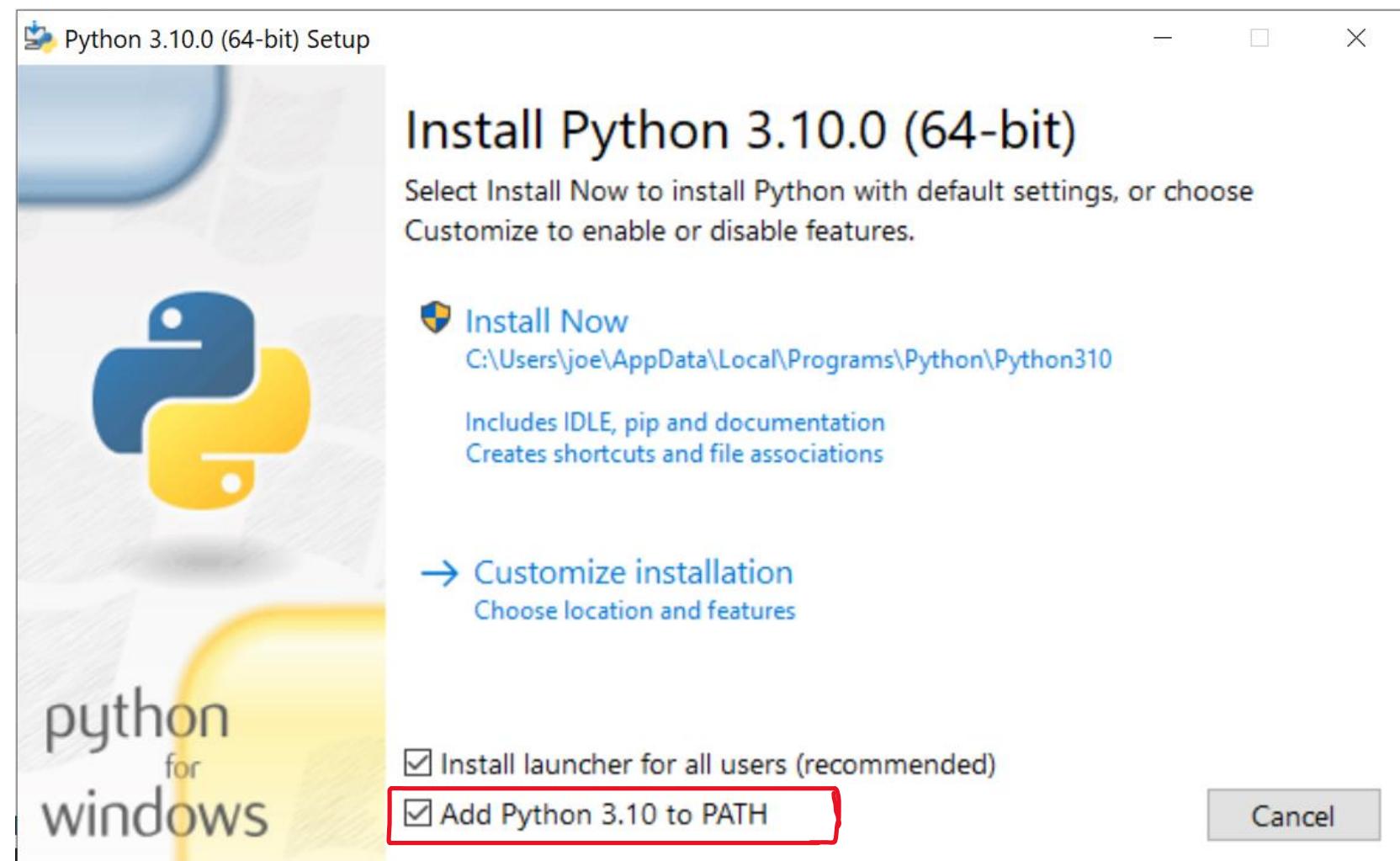
# Setting up the environment for Python

- Head to [python.org](https://www.python.org/downloads/) & click on "Download Python".





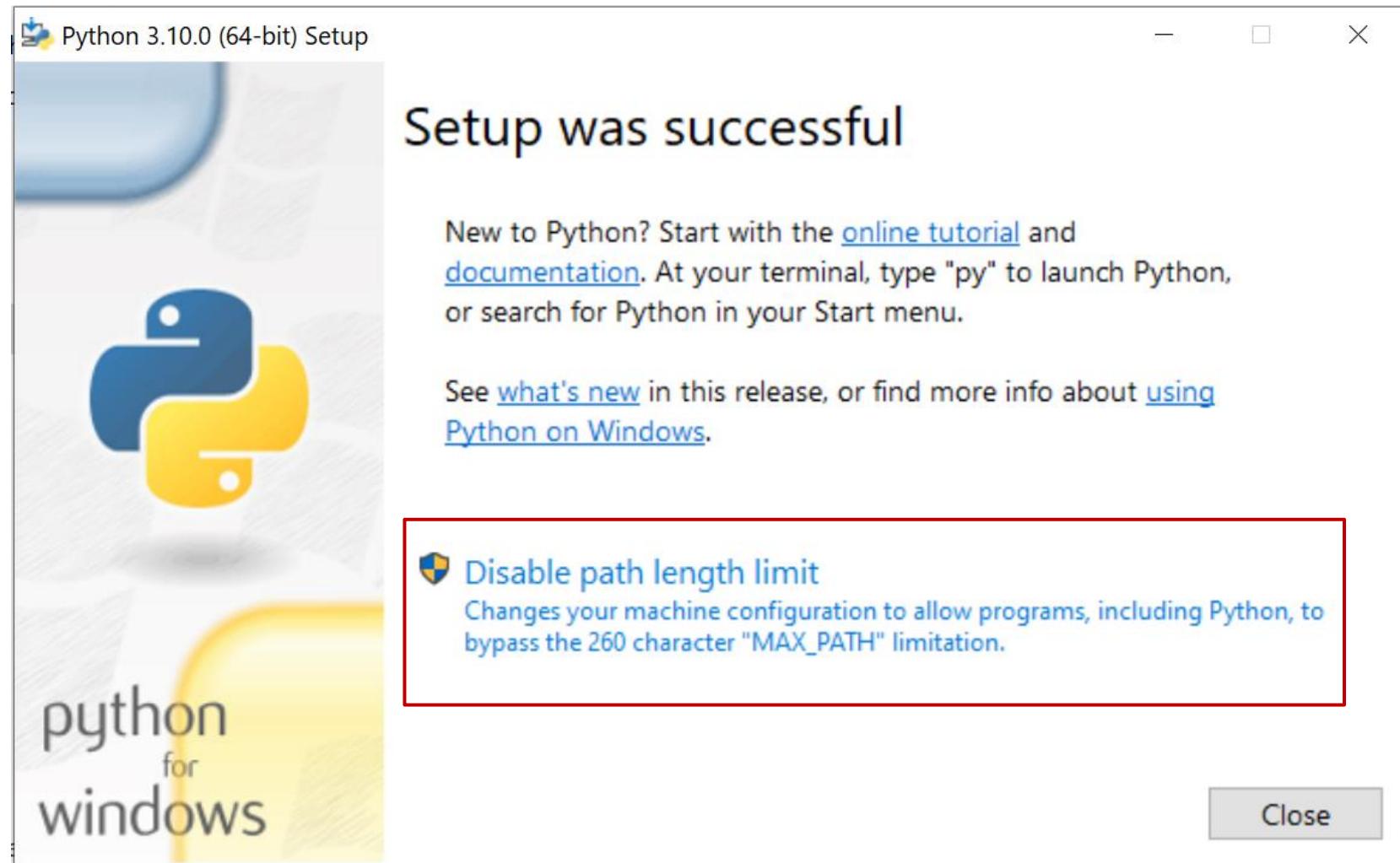
- Open the installation file, make sure you click on the checkbox "Add Python 3.10 to PATH" before clicking on "Install Now".





SECURITY FLAME

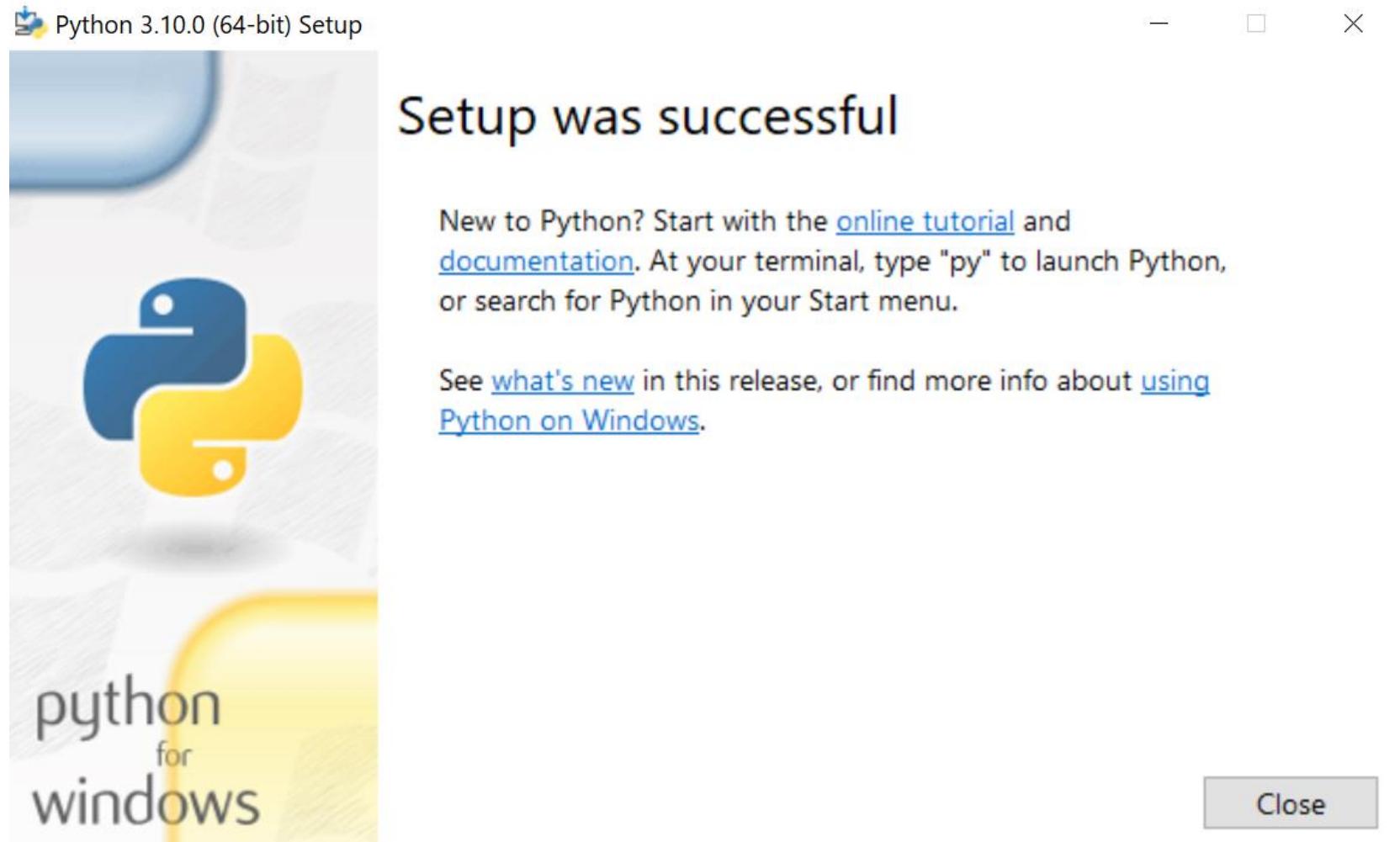
- Click on the "Disable path length limit".





SECURITY FLAME

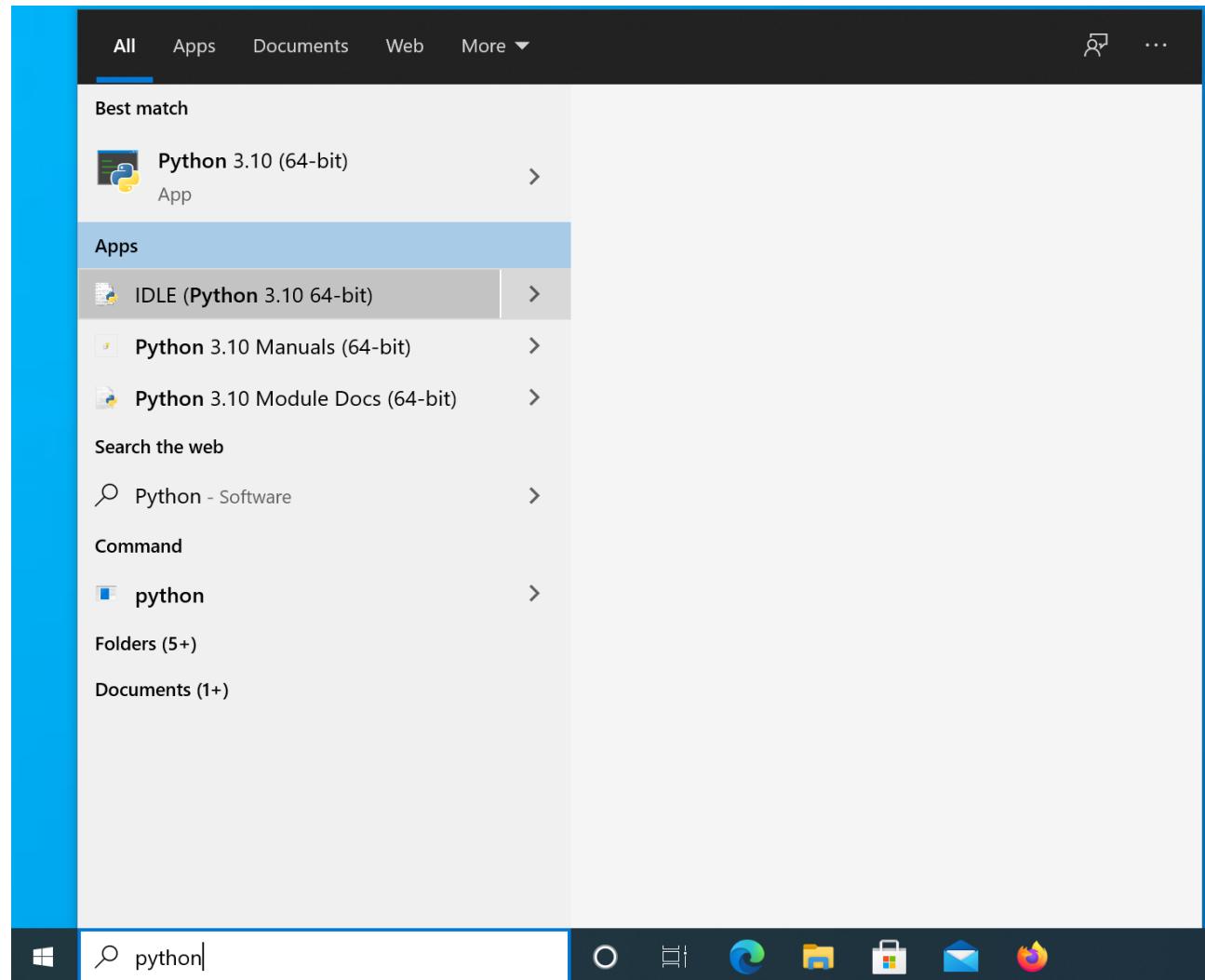
- Python has been successfully installed.



Close



- Launching Python Integrated Development Environment (IDLE) to verify successful installation.





SECURITY FLAME

➤ IDLE is ready!

IDLE Shell 3.10.0

File Edit Shell Debug Options Window Help

```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> |
```

Ln: 3 Col: 0



SECURITY FLAME

➤ First Python program on IDLE

The screenshot shows the IDLE Shell 3.10.0 interface. The title bar reads "IDLE Shell 3.10.0". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the Python version information and a simple print statement:

```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hello!")
Hello!
>>>
```



SECURITY FLAME

# Installing VSCode

- In this section, we will install VSCode.
- Go to [code.visualstudio.com/download](https://code.visualstudio.com/download)

## Download Visual Studio Code

Free and built on open source. Integrated Git, debugging and extensions.



⬇ Windows  
Windows 7, 8, 10, 11

User Installer  
System Installer  
.zip



⬇ .deb  
Debian, Ubuntu  
.rpm  
Red Hat, Fedora, SUSE

.deb  
.rpm  
.tar.gz



⬇ Mac  
macOS 10.11+  
.zip  
Universal  
Intel Chip  
Apple Silicon

Snap Store



SECURITY FLAME

- Launch the downloaded file

Setup - Microsoft Visual Studio Code (User)

### License Agreement

Please read the following important information before continuing.

Please read the following License Agreement. You must accept the terms of this agreement before continuing with the installation.

*This license applies to the Visual Studio Code product. Source Code for Visual Studio Code is available at <https://github.com/Microsoft/vscode> under the MIT license agreement at <https://github.com/microsoft/vscode/blob/master/LICENSE.txt>. Additional license information can be found in our FAQ at <https://code.visualstudio.com/docs/supporting/faq>.*

## MICROSOFT SOFTWARE LICENSE TERMS

## MICROSOFT VISUAL STUDIO CODE

I accept the agreement  
 I do not accept the agreement

Next > Cancel



SECURITY FLAME

## ➤ Selecting options

Setup - Microsoft Visual Studio Code (User)

### Select Additional Tasks

Which additional tasks should be performed?

Select the additional tasks you would like Setup to perform while installing Visual Studio Code, then click Next.

Additional icons:

Create a desktop icon

Other:

Add "Open with Code" action to Windows Explorer file context menu

Add "Open with Code" action to Windows Explorer directory context menu

Register Code as an editor for supported file types

Add to PATH (requires shell restart)

< Back    Next >    Cancel





SECURITY FLAME

## ➤ Starting VSCode

The screenshot shows the 'Get Started' screen in Visual Studio Code. The interface is dark-themed. On the left, there's a vertical sidebar with icons for file operations, search, and other settings. The main area has a title 'Get Started with VS Code' with a lightning bolt icon. Below it, a sub-section titled 'Choose the look you want' discusses color palettes and includes a 'Browse Color Themes' button and a keyboard tip. To the right of this section are three theme preview cards: 'Light' (light gray background with colored lines), 'Dark' (dark gray background with colored lines, highlighted with a blue border), and 'High Contrast' (dark background with colored lines). At the bottom, there are five more bullet points: 'Sync to and from other devices', 'One shortcut to access everything', 'Rich support for all your languages', and 'Open up your code'. At the very bottom, there are links for 'Mark Done' and 'Next Section →', and a note about usage data collection.

Get Started - Visual Studio Code

Get Started X

Get Started

Get Started with VS Code

Discover the best customizations to make VS Code yours.

Choose the look you want

The right color palette helps you focus on your code, is easy on your eyes, and is simply more fun to use.

Browse Color Themes

Tip: Use keyboard shortcut (Ctrl+K Ctrl+T)

Sync to and from other devices

One shortcut to access everything

Rich support for all your languages

Open up your code

Mark Done

Next Section →

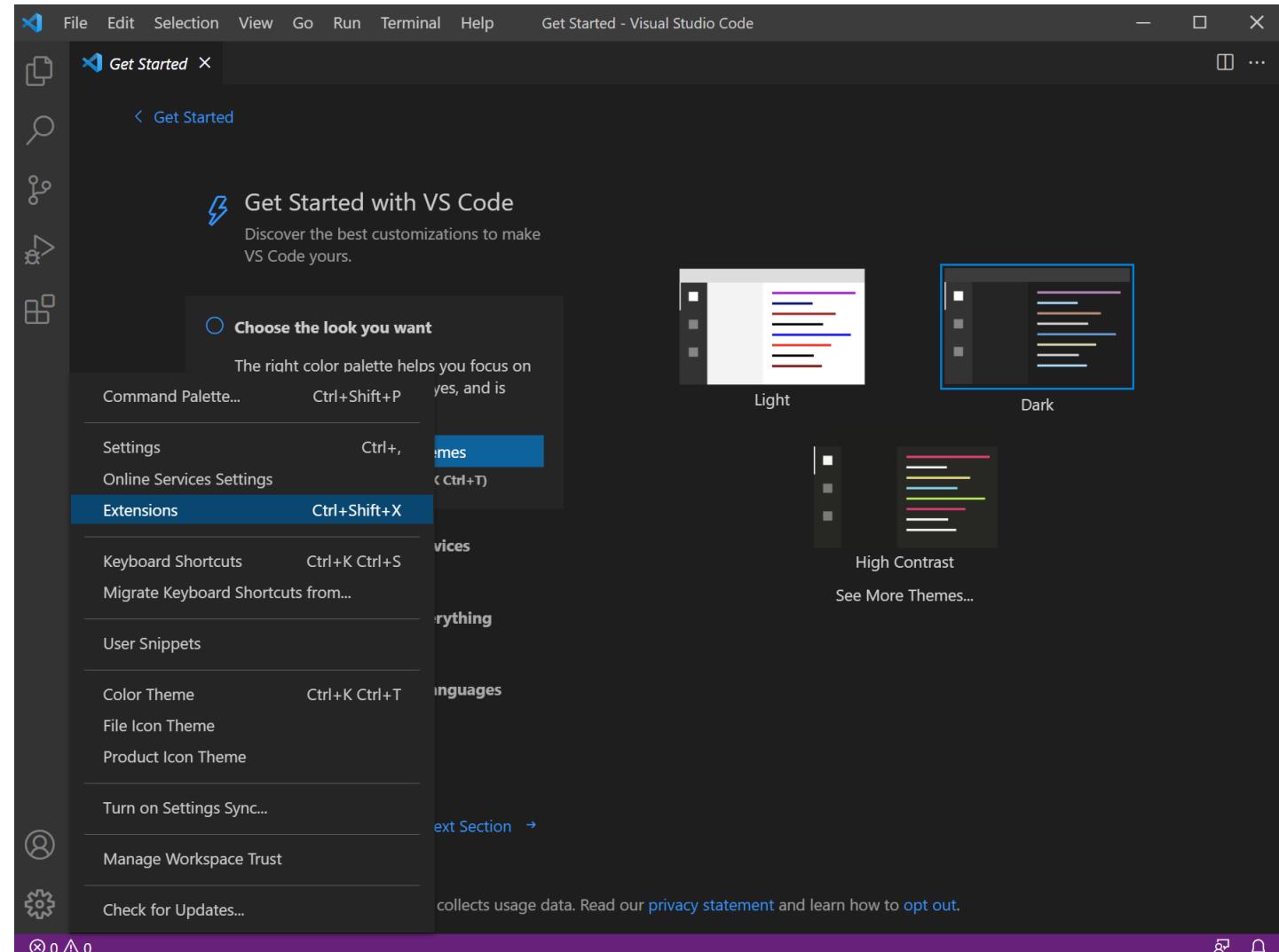
Code collects usage data. Read our [privacy statement](#) and learn how to [opt out](#).



SECURITY FLAME

# Configuring VSCode

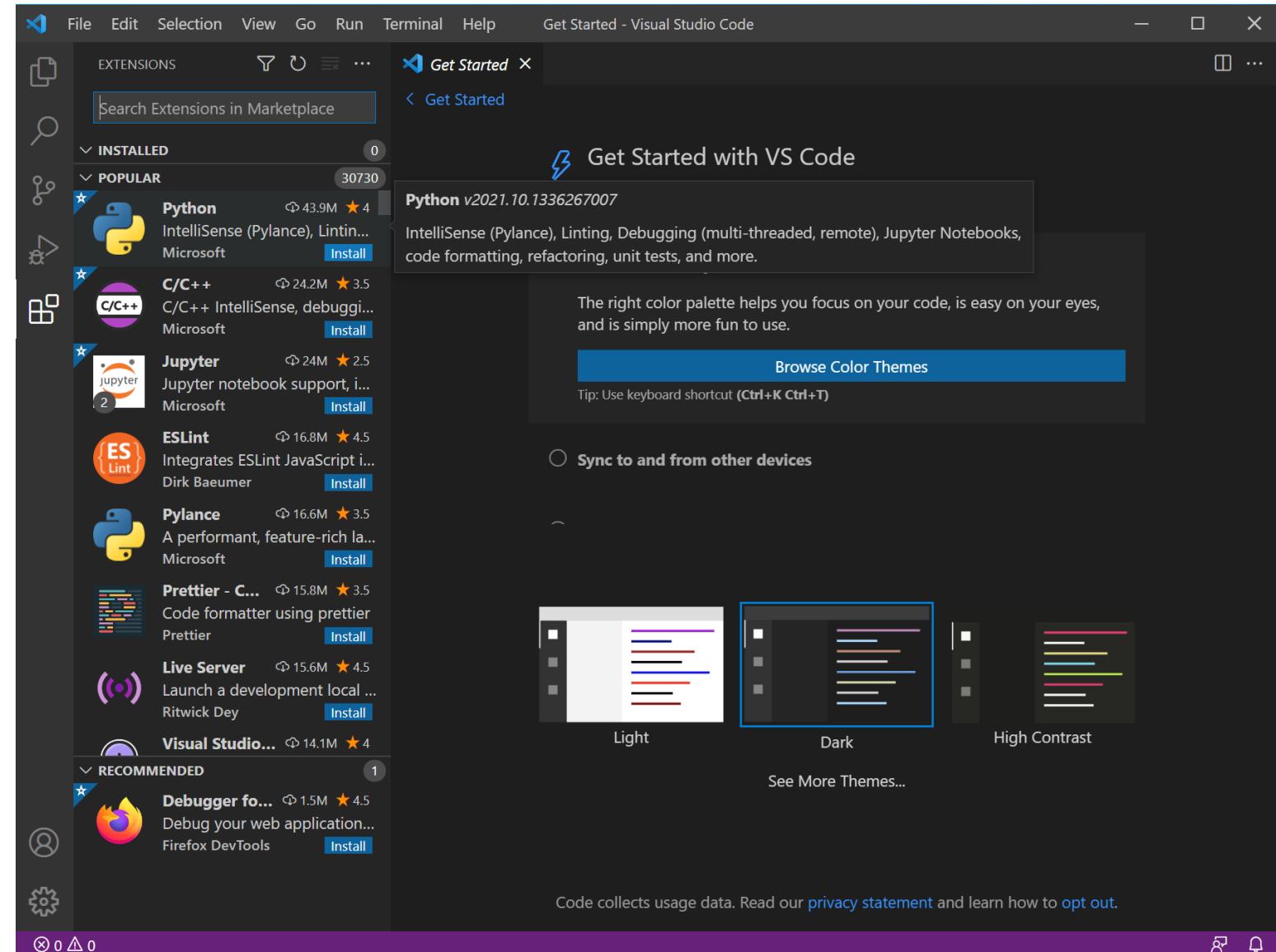
- Open the extension window





SECURITY FLAME

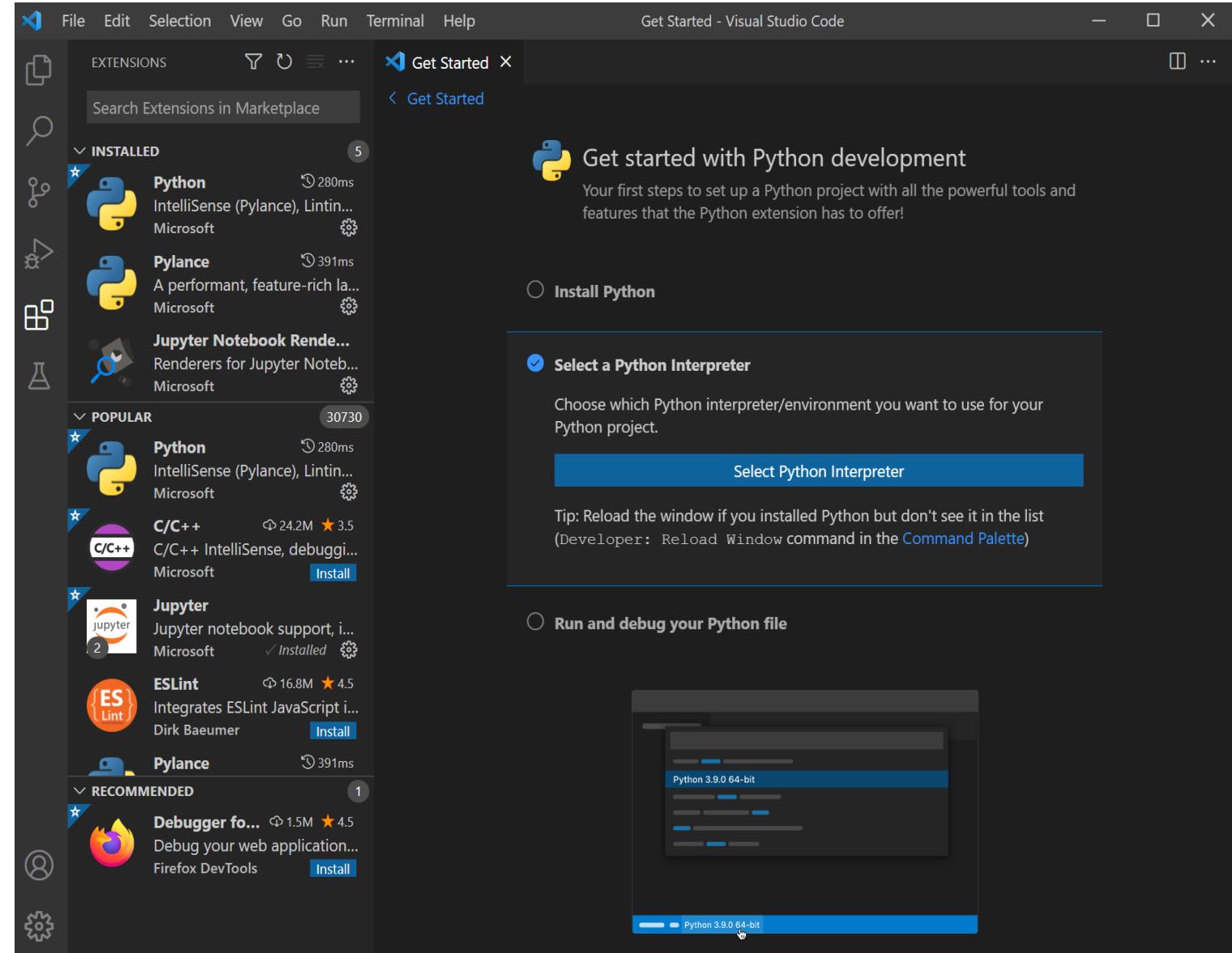
- Select the Python extension and click on "install"





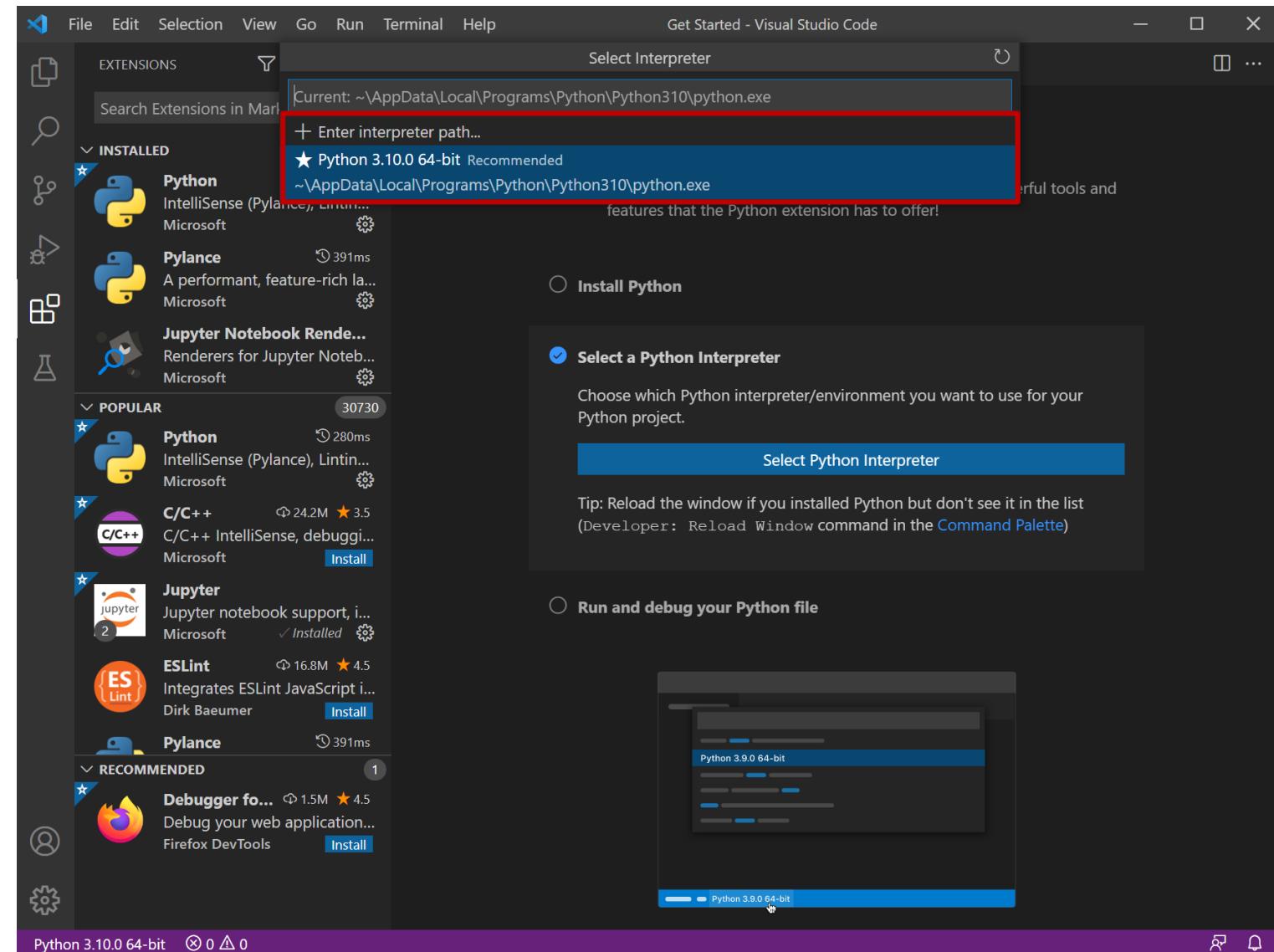
SECURITY FLAME

- Once installed, click on the installed extension and click on "Select Python Interpreter".





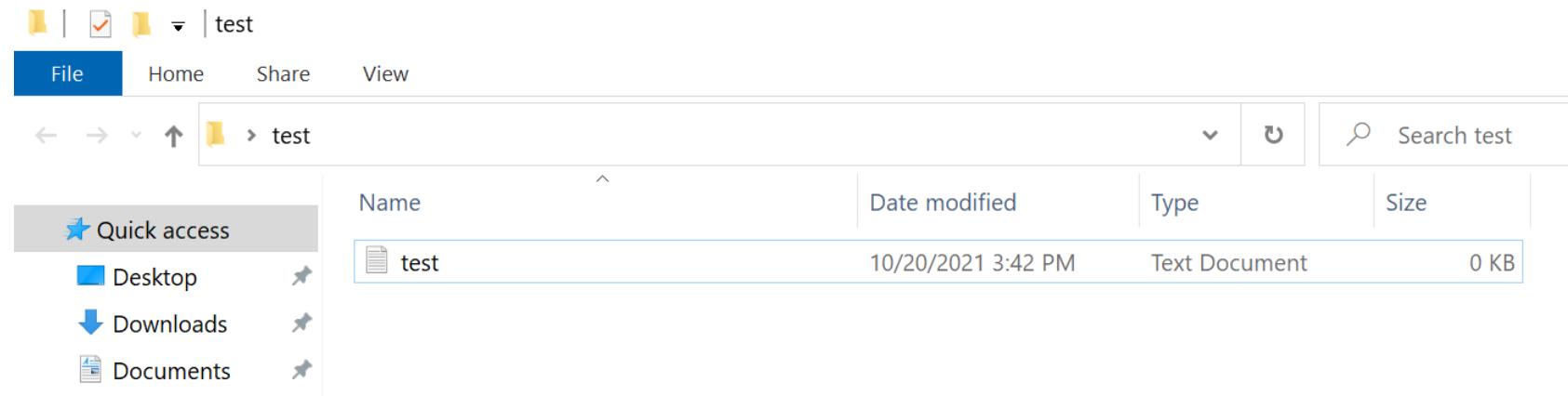
- Select Python interpreter location on your computer





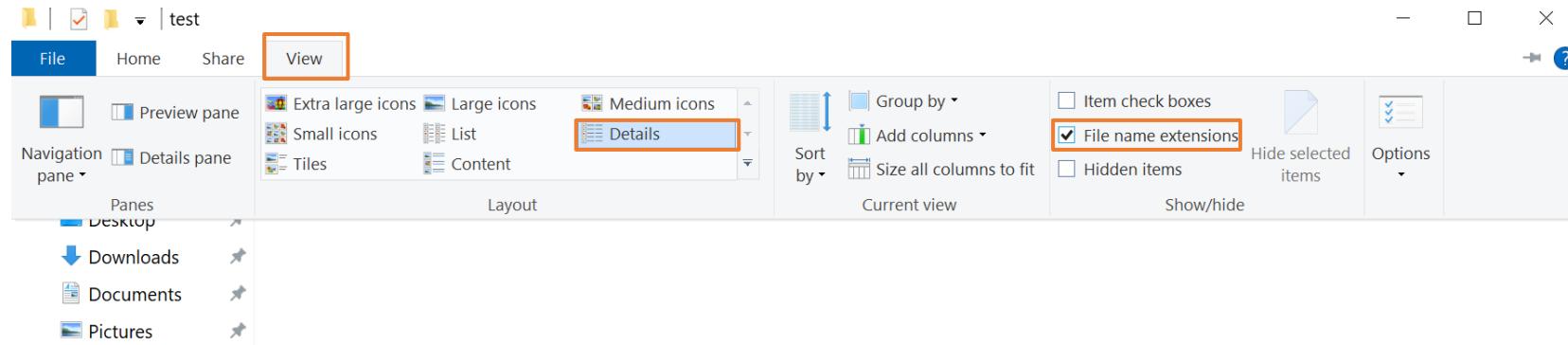
SECURITY FLAME

- Once done, we can create a new folder called "test" and create a new text file that is also called "test"



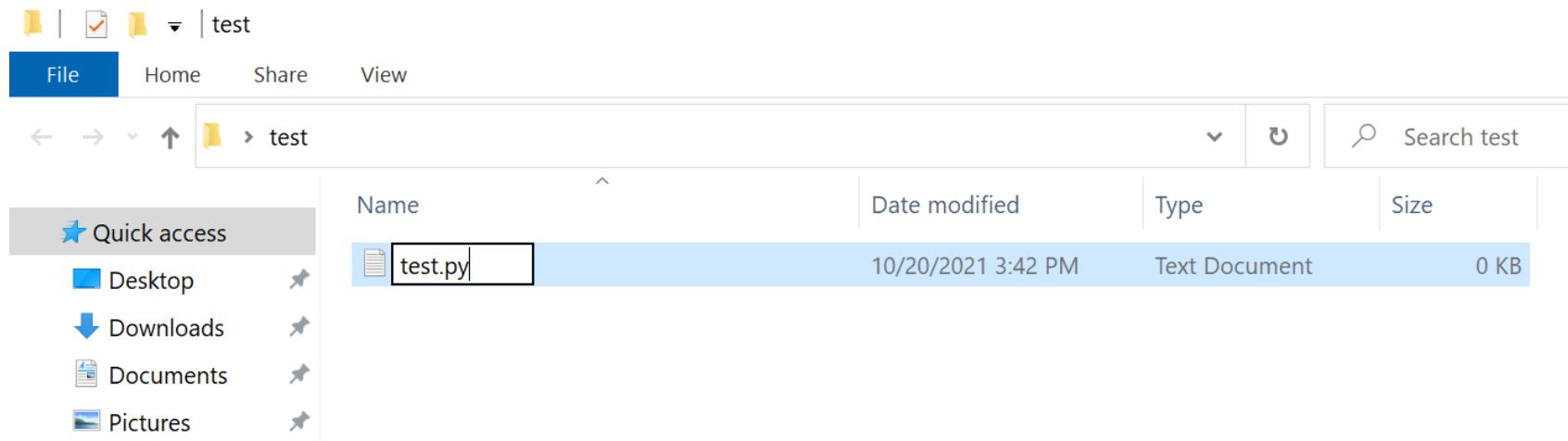


- Next, click on "View" and check the "File name extensions" checkbox



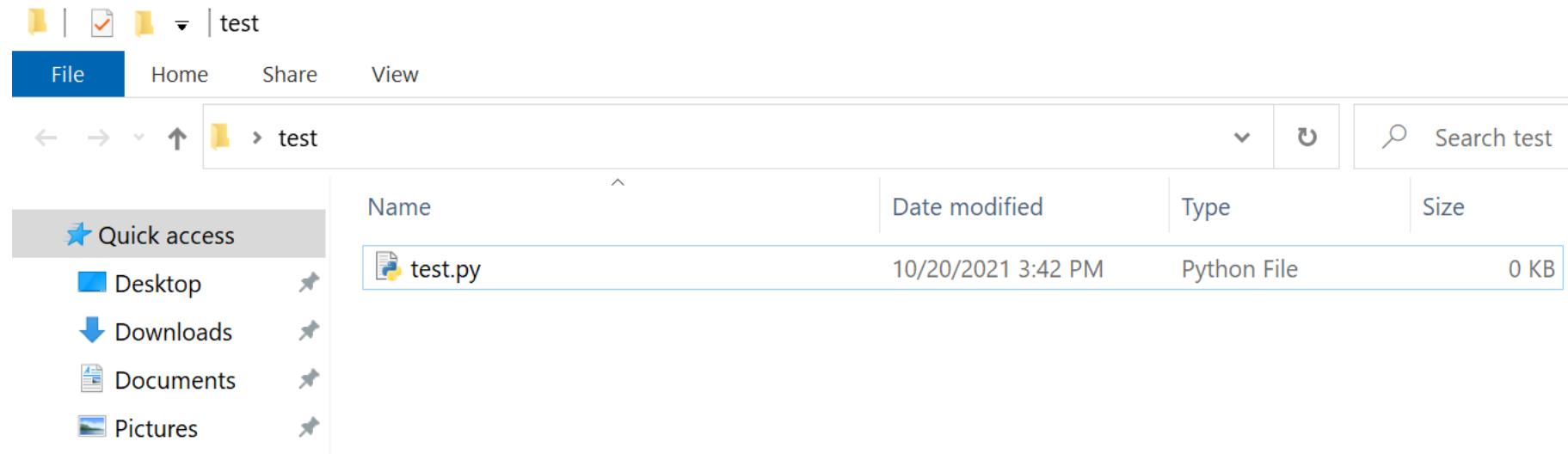


- Rename the test file to “test.py”





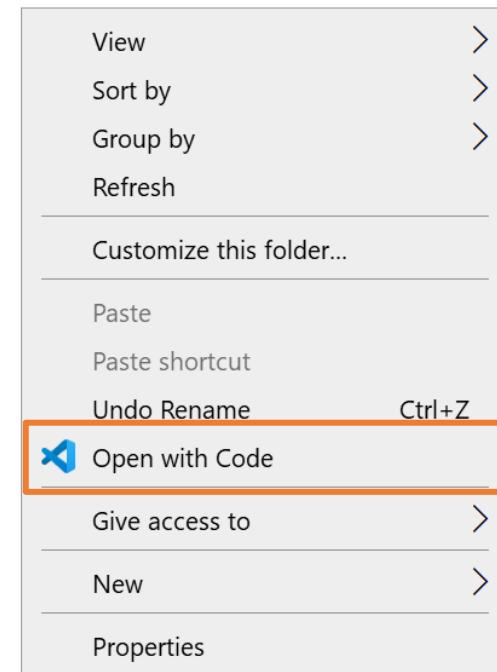
- It will look like a Python file once renamed





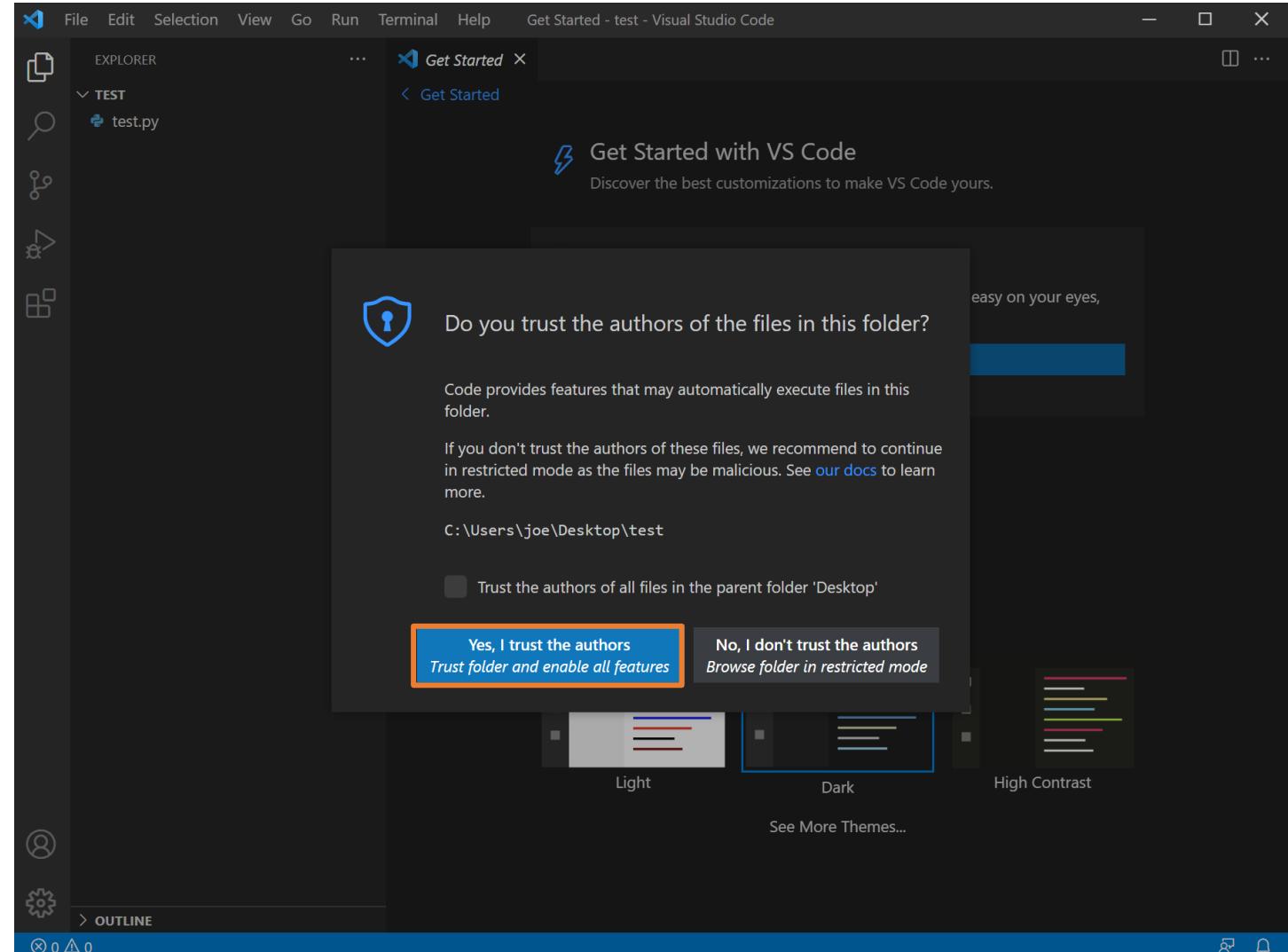
- Right click on the folder and click on "Open with Code"

Name	Date modified	Type	Size
test.py	10/20/2021 3:42 PM	Python File	0 KB





- Since we created the folder ourselves, you can click on "Yes, I trust the authors"





- Next, we can write the following statement on the “test.py” file:

```
print("Hello!")
```

A screenshot of the Visual Studio Code interface. The window title is "test.py - test - Visual Studio Code". The left sidebar shows an "EXPLORER" view with a folder named "TEST" containing a file "test.py". The main editor area displays the following Python code:

```
1 print("Hello!")
```

The status bar at the bottom indicates "Python 3.10.0 64-bit" and "Ln 1, Col 16 Spaces: 4 UTF-8 CRLF Python".



- Next, you can click on “Play” button located in the upper right corner

A screenshot of the Visual Studio Code interface. The top bar shows "File Edit Selection View Go Run Terminal Help" and the title "test.py - test - Visual Studio Code". The left sidebar has icons for Explorer, Search, Symbols, Find, and Outline, with "TEST" expanded and "test.py" selected. The main editor area shows the code:

```
1 print("Hello!")
```

The bottom right corner of the editor area is highlighted with an orange box. The bottom navigation bar includes "PROBLEMS", "OUTPUT", "DEBUG CONSOLE", and "TERMINAL". The terminal tab is active, showing the command "xe c:/Users/joe/Desktop/test/test.py" and the output "Hello!". The bottom status bar indicates "Python 3.10.0 64-bit" and "Ln 1, Col 16 Spaces: 4 UTF-8 CRLF Python".



- Now we're ready to start learning Python!
- Before we do though, **we need to note that when there's a >>> in an example code, it means that the code is run directly on the interactive interpreter.**

```
>>> print("Hello")
```

Run through the Interactive interpreter

- **Otherwise, it'll be run as a script on a single file as we've seen.**

```
print("Hello")
```

Run as a single file on VSCode



# Getting started

## Playing with Python interpreter

- Comments can be defined with # as follows

```
>>> # Hello, I'm just a comment :)  
>>> # Comments help you document your code.  
>>>
```



SECURITY FLAME

# Variables

- You define variables with the assignment operator =
- You don't need to indicate the variable type

```
>>> a = 5  
>>> b = 8.5  
>>> name = "Yousuf"
```



SECURITY FLAME

# Numbers

- Types of numbers:
  - Integers (int): 5, 8, 10, 5129, ...
  - Float (float): 1.8, 9.50, 103.8001, 3102.818234, ...
  - Complex: 2+3j, ...



# Arithmetic operators

## ❖ Addition (+)

```
>>> 4 + 2  
6  
>>> 8 + 8  
16
```

## ❖ Subtraction (-)

```
>>> 4 - 2  
2  
>>> 8 - 8  
0
```

## ❖ Multiplication (\*)

```
>>> 8*5  
40  
>>> 4*(5-2)  
12  
>>> (7-2)*(5+4)  
45
```

## ❖ Division (/)

```
>>> 10/2      # returns a float  
5.0  
>>> 5/2       # returns a float  
2.5  
>>> 5//2      # returns an integer  
2
```



# Arithmetic operators (cont.)

## ❖ Modulo (%)

```
>>> 5%2  
1  
>>> 12%3  
0  
>>> 2%3  
2  
>>> 15%4  
3
```

## ❖ Exponent (\*\*)

```
>>> 5**2    # 5 to the power of 2    = 5*5  
25  
>>> 6**3    # 6 to the power of 3    = 6*6*6  
216  
>>> 10**10   # 10 to the power of 10 = 10*10*10*10*10*...  
10000000000
```



# Integer representations

## ❖ Binary (b)

```
>>> 0b01  
1  
>>> 0b10  
2  
>>> 0b11  
3  
>>> 0b100  
4
```

## ❖ Octal (o)

```
>>> 0o7  
7  
>>> 0o10  
8  
>>> 0o17  
15  
>>> 0o20  
16
```

## ❖ Hexadecimal (x)

```
>>> 0x5  
5  
>>> 0xf  
15  
>>> 0x10  
16  
>>> 0xff  
255
```



SECURITY FLAME

# Large numbers

```
>>> 3600050 - 350
3599700
>>> 3_600_050 - 350
3599700
>>>
>>> 100000000 + 500000
100500000
>>> 100_000_000 + 500_000
100500000
```



SECURITY FLAME

## Large numbers (cont.)

- Python supports very large numbers



# Scientific notations

- Exponents can be expressed with “e”

```
>>> 2e10  
20000000000.0  
>>> 4.21e5  
421000.0  
>>> 9.4192e3  
9419.2  
>>> 3.2e-1  
0.32  
>>> 0.4192e-2  
0.004192  
>>> 1.33315e-5  
1.33315e-05
```



SECURITY FLAME

# Numbers playground

```
>>> a = 2
>>> b = 3.5
>>> c = a*b
>>> c
7.0
>>> c * 3
21.0
>>> d = _ / 7    # _ means the result of the previous operation
>>> d
3.0
```



## Numbers playground (cont.)

```
>>> a = 0xf          # 15 in decimal
>>> b = 0b101        # 5 in decimal
>>> c = 0o10          # 8 in decimal
>>> d = a * b * c
>>> d
600
```



SECURITY FLAME

## Numbers playground (cont.)

```
>>> 5 * 8 - 6  
34  
>>> 5 * (8 - 6)  
10  
>>> 2**3 * 2 - 1  
15  
>>> 2**3 * (2 - 1)  
8  
>>> -2**2  
-4  
>>> (-2)**2  
4
```



# Bitwise operators

## ❖ & (Bitwise AND)

```
>>> # 7 is 0111 in binary
>>> # 10 is 1010 in binary
>>> 7 & 10
2
>>> # 2 is 0010 in binary
```

## ❖ | (Bitwise OR)

```
>>> # 7 is 0111 in binary
>>> # 10 is 1010 in binary
>>> 7 | 10
15
>>> # 15 is 1111 in binary
```

## ❖ ~ (Bitwise NOT)

```
>>> # 7 is 0111 in binary
>>> ~7 & 0xf # ANDing it with 0xf to discard higher bits.
8
>>> # 8 is 1000 in binary.
```



# Bitwise operators (cont.)

## ❖ ^ (Bitwise XOR)

```
>>> # 7 is 0111 in binary
>>> # 10 is 1010 in binary
>>> 7 ^ 10
13
>>> # 13 is 1101 in binary
```

## ❖ >> (Bitwise right shift)

```
>>> # 7 is 0111 in binary
>>> 7 >> 1
3
>>> # 3 is 0011 in binary
>>> 7 >> 2
1
>>> # 1 is 0001 in binary
```

## ❖ << (Bitwise left shift)

```
>>> # 7 is 0111 in binary
>>> 7 << 1
14
>>> # 14 is 1110 in binary
>>> 7 << 2
28
>>> # 28 is 11100 in binary
```



## Other assignment operators

- Any arithmetic or bitwise operator can be combined with `=` to alter a value of a variable.

```
>>> x = 2
>>> x += 4      # x = x + 4
>>> x
6
>>> x //= 2     # x = x // 2
>>> x
3
>>> x <=> 2    # x = x << 2
>>> x
12
```



# Boolean

- Boolean variables refer to the variables that take either True or False. **Notice that the initial letter is a capital letter.**

```
>>> a = True  
  
>>> b = False
```



# Logical operators

❖ and

```
>>> a = True  
>>> b = False  
>>> a and b  
False
```

❖ or

```
>>> a = True  
>>> b = False  
>>> a or b  
True
```

❖ not

```
>>> a = True  
>>> b = False  
>>> not a  
False  
>>> not b  
True  
>>> a and not b  
True
```



# Comparison Operators

❖ > (larger than)

```
>>> 2 > 3  
False  
>>> 3 > 2  
True  
>>> 3 > 3  
False
```

❖ < (less than)

```
>>> 2 < 3  
True  
>>> 3 < 2  
False  
>>> 3 < 3  
False
```

❖ == (equality)

```
>>> 2 == 3  
False  
>>> 3 == 2  
False  
>>> 3 == 3  
True
```



## Comparison Operators (cont.)

❖ != (not equal)

```
>>> 2 != 3
True
>>> 3 != 2
True
>>> 3 != 3
False
```

❖ >= (larger or equal)

```
>>> 2 >= 3
False
>>> 3 >= 2
True
>>> 3 >= 3
True
```

❖ <= (less or equal)

```
>>> 2 <= 3
True
>>> 3 <= 2
False
>>> 3 <= 3
True
```



# Strings

```
>>> "I like Winter"
'I like Winter'
>>> 'I don\'t like Summer'
"I don't like Summer"
>>> phrase = "In Python, you have to enclose your strings with
either \" or '."
>>> print(phrase)
In Python, you have to enclose your strings with either " or '.
>>> print('My mother doesn\'t like Winter')
My mother doesn't like Winter
```



## Strings (cont.)

```
>>> print("The file is located at C:\test\nasser\test.txt")
The file is located at C:      est
asser  est.txt

>>> # \t means a tab
>>> # \n means a newline

>>> print(r"The file is located at C:\test\nasser\test.txt") # raw
The file is located at C:\test\nasser\test.txt
```



SECURITY FLAME

## Strings (cont.)

```
>>> print("""\n... You can span your string through multiple lines\n... using triple-quotes.\n... """)\nYou can span your string through multiple lines\nusing triple-quotes.
```



## Strings (cont.)

```
>>> "Hello" + " " + "my friend!"  
'Hello my friend!'  
>>> "Hello"*4  
'HelloHelloHelloHello'  
>>> "I then laughed so hard! " + "Ha"*8 + "!"  
'I then laughed so hard! HaHaHaHaHaHaHa!'
```



## Strings (cont.)

```
>>> "Python" "Is" "Powerful"
'PythonIsPowerful'
>>> questions = ("Is Earth flat? "
...                 "Why is sky blue? "
...                 "What's the meaning of life?"
...             )
>>> questions
"Is Earth flat? Why is sky blue? What's the meaning of life?"
```



## Strings (cont.)

```
>>> text = "There's so much potential in Python!"  
>>> text[0]  
'T'  
>>> text[1]  
'h'  
>>> text[2]  
'e'  
>>> text[-1]  
'!'  
>>> text[-2]  
'n'  
>>> text[0:5]  
'There'  
>>> text[0:15]  
"There's so much"  
>>> text[-7:]  
'Python!'
```



SECURITY FLAME

## Strings (cont.)

```
>>> text = "rat ops cat kit eat tab"  
>>> text[0::4]  
'rocket'
```



## Strings (cont.)

- In Python, you can reverse a string easily

```
>>> text = "Hello there!"  
>>> text[::-1]  
'!ereht olleH'  
>>> text[-1::-1]      # text[-1::-1] == text[::-1]  
'!ereht olleH'
```



## Strings (cont.)

```
>>> "I like number " + 55
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

- We will talk more about strings later.



# Lists

- A List is used to store multiple elements of possibly different types.

```
>>> [5, 1, 4]
[5, 1, 4]
>>> numbers = [5, 1, 4]
>>> numbers[0]
5
>>> numbers[-1]
4
```



## Lists (cont.)

```
>>> names = ["Yousuf", "Khalid", "Mohammed", "Ibrahim"]
>>> names[0]
'Yousuf'
>>> names[3]
'Ibrahim'
>>> names[-1]
'Ibrahim'
>>> names[-1][0]
'I'
```



## Lists (cont.)

```
>>> names = ["Yousuf", "Khalid", "Mohammed", "Ibrahim"]
>>> names[:2]
['Yousuf', 'Khalid']
>>> names[2:]
['Mohammed', 'Ibrahim']
>>> names[:]
['Yousuf', 'Khalid', 'Mohammed', 'Ibrahim']
>>> names[::2]
['Yousuf', 'Mohammed']
```



## Lists (cont.)

- A list can be reversed the same way we did with strings earlier.

```
>>> names = ["Yousuf", "Khalid", "Mohammed", "Ibrahim"]
>>> names[::-1]
['Ibrahim', 'Mohammed', 'Khalid', 'Yousuf']
>>> names[-1::-1]
['Ibrahim', 'Mohammed', 'Khalid', 'Yousuf']
```



## Lists (cont.)

- Lists can be merged in other lists.

```
>>> names
['Yousuf', 'Khalid', 'Mohammed', 'Ibrahim']
>>> numbers
[5, 1, 4]
>>> two_lists = [numbers, names]
>>> two_lists
[[5, 1, 4], ['Yousuf', 'Khalid', 'Mohammed', 'Ibrahim']]
>>> two_lists[0]
[5, 1, 4]
>>> two_lists[1]
['Yousuf', 'Khalid', 'Mohammed', 'Ibrahim']
>>> two_lists[1][2]
'Mohammed'
>>> two_lists[1][2][0]
'M'
```

- We will talk more about lists later.



# Tuples

```
>>> 5,  
(5,)  
>>> 5, 1, 4  
(5, 1, 4)  
>>> (5, 1, 4)  
(5, 1, 4)  
>>> numbers_tuple = (5, 1, 4)  
>>> numbers_tuple  
(5, 1, 4)
```

```
>>> numbers_tuple  
(5, 1, 4)  
>>> numbers[0]  
5  
>>> numbers[-1]  
4
```



## Tuples (cont.)

```
numbers
(5, 1, 4)
>>> x, y, z = numbers
>>> x
5
>>> y
1
>>> z
4
```

- We will talk more about tuples later.



# Sets

```
>>> {5, 1, 4}  
{1, 4, 5}  
>>> {5, 1, 4, 5, 5, 5, 4, 4, 4, 1, 1, 1}  
{1, 4, 5}  
>>> numbers = {5, 1, 4}      # No duplicates  
>>> numbers  
{1, 4, 5}
```

```
>>> letters = {'a', 'p', 'p', 'l', 'e'}  
>>> letters  
{'e', 'a', 'p', 'l'}      # No duplicates
```

- We will talk more about sets later.



# Dictionaries

- A dictionary is used to store related elements in key:value pairs.

```
>>> {} # empty dictionary
{}
>>> {'a':1, 'b':2, 'c':3}
{'a': 1, 'b': 2, 'c': 3}
>>> my_dictionary = {'a':1, 'b':2, 'c':3}
>>> my_dictionary['a']
1
>>> my_dictionary['b']
2
>>> my_dictionary['c']
3
>>> my_dictionary['d']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'd'
```



## Dictionaries (cont.)

```
>>> users = {"Yousuf": [94528506, "Bidiyah"],  
...           "Khalid": [97979797, "Muscat"],  
...           "Mohammed": [96969696, "Salalah"],  
...           "Ibrahim": [95959595, "Alburaimi"]  
...       }  
>>> users['Khalid']  
[97979797, 'Muscat']  
>>> users['Khalid'][0]  
97979797  
>>> users['Khalid'][1]  
'Muscat'  
>>> users['Khalid'][0] = 90909090  
>>> users['Khalid']  
[90909090, 'Muscat']
```



## Dictionaries (cont.)

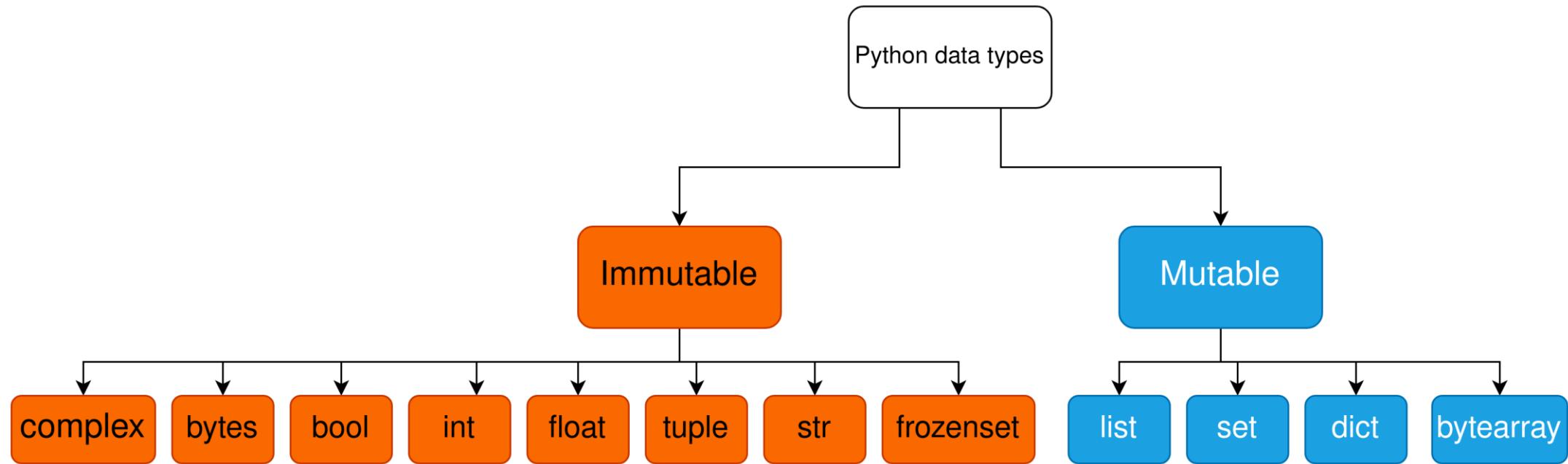
```
>>> d = {"test": 123, "test": 456}  
>>> d  
{'test': 456}
```

- We will talk more about dictionaries later.



# Mutability

- ❖ Mutability basically means the tendency of something to change.
- ❖ In Python, this means the tendency of an “object” to change.





## Mutability (cont.)

```
>>> a = 5      # Immutable
>>> b = a
>>> id(a)    # id returns an integer that is the identity of the object.
140235258083760
>>> id(5)
140235258083760
>>> id(a) == id(5)
True
>>> id(a) == id(b)
True
>>> a = a + 1      # Immutable
>>> id(a)          # The object's identity has totally changed.
140235258083792
>>> id(a) == id(b)
False
```



## Mutability (cont.)

```
>>> a = [1, 2, 3, 4] # Mutable
>>> b = a
>>> id(a)
140235255723136
>>> a.append(5)      # Adds the item 5 to the list.
>>> a
[1, 2, 3, 4, 5]
>>> id(a)          # a's identity has not changed.
140235255723136
>>> b              # Since b refers to the same object, b has also changed.
[1, 2, 3, 4, 5]
```



## Mutability (cont.)

```
>>> text = "Mohammed"    # Immutable
>>> id(text)
140235255724592
>>> text += " likes Winter."
>>> text
'Mohammed likes Winter.'
>>> id(text)
140235255720816
```



## Mutability (cont.)

```
>>> text = "Oman"          # Immutable
>>> id(text)
140235255724656
>>> text[0] = "A"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> text = "A" + text[1:]
>>> text
'Aman'
>>> id(text)
140235255735728
```



## Mutability (cont.)

```
>>> l = [1, 2, 3, 4]      # Mutable
>>> t = (1, 2, 3, 4)      # Immutable
>>> l[0] = 0
>>> l
[0, 2, 3, 4]
>>> t[0] = 0              # You can't modify elements in a tuple.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```



## Mutability (cont.)

```
>>> a = [1, 2, 3, 4]      # Mutable
>>> b = a
>>> text1 = "Hello"       # Immutable
>>> text2 = text1
>>> b.append(5)           # Adds item 5 to the list
>>> text2 += " there"
>>> a
[1, 2, 3, 4, 5]
>>> text1
'Hello'
```



# Mutability (cont.)

## ➤ **Mutable VS Immutable**

- Python Mutable data types are appropriate when you need to change them throughout your program.

```
>>> results = []
>>> results.append(5+2)
>>> results.append(8-3)
>>> results.append(5/2)
>>> results
[7, 5, 2.5]
```

- Python Immutable data types are appropriate when your data does not change throughout your program.

```
>>> config = ("192.168.1.8", 8888)
>>>
```



# Lists, Tuples, Sets, Dictionaries

	Lists	Tuples	Sets	Dictionaries
Mutable	✓	✗	✓	✓
Ordered	✓	✓	✗	✗
Slicing	✓	✓	✗	✗
Duplicate items	✓	✓	✗	Cannot have duplicate keys. Can have duplicate values.



SECURITY FLAME

# Lists, Tuples, Sets, Dictionaries

- The following can help you decide when to use which data type
  - Use a list when
    - You have a specific order for your list items.
    - You want to modify those items at run time.
  - Use a tuple when
    - You have a specific order for your tuple items.
    - You don't need to modify those items at run time.
  - Use a set when
    - You don't necessarily care about the order in your set.
    - You need each item in your set to be unique (no duplicates).
    - You want to take advantage of the set features.
    - All your set items are immutable; sets cannot contain mutable items such as lists.
  - Use a dictionary when
    - You need a key/value association.



# Flow control & Loop statements

## □ if statement

```
>>> a = True
>>> if(a == True):
...     print("a is true!")
...
a is true!
>>> if(a):
...     print("a is not false!")
...
a is not false!
>>> a = [1, 2, 3, 4]
>>> if(a):
...     print("a is not empty.")
...
a is not empty.
>>> b = []
>>> if(b):
...     print("b is not empty")
...
```



SECURITY FLAME

## Flow control & Loop statements (cont.)

### ❑ if statement (cont.)

```
>>> a = 5
>>> if(a > 0):
...     print("a is greater than 0")
...
a is greater than 0
>>> if(a > 0 and a < 10):
...     print("a is greater than 0 and less than 10")
...
a is greater than 0 and less than 10
```



SECURITY FLAME

## Flow control & Loop statements (cont.)

### □ if statement (cont.)

Now we will switch to VSCode.

```
a = int(input("Enter a number: "))
if(a == 5):
    print("You are correct!")
```

When a is not 5

```
Enter a number: 1
```

When a is 5

```
Enter a number: 5
You are correct!
```



SECURITY FLAME

## Flow control & Loop statements (cont.)

- Combining **if** with **else** statement

```
a = int(input("Enter a number: "))
if(a == 5):
    print("You are correct!")
else:
    print("You are not correct.")
```

When a is not 5

```
Enter a number: 3
You are not correct.
```

When a is 5

```
Enter a number: 5
You are correct!
```



## Flow control & Loop statements (cont.)

- Combining **elif** with both **if** and **else** statements

```
a = int(input("Enter a number: "))
if(a == 5):
    print("You are correct!")
elif(a == 4):
    print("You are getting close!")
elif(a == 6):
    print("You are getting further away!")
else:
    print("You are not correct.")
```

When a is 4

```
Enter a number: 4
You are getting close!
```

When a is 6

```
Enter a number: 6
You are getting further away!
```

When a is 5

```
Enter a number: 5
You are correct!
```



# Flow control & Loop statements (cont.)

## □ while statement

As long as the answer is incorrect, while loop block will be executed.

The only way to exit the loop is through entering the correct answer(i.e., 5).

Output

```
correct_answer = 5
a = 0 # temporary starting value
while a != correct_answer:
    a = int(input("Enter a number: "))
    if(a == correct_answer):
        print("You are correct!")
    elif(a == 4):
        print("You are getting close!")
    elif(a == 6):
        print("You are getting further away!")
    else:
        print("You are not correct.")
```

```
Enter a number: 2
You are not correct.
Enter a number: 4
You are getting close!
Enter a number: 6
You are getting further away!
Enter a number: 5
You are correct!
```



SECURITY FLAME

## Flow control & Loop statements (cont.)

### ❑ for statement

```
l = [1, 2, 3, 4, 5]  
  
for i in l:  
    print(i)
```

Output

```
1  
2  
3  
4  
5
```



SECURITY FLAME

## Flow control & Loop statements (cont.)

### ❑ for statement (cont.)

```
fruits = ["banana", "watermelon", "pineapple", "orange", "apple"]

for fruit in fruits:
    if(fruit == "orange"):
        print("We found " + fruit + "!")
```

Output

We found orange!



# Flow control & Loop statements (cont.)

## □ range function

```
>>> range(5)
range(0, 5)
>>> range(10)
range(0, 10)
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 15))
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> list(range(2, 8, 2))
[2, 4, 6]
>>> list(range(8, 0, -1))
[8, 7, 6, 5, 4, 3, 2, 1]
>>> list(range(8, -1, -1))
[8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> [i for i in range(5)]
[0, 1, 2, 3, 4]
>>> [i*10 for i in range(5)]
[0, 10, 20, 30, 40]
```



## Flow control & Loop statements (cont.)

### ❑ for statement (cont.)

```
fruits = ["banana", "watermelon", "pineapple", "orange", "apple"]
number_of_fruit_types = len(fruits) # length of fruits list

for i in range(number_of_fruit_types):
    print("Looking at " + str(i))
    if(fruits[i] == "orange"):
        print("We found an orange at index " + str(i) + "!")
```

Output

```
Looking at 0
Looking at 1
Looking at 2
Looking at 3
We found an orange at index 3!
Looking at 4
```



SECURITY FLAME

# Flow control & Loop statements (cont.)

## ❑ for statement (cont.)

```
correct_answer = 5
for i in range(3): # Only tries 3 times!
    a = int(input("Enter a number: "))
    if(a == correct_answer):
        print("You are correct!")
    elif(a == 4):
        print("You are getting close!")
    elif(a == 6):
        print("You are getting further away!")
    else:
        print("You are not correct.")
```

Output

```
Enter a number: 2
You are not correct.
Enter a number: 8
You are not correct.
Enter a number: 6
You are getting further away!
```



## Flow control & Loop statements (cont.)

### ❑ for statement (cont.)

There's a problem though...

What if we already know the answer?

Output

```
correct_answer = 5
for i in range(3): # Only tries 3 times!
    a = int(input("Enter a number: "))
    if(a == correct_answer):
        print("You are correct!")
    elif(a == 4):
        print("You are getting close!")
    elif(a == 6):
        print("You are getting further away!")
    else:
        print("You are not correct.")
```

```
Enter a number: 5
You are correct!
Enter a number: 5
You are correct!
Enter a number: 5
You are correct!
```



## Flow control & Loop statements (cont.)

- ❑ break statement on loops

```
correct_answer = 5
for i in range(3):
    a = int(input("Enter a number: "))
    if(a == correct_answer):
        print("You are correct!")
        break
    elif(a == 4):
        print("You are getting close!")
    elif(a == 6):
        print("You are getting further away!")
    else:
        print("You are not correct.")
```

Output

```
Enter a number: 2
You are not correct.
Enter a number: 5
You are correct!
```



# Flow control & Loop statements (cont.)

## ❑ continue statement on loops

What if we want to ignore any value larger than 10? (i.e., not displaying any output if input is larger than 10)

```
correct_answer = 5
for i in range(3):
    a = int(input("Enter a number: "))
    if(a == correct_answer):
        print("You are correct!")
        break
    elif(a == 4):
        print("You are getting close!")
    elif(a == 6):
        print("You are getting further away!")
    elif(a > 10):
        continue      # Go to next loop. Don't execute the code below.
    else:
        print("You are not correct.")
```

Output

```
Enter a number: 9
You are not correct.
Enter a number: 11
Enter a number: 20
```



## Flow control & Loop statements (cont.)

### ❑ break vs continue statement on loops

- **break** breaks the loop. No more loops get executed.
- **continue** “continues” the loop at the next iteration without executing the code below it at that iteration.



# Flow control & Loop statements (cont.)

## ❑ else statement on loops

**else** block gets executed **when the loop does not hit any break**.

```
correct_answer = 5
for i in range(3):
    a = int(input("Enter a number: "))
    if(a == correct_answer):
        print("You are correct!")
        break
    elif(a == 4):
        print("You are getting close!")
    elif(a == 6):
        print("You are getting further away!")
    elif(a > 10):
        continue
    else:
        print("You are not correct.")
else:
    # This will be executed only if the answer is not correct (no break).
    print("You have failed the mission!")
```

Output

```
Enter a number: 8
You are not correct.
Enter a number: 1
You are not correct.
Enter a number: 2
You are not correct.
You have failed the mission!
```



# Flow control & Loop statements (cont.)

## ❑ match statement

Added as of Python 3.10

```
f_name = "abc.log"
action = int(input("Enter your action to the file: "))
match action:
    case 0:
        print("Deleting the file..")
        # .. delete_file(f_name)
    case 1:
        print("Creating the file..")
        # .. create_file(f_name)
    case 2:
        print("Updating the file..")
        # .. update_file(f_name)
    case _:
        print("Unknown command!")
```

Output when action is 1

```
Enter your action to the file: 1
Creating the file..
```

Output when action is 2

```
Enter your action to the file: 2
Updating the file..
```

Output when action is 4

```
Enter your action to the file: 4
Unknown command!
```



# Flow control & Loop statements (cont.)

## □ match statement (cont.)

```
error_code = 3
match error_code:
    case 0:
        error_text = "Invalid input"
    case 1:
        error_text = "File is too large"
    case 2:
        error_text = "Cannot establish connection"
    case _:
        error_text = "Other error"
print(error_text)
```

Output

Other error

What if the error\_code was a negative number?

We don't want error\_code to be valid when it's negative.

In other words, we don't want error\_codes with negative numbers.

Meaning only positive (+0) error\_codes are acceptable.



# Flow control & Loop statements (cont.)

## ❑ match statement (cont.)

```
error_code = -1
match error_code:
    case 0:
        error_text = "Invalid input"
    case 1:
        error_text = "File is too large"
    case 2:
        error_text = "Cannot establish connection"
    case _ if error_code > 0:
        error_text = "Other error"
    case _:
        error_text = None
print(error_text)
```

Output

None



# Flow control & Loop statements (cont.)

## ❑ assert statement

- **assert** is used to make sure that a condition is met. If it's not met, the program raises `AssertionError`.
- **assert** is usually used as a debugging statement since it causes an error that halts the program in case the condition is not met.
- **assert** takes a condition and an optional error message.

```
name = input("Enter your name: ")
assert len(name) != 0 # make sure name length is not 0
print(name[::-1])
```

Output when entering valid input

```
Enter your name: hello
olleh
```

Output when hitting enter twice

```
Enter your name:
Traceback (most recent call last):
  File "/tmp/test.py", line 3, in <module>
    assert len(name) != 0
AssertionError
```



## Flow control & Loop statements (cont.)

### ❑ assert statement (cont.)

- You can add an optional message

```
name = input("Enter your name: ")
assert len(name) != 0, "name is empty"
print(name[::-1])
```

Output when hitting enter twice

```
Enter your name:
Traceback (most recent call last):
  File "/tmp/test.py", line 2, in <module>
    assert len(name) != 0, "name is empty"
AssertionError: name is empty
```



## Flow control & Loop statements (cont.)

### ❑ pass statement

- **pass** means literally just to let the execution pass without doing anything.

```
# This will loop forever doing nothing (no action).
while True:
    pass
```

A place holder for the function's code (we will discuss functions next)

```
# Remember to implement this function!
def my_function():
    pass
```

Creating a simple class with minimal code

```
class EmptyClass:
    pass
```



# Functions

- ❑ Defining a function

```
def do_nothing():
    pass
```

- ❑ Defining a function & calling it

```
def say_hi():
    print("Hi!")

say_hi()
```

Output of function say\_hi

```
Hi!
```



## Functions (cont.)

```
def say_hi_to(name):
    print("Hi " + name + "!")

say_hi_to("Mohammed")
say_hi_to("Khalid")
```

Output

```
Hi Mohammed!
Hi Khalid!
```



SECURITY FLAME

## Functions (cont.)

```
def add(x, y):  
    return x+y  
  
print(add(2, 4))
```

Output

6



## Functions (cont.)

```
def average(l):
    total = 0
    for i in l:
        total += i
    return total/len(l)

print("The average is:", average([1,2,3,4,5]))
```

Output

The average is: 3.0



# Functions (cont.)

- Default arguments

```
def add(x, y=0):  
    return x+y  
  
print(add(2))  
print(add(2, 4))
```

Output

```
2  
6
```



# Functions (cont.)

- Default arguments (cont.)

```
def say_hi(name="Default"):
    print("Hi " + name + "!")

say_hi()
say_hi("Mohammed")
```

Output

```
Hi Default!
Hi Mohammed!
```



## Functions (cont.)

- Default arguments (cont.)
  - Normal arguments cannot come after default arguments

```
def add(x, y=10, z):  
    return x+y+z
```

Output

```
def add(x, y=10, z):  
    ^  
SyntaxError: non-default argument follows default argument
```



# Functions (cont.)

## ➤ Keyword arguments

```
def say_hi(name="Default"):
    print("Hi " + name + "!")

say_hi()                      # Hi Default!
# passing a value through the name keyword
say_hi(name="Mohammed")       # Hi Mohammed!
```

```
def add(x=0, y=0):
    return x+y

print(add())                  # 0
print(add(x=2))              # 2
print(add(x=2, y=4))         # 6
```



# Functions (cont.)

- Keyword arguments (cont.)

```
def do(op, *arguments):
    if(op == '+'):
        total = 0
        for i in arguments:
            total += i
    elif(op == '*'):
        total = 1
        for i in arguments:
            total *= i
    return total
print(do('+', 5, 4, 3, 2))
print(do('*', 5, 5))
```

Output

14  
25



# Functions (cont.)

- Keyword arguments (cont.)

```
def func(simple, *arguments, **keywords):
    print("simple:", simple)
    print("arguments:", arguments)
    print("keywords:", keywords)

func(1, 2, 3, 4, 5, 6, seven=7, eight=8, nine=9)
```

Output

```
simple: 1
arguments: (2, 3, 4, 5, 6)
keywords: {'seven': 7, 'eight': 8, 'nine': 9}
```



## Functions (cont.)

- Standard, Positional-only, Positional-or-Keyword, and Keyword-Only arguments
  - By default, **standard arguments can be passed either by position or by keyword**

```
def func(name):  
    print(name)  
func("Mohammed")      # by position  
func(name="Mohammed") # by keyword
```

Output

```
Mohammed  
Mohammed
```



## Functions (cont.)

- Standard, Positional-only, Positional-or-Keyword, and Keyword-Only arguments (cont.)

- What if you see this function?

```
def add_person(name, age=0, mobile=None):  
    print(name, age)  
    print(mobile)
```

- This function can be called through several ways

```
add_person("Saif")  
add_person(name="Saif")  
add_person("Saif", age=29)  
add_person("Saif", 29, 90909090)  
add_person("Saif", 29, mobile=90909090)  
add_person("Saif", age=29, mobile=90909090)  
add_person(name="Saif", age=29, mobile=90909090)
```

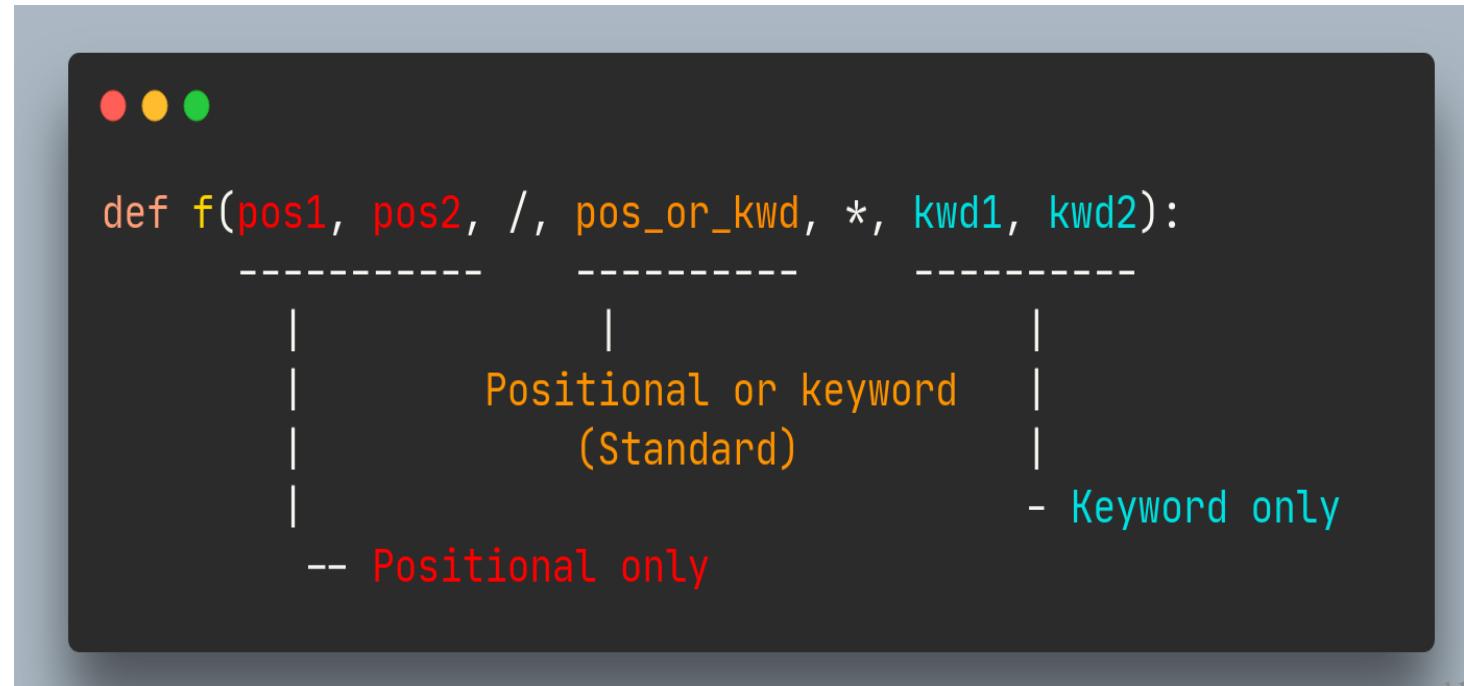
- This can cause some confusion, especially when there are several arguments and several people developing the code.
  - What if we want to restrict the way we provide arguments?



## Functions (cont.)

- Standard, Positional-only, Positional-or-Keyword, and Keyword-Only arguments (cont.)
  - Removing the confusion: (Python Enhancement Proposal) PEP 570 (Python 3.8)
  - Thanks to PEP 570, we can now specify which arguments are positional-only, and keyword-only.

- **RED**: can only be passed by position.
- **Orange**: can be passed by position/keyword.
- **Light blue**: can be passed by keyword only.





## Functions (cont.)

- Standard, Positional-only, Positional-or-Keyword, and Keyword-Only arguments (cont.)

```
● ● ●  
  
def func(name, age, /, mobile, *, other_info):  
    print("Positional-Only:", name, age)  
    print("Positional-Or-Keyword (Standard):", mobile)  
    print("Keyword-Only:", other_info)
```



## Functions (cont.)

- Standard, Positional-only, Positional-or-Keyword, and Keyword-Only arguments (cont.)

```
def func(name, age, /, mobile, *, other_info):
    print("Positional-Only:", name, age)
    print("Positional-Or-Keyword (Standard):", mobile)
    print("Keyword-Only:", other_info)

func("Saif", 29, 90909090, other_info={"address": "1234 Muscat"})
func("Saif", 29, mobile=90909090, other_info={"address": "1234 Muscat"})

# The following line will cause an error:
func(name="Saif", age=29, mobile=90909090, other_info={"address": "1234 Muscat"})
```

### Output

```
Positional-Only: Saif 29
Positional-Or-Keyword (Standard): 90909090
Keyword-Only: {'address': '1234 Muscat'}
```

```
Positional-Only: Saif 29
Positional-Or-Keyword (Standard): 90909090
Keyword-Only: {'address': '1234 Muscat'}
```

```
Traceback (most recent call last):
...
func(name="Saif", age=29, mobile=90909090, other_info={"address": "1234 Muscat"})
TypeError: func() got some positional-only arguments passed as keyword arguments: 'name, age'
```



## Functions (cont.)

- Standard, Positional-only, Positional-or-Keyword, and Keyword-Only arguments (cont.)
  - You can exclusively use Positional-Only & Keyword-Only arguments

```
def func(name, age, /, *, mobile, other_info):  
    print("Positional-Only:", name, age)  
    print("Keyword-Only:", mobile, other_info)  
  
func("Saif", 29, mobile=90909090, other_info={"address": "1234 Muscat"})  
  
# The next line will cause an error:  
func("Saif", 29, 90909090, other_info={"address": "1234 Muscat"})
```

### Output

```
Positional-Only: Saif 29  
Keyword-Only: 90909090 {'address': '1234 Muscat'}  
  
Traceback (most recent call last):  
...  
  func("Saif", 29, 90909090, other_info={"address": "1234 Muscat"})  
TypeError: func() takes 2 positional arguments but 3 positional arguments (and 1 keyword-only argument) were given
```



## Functions (cont.)

- Standard, Positional-only, Positional-or-Keyword, and Keyword-Only arguments (cont.)
  - You can exclusively use Positional-Only arguments

```
def func(name, age, mobile, other_info, /):  
    print("Positional-Only:", name, age, mobile, other_info)  
  
func("Saif", 29, 90909090, {"address": "1234 Muscat"})  
  
# The next line will cause an error:  
func("Saif", 29, mobile=90909090, other_info={"address": "1234 Muscat"})
```

### Output

```
Positional-Only: Saif 29 90909090 {'address': '1234 Muscat'}  
Traceback (most recent call last):  
...  
func("Saif", 29, mobile=90909090, other_info={"address": "1234 Muscat"})  
TypeError: func() got some positional-only arguments passed as keyword arguments: 'mobile, other_info'
```



## Functions (cont.)

- Standard, Positional-only, Positional-or-Keyword, and Keyword-Only arguments (cont.)
  - You can exclusively use Keyword-Only arguments

```
def func(*, name, age, mobile, other_info):  
    print("Keyword-Only:", name, age, mobile, other_info)  
  
func(name="Saif", age=29, mobile=90909090, other_info={"address":"1234 Muscat"})  
  
# The next line will cause an error:  
func("Saif", 29, 90909090, {"address":"1234 Muscat"})
```

### Output

```
Keyword-Only: Saif 29 90909090 {'address': '1234 Muscat'}  
Traceback (most recent call last):  
...  
func("Saif", 29, 90909090, {"address":"1234 Muscat"})  
TypeError: func() takes 0 positional arguments but 4 were given
```



## Functions (cont.)

➤ Standard, Positional-only, Positional-or-Keyword, and Keyword-Only arguments (cont.)

❖ Ok. Now when to use which?

- Positional-Only

- If your parameter names have no real meaning to the user.
- If you want to enforce the order of the function's arguments.
- If you want the parameter names to not be available to the user.
- Positional-Only arguments are recommended when writing an API. This helps preventing API issues if the parameter names are modified in future.

- Keyword-Only

- If you don't want to enforce the order of the function's arguments.
- If your parameter names have understandable meaning to the user.
- In case you want to prevent the user from supplying them in positional arguments.

- Positional-Or-Keyword (standard)

- If you're not writing an API & if you don't care so much 😊



## Functions (cont.)

- Unpacking arguments

```
def add(x, y, z):  
    return x + y + z  
  
l = [1, 2, 3]  
print(add(*l))          # Output: 6
```



# Functions (cont.)

## ➤ Anonymous functions

- Anonymous functions are defined with **lambda**
- We use them when we require a function for a short period of time.

Example 1

```
anon_add_four = lambda x: x + 4  
  
print(anon_add_four(6))          # 10  
print(anon_add_four(8))          # 12
```

Example 2

```
square = lambda x: x*x  
  
print(square(4))                # 16  
print(square(7))                # 49  
print(square(8))                # 64
```

Example 3

```
multiply = lambda x, y: x*y  
  
print(multiply(2, 5))            # 10
```



## Functions (cont.)

- Anonymous functions (cont.)
  - We can define it and call it in one line

```
print((lambda x, y: x*y)(2, 5))
```



# Functions (cont.)

## ➤ Anonymous functions (cont.)

- How about making it a little bit more complicated?

```
l = [5, 99, 21, 3123, 88, 99, 10, 22, 88, 109]
print((lambda l, f: [i for i in range(len(l)) if(l[i]==f)])(l, 88)) # Output: [4, 8]
```

- A clearer version of it

```
l = [5, 99, 21, 3123, 88, 99, 10, 22, 88, 109]
length_of_l = len(l)
find_item_index = lambda l, f: [i for i in range(length_of_l) if(l[i]==f)]
result = find_item_index(l, 88)
print(result) # Output: [4, 8]
```

- We'll see later how/when anonymous functions get more useful.



# Functions (cont.)

## ➤ Function documentation (docstrings)

- You can document your function. It's optional but recommended.
- Documenting your function adds the documentation to the `__doc__` property.
- Starts and ends with either `"""` or `' '' '`.

```
def add(x, y):
    """
    Adds x and y and returns the value.

    Parameters:
        x (int): a decimal integer
        y (int): a decimal integer

    Returns:
        (x + y) (int): an integer representing the sum of x and y
    """
    return x + y

print(add.__doc__) # Output: *the docstring written above*
```



# Functions (cont.)

- Function documentation (docstrings)
  - This documentation can be displayed conveniently with **help** built-in function

```
>>> help(add)
Help on function add in module __main__:

add(x, y)
    Adds x and y and returns the value.

    Parameters:
        x (int): a decimal integer
        y (int): a decimal integer

    Returns:
        (x + y) (int): an integer representing the sum of x and y
```



# Functions (cont.)

## ➤ Function Annotations

- Function annotations are optional.
- Annotations indicate the parameter type & return value.
- Annotating your function adds the annotations to the `__annotations__` property.

```
def add(x: int, y: int) -> int:  
    return x + y  
  
def average(l: list) -> float:  
    return sum(l)/len(l)
```



# Functions (cont.)

## ➤ Built-in functions: `hex` function

- Converts an integer to its hexadecimal representation value.
- Note that it returns a `str` type value.

```
>>> hex(1)
'0x1'
>>> hex(11)
'0xb'
>>> hex(15)
'0xf'
>>> hex(16)
'0x10'
>>> hex(0o10)
'0x8'
>>> hex(0b101)
'0x5'
>>> hex(0x11)
'0x11'
>>> type(hex(0x11))
<class 'str'>
```



# Functions (cont.)

- Built-in functions: **list** function
  - Converts an iterable item such as tuple or set into a list

```
>>> t = (1, 2, 3, 4)
>>> type(t)
<class 'tuple'>
>>> l = list(t)
>>> l
[1, 2, 3, 4]
>>> type(l)
<class 'list'>
```

- If no item is passed to **list**, then it creates an empty list

```
>>> l = list()
>>> l
[ ]
```



# Functions (cont.)

## ➤ Built-in functions: **tuple** function

- Converts an iterable item such as list or set into a tuple

```
>>> l = [1, 2, 3, 4]
>>> t = tuple(l)
>>> t
(1, 2, 3, 4)
```

- If no item is passed to **tuple**, then it creates an empty tuple

```
>>> empty_tuple = tuple()
>>> empty_tuple
()
```



## Functions (cont.)

### ➤ Built-in functions: `set` function

- Converts an iterable item such as list or tuple into a set

```
>>> l = [1, 2, 3, 4]
>>> s = set(l)
>>> s
{1, 2, 3, 4}
```

- If no item is passed to `set`, then it creates an empty set

```
>>> empty_set = set()
>>> empty_set
set()
```



# Functions (cont.)

## ➤ Built-in functions: **map** function

- **map** is an important built-in function
- It takes two arguments
  - A function.
  - Arguments for that function through an iterable object such as lists.
- It returns an iterator (we will talk more about iterators later) of the results after it passes them to the target function.

```
def add_two(number):  
    return number + 2  
  
l = [0, 8, 16, 24, 32]  
result = map(add_two, l)  
print(list(result))      # Output: [2, 10, 18, 26, 34]
```



## Functions (cont.)

- Built-in functions: **map** function (cont.)

```
def add_two_numbers(number1, number2):  
    return number1 + number2  
  
l1 = [0, 8, 16, 24, 32]  
l2 = [2, 4, 6, 8, 10]  
  
result = map(add_two_numbers, l1, l2)  
print(list(result))                                # Output: [2, 12, 22, 32, 42]
```



## Functions (cont.)

- Built-in functions: **map** function (cont.)
  - Recreating the previous example with **anonymous function**

```
l1 = [0, 8, 16, 24, 32]
l2 = [2, 4, 6, 8, 10]

result = map(lambda n1,n2: n1 + n2, l1, l2)
print(list(result))                                # Output: [2, 12, 22, 32, 42]
```



# Functions (cont.)

- Built-in functions: **map** function (cont.)

```
l1 = [0, 8, 16, 24, 32]
l2 = [2, 4, 6, 8, 10]
l3 = [5, 9, 2, 1, 4]

# adds (l1 + l2) * l3
result = map(lambda n1,n2,n3: (n1 + n2) * n3, l1, l2, l3)
result = list(result)

for i in range(len(result)):
    print('(', l1[i], '+', l2[i], ')', '*', l3[i], '=', result[i])
```

Output

```
( 0 + 2 ) * 5 = 10
( 8 + 4 ) * 9 = 108
( 16 + 6 ) * 2 = 44
( 24 + 8 ) * 1 = 32
( 32 + 10 ) * 4 = 168
```



# Functions (cont.)

- Built-in functions: **map** function (cont.)

What if the lists aren't of equal size?

**map** will continue until it reaches the end of one of the lists.

```
l1 = [0, 8, 16, 24, 32]
l2 = [2, 4, 6, 8, 10]
l3 = [5, 9, 2] # smaller than l1 & l2

# adds (l1 + l2) * l3
result = map(lambda n1,n2,n3: (n1 + n2) * n3, l1, l2, l3)
result = list(result)

for i in range(len(result)):
    print('(', l1[i], '+', l2[i], ')', '*', l3[i], '=', result[i])
```

Output

```
( 0 + 2 ) * 5 = 10
( 8 + 4 ) * 9 = 108
( 16 + 6 ) * 2 = 44
```



## Functions (cont.)

- Built-in functions: **map** function (cont.)

```
names = ["Rashid", "Mohammed", "Tariq", "Khalid"]

result = map(len, names)
result = tuple(result)

print(result)                      # Output: (6, 8, 5, 6)
```



## Functions (cont.)

### ➤ Built-in functions: **filter** function (cont.)

- **filter** behaves similarly to **map**, but it returns the results that match the condition of the target function.
- Note that unlike map, filter can only take one iterable item.

Only names that start with the letter “T”.

```
def starts_with_T(name):
    if(name[0] == 'T'):
        return True
    return False

names = ["Rashid", "Mohammed", "Tariq", "Khalid"]

result = filter(starts_with_T, names)
result = tuple(result)

print(result)      # Output: ['Tariq']
```



## Functions (cont.)

- Built-in functions: **filter** function (cont.)

Only names that consist of 6 letters

```
names = ["Rashid", "Mohammed", "Tariq", "Khalid"]

result = filter(lambda name: len(name) == 6, names)
result = tuple(result)

print(result)      # ('Rashid', 'Khalid')
```



SECURITY FLAME

## Functions (cont.)

- Built-in functions: **filter** function (cont.)

Filter out empty strings

```
names = ['', "Rashid", "Mohammed", "", "Tariq", "Khalid", ""]

result = filter(None, names)
result = list(result)

print(result) # Output: ['Rashid', 'Mohammed', 'Tariq', 'Khalid']
```



## Functions (cont.)

- Built-in functions: **filter** function (cont.)

Only values that are equal or larger than 0

```
l = [5, -8, -1, 22, 0, 58, -19, 13, 15]

result = filter(lambda x: x >= 0, l)
result = tuple(result)

print(result)      # Output: (5, 22, 0, 58, 13, 15)
```



## Functions (cont.)

- Built-in functions: **bytes** and **bytearray** function (cont.)

- **bytes** returns an object of type **bytes**.
- **bytearray** returns an object of type **bytearray**.

```
>>> l = [1, 2, 3, 4]
>>> bytes(l)
b'\x01\x02\x03\x04'
>>> type(bytes(l))
<class 'bytes'>
>>> bytearray(l)
bytearray(b'\x01\x02\x03\x04')
>>> type(bytearray(l))
<class 'bytearray'>
```



## Functions (cont.)

### ➤ Built-in functions: `bytes` and `bytearray` function (cont.)

- The most significant difference between them is `bytes` returns an **immutable** object, whereas `bytearray` returns a **mutable** object.

```
>>> l = [1, 2, 3, 4]
>>> my_bytes = bytes(l)
>>> my_bytes[0] = 5          # Error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment

>>> my_bytearray = bytearray(l)
>>> my_bytearray[0] = 5
>>> my_bytearray
bytearray(b'\x05\x02\x03\x04')
```



## Functions (cont.)

### ➤ Built-in functions: **bytes** and **bytearray** function (cont.)

- You can initialize **bytes** or **bytearray** objects to zero by passing the number of bytes you would like to have.

```
>>> bytes(5)
b'\x00\x00\x00\x00\x00'
>>> bytearray(5)
bytearray(b'\x00\x00\x00\x00\x00')
```

- You can pass a string to convert into bytes-like object.

```
>>> bytes("hello", encoding="utf-8")
b'hello'
>>> bytearray("hello", encoding="utf-8")
bytearray(b'hello')
```



## Functions (cont.)

- Built-in functions: **abs** function (cont.)
  - Returns the absolute value of the passed argument

```
>>> abs(5)
5
>>> abs(-5)
5
>>> abs(0 - 8)
8
>>> abs(35 - 37)
2
```



SECURITY FLAME

## More On Numbers

- Comparing floats

```
>>> (0.1 + 0.2) == 0.3
False
>>> 0.1 + 0.1 + 0.1 - 0.3 == 0
False
```

- Why?



## More On Numbers

- Comparing floats (cont.)
  - The reason is [IEEE 754](#)
  - Essentially, there are limitations in representing some floating-point numbers

```
>>> 0.2 + 0.1  
0.3000000000000004  
>>> 0.1 + 0.1 + 0.1 - 0.3  
5.551115123125783e-17
```

- For further read on why this happens, you can visit [0.3000000000000004.com](http://0.3000000000000004.com)
- Let's discuss how we can deal with it in Python.



# More On Numbers

- Comparing floats (cont.)
  - We can overcome this by implementing a function `is_equal_float`

```
def is_equal_float(a, b):  
    EPSILON = 1e-2  
    return abs(a - b) <= EPSILON  
  
a = 0.2 + 0.1  
b = 0.3  
print(is_equal_float(a, b))          # True  
print(is_equal_float(0.29, b))      # False  
print(is_equal_float(0.309, b))     # True  
print(is_equal_float(0.31, b))      # False
```



## More On Numbers

- Comparing floats (cont.)
  - We can also use the decimal module provided by Python

```
>>> from decimal import Decimal
>>> Decimal('0.2') + Decimal('0.1') == Decimal('0.3')
True
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3') == 0
True
```

- decimal module is recommended for accounting applications that deal a lot with numbers and have “strict equality invariants.”



## More On Strings

- Strings methods: **count** method
  - Counts the occurrences of the specified character
  - It can take optional start and end index.

```
>>> my_string = "Hello There!"  
>>> my_string.count('e')  
3  
>>> my_string.count('e', 0, 5)  
1
```



## More On Strings

- Strings methods: **encode** method
  - Converts the string into the specified encoding.
  - It returns an object of type “bytes”.

```
>>> my_string = "Hello There!"  
>>> my_string.encode() # UTF-8  
b'Hello There!'  
>>> type(my_string.encode())  
<class 'bytes'>  
>>> my_string.encode(encoding="UTF-16LE")  
b'H\x00e\x001\x001\x00o\x00 \x00T\x00h\x00e\x00r\x00e\x00!\x00'
```



## More On Strings

- Strings methods: `startswith/endswith` methods

```
>>> my_string
'Hello There!'
>>> my_string.startswith('B')
False
>>> my_string.startswith('H')
True
>>> my_string.startswith('Hello')
True
>>> my_string.endswith('!')
True
>>> my_string.endswith('e!')
True
```



# More On Strings

- Strings methods: **startswith/endswith** methods (cont.)

```
>>> my_string = "Hello There!"  
>>> my_string.startswith('T', 6)  
True  
>>> my_string.startswith('Th', 6)  
True  
>>> my_string.endswith('lo', 0, 3)  
False  
>>> my_string.endswith('lo', 0, 5)  
True
```



## More On Strings

- Strings methods: **find/rfind** methods

```
>>> my_string = "Hello There!"  
>>> my_string.find('e')  
1  
>>> my_string[1]  
'e'  
>>> my_string[1:]  
'ello There!'  
>>> my_string.rfind('e')  
10  
>>> my_string[10:]  
'e!'
```



## More On Strings

- Strings methods: **find/rfind** methods (cont.)

```
>>> my_string
'Hello There!'
>>> my_string.find('e', 6)
8
>>> my_string[8:]
'ere!'
>>> my_string.find('x')      # If not found returns -1
-1
```



# More On Strings

- Strings methods: **index/rindex** methods
  - Behaves similarly to **find / rfind** but throws an exception if the substring is not found.
  - **index** is available to strings, tuples, and lists while **find** is only available to strings.

```
>>> my_string = "Hello There!"  
>>> my_string.index('e')  
1  
>>> my_string.rindex('e')  
10  
>>> my_string.rindex('v')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: substring not found
```

- You can use specify the start and end for the second and third argument.



## More On Strings

- Strings methods: `lower/upper/swapcase` methods

```
>>> my_string
'Hello There!'
>>> my_string.upper()
'HELLO THERE!'
>>> my_string.lower()
'hello there!'
>>> my_string.swapcase()
'hELLO tHERE!'
```



## More On Strings

- Strings methods: **strip/rstrip/lstrip** methods
  - Removes white spaces by default

```
>>> my_string = "      Hello There!      "
>>> my_string.strip()
'Hello There!'
>>> my_string.rstrip()
'      Hello There!'
>>> my_string.lstrip()
'Hello There!      '
```



## More On Strings

- Strings methods: **strip/rstrip/lstrip** methods (cont.)

```
>>> my_string = ".....Hello There!*****"
>>> my_string.strip()
'.....Hello There!*****'
>>> my_string.strip('*')
'.....Hello There!'
>>> my_string.strip('.')
'Hello There!*****'
>>> my_string.strip('.').strip("*")
'Hello There!'
>>> my_string.lstrip('.').rstrip("*")
'Hello There!'
```



## More On Strings

- Strings methods: **split** method

```
>>> my_string = "Hello There!"  
>>> my_string.split(" ")  
['Hello', 'There!']  
>>> my_string = "Hello There! Python is so powerful!"  
>>> my_string.split(" ")  
['Hello', 'There!', 'Python', 'is', 'so', 'powerful!']  
>>> my_string.split(" ")[0]  
'Hello'  
>>> my_string.split(" ")[3]  
'is'
```



## More On Strings

- Strings methods: **split** method (cont.)

```
>>> my_string = "Hello There! Python is so powerful!    "
>>> my_string.split(" ")
['Hello', 'There!', 'Python', 'is', 'so', 'powerful!', '', '', '']
>>> list(filter(None, my_string.split(" ")))
['Hello', 'There!', 'Python', 'is', 'so', 'powerful!']
```



## More On Strings

- Strings methods: **replace** method

```
>>> my_string = "Hello There!"  
>>> my_string.replace("e", "3")  
'H3llo Th3r3!'  
>>> my_string.replace("e", "3").replace("l", "1")  
'H311o Th3r3!'  
>>> my_string.replace("There", "World")  
'Hello World!'
```



# More On Strings

- Strings methods: **maketrans / translate** methods
  - You can “translate” or “map” letters/characters so that they get replaced by others

```
>>> my_string = "Hello There!"  
>>> my_string.replace("e", "3").replace("o", "0").replace("T", "7")  
'H3ll0 7h3r3!'  
>>> my_string.maketrans("eloT", "3107")  
{101: 51, 108: 49, 111: 48, 84: 55}  
>>> my_string.translate({101: 51, 108: 49, 111: 48, 84: 55})  
'H3ll0 7h3r3!'
```



## More On Strings

- Strings methods: **join** method
  - Joins an iterable of strings

```
>>> words = ["Khalid", "likes", "Python"]
>>> ''.join(words)
'KhalidlikesPython'
>>> ' '.join(words)
'Khalid likes Python'
>>> ','.join(words)
'Khalid,likes,Python'
```



SECURITY FLAME

## More On Strings

- Strings methods: **format** method

```
>>> my_string = "Hello {}! You probably like {} and {}"  
>>> my_string.format("Khalid", "Python", "Formatting")  
'Hello Khalid! You probably like Python and Formatting'
```



## More On Strings

- Strings methods: **format** method (cont.)

```
>>> my_string = "Hello {0}! You probably like {1} and {2}"
>>> my_string.format("Khalid", "Python", "Formatting")
'Hello Khalid! You probably like Python and Formatting'
```

```
>>> my_string = "Hello {0}! You probably like {2} and {1}"
>>> my_string.format("Khalid", "Python", "Formatting")
'Hello Khalid! You probably like Formatting and Python'
```



## More On Strings

- Strings methods: **format** method (cont.)

```
>>> my_string = "Hello {name}! You probably like {item1} and {item2}"
>>> my_string.format(name="Khalid", item1="Python", item2="Formatting")
'Hello Khalid! You probably like Python and Formatting'
```

```
>>> my_string = "Hello {name}! You probably like {0} and {1}"
>>> my_string.format("Python", "Formatting", name="Khalid")
'Hello Khalid! You probably like Python and Formatting'
```



SECURITY FLAME

## More On Strings

- f-strings

```
>>> name = "Khalid"  
>>> print(f"Hello {name}")  
Hello Khalid
```

```
>>> name, items = "Khalid", ("Python", "Formatting")  
>>> my_string = f"Hello {name}! You probably like {items[0]} and {items[1]}"  
>>> print(my_string)  
Hello Khalid! You probably like Python and Formatting
```



# More On Strings

- f-strings (cont.)

```
def print_info(info):
    print("{0:8} - {1:8}".format("Name", "Phone"))
    for name, phone in info.items():
        print(f"{name:8} - {phone:8d}") # d for decimal.

info = {"Khalid": 90909090, "Mohammed": 91919191, "Tariq": 92929292}
print_info(info)
```

Output

```
Name      - Phone
Khalid   - 90909090
Mohammed - 91919191
Tariq    - 92929292
```



## More On Strings

- f-strings (cont.)

```
>>> name = "khalid"
>>> my_string = f"{name.upper()} likes the number {55*2}"
>>> print(my_string)
KHALID likes the number 110
```

- You can find more on f-strings [here](#)



## More On Strings

- The Old Way of formatting strings
  - You can use the % operator to indicate the location where the string/value should be placed.
  - This is called **string interpolation**.

```
>>> my_string = "%s likes the number %d" % ("Khalid", 55*2)
>>> print(my_string)
Khalid likes the number 110
```



## More On Strings

- The Old Way of formatting strings
  - You can use the % operator to indicate the location where the string/value should be placed.
  - This is called **string interpolation**.

```
>>> my_string = "%s likes the number %d" % ("Khalid", 55*2)
>>> print(my_string)
Khalid likes the number 110
```



# More On Strings

## ➤ Other string functions

- For an extended list of string methods, you can visit [Python page documentation](#).
- You can also get help interactively through Python's interactive shell

```
>>> my_string.  
my_string.capitalize()      my_string.ljust()  
my_string.casefold()       my_string.lower()  
...  
my_string.find()           my_string.replace()  
my_string.format()         my_string.rfind()  
...  
>>> help(my_string.replace)  
replace(old, new, count=-1, /) method of builtins.str instance  
    Return a copy with all occurrences of substring old replaced by new.  
  
count  
    Maximum number of occurrences to replace.  
    -1 (the default value) means replace all occurrences.
```

If the optional argument count is given, only the first count occurrences are replaced.



## More On Lists

- List methods: **append** method
  - Appends/adds the item to the end of the list.

```
>>> l = [1, 2, 3, 4, 5]
>>> l.append(6)
>>> l
[1, 2, 3, 4, 5, 6]
>>> l.append("Hello there")
>>> l
[1, 2, 3, 4, 5, 6, 'Hello there']
```



## More On Lists

- List methods: **insert** method

- Insert the item at index.

```
>>> l = ['a', 'c', 'e']
>>> l.insert(1, 'b')
>>> l
['a', 'b', 'c', 'e']
>>> l.insert(3, 'd')
>>> l
['a', 'b', 'c', 'd', 'e']
```



## More On Lists

- List methods: **extend** method
  - Appends the items of an iterable to the end of the list.

```
>>> l = [1, 2, 3, 4, 5]
>>> l.extend((6, 7, 8, 9))
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l.extend({10, 11, 12, 13})
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
>>> l.extend([14, 15])
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```



SECURITY FLAME

## More On Lists

- List methods: **count** method
  - Counts the occurrences of an item in a list.

```
>>> l = [ 'a', 'p', 'p', 'l', 'e' ]
>>> l.count('a')
1
>>> l.count('p')
2
>>> l = [8, 5, 9, 5, 5, 8]
>>> l.count(8)
2
>>> l.count(5)
3
>>> l.count(9)
1
```



# More On Lists

- List methods: **index** method
  - Returns the index where the item can be found

```
>>> word = ['a', 'p', 'p', 'l', 'e']
>>> word.index('e')
4
>>> word[4]
'e'
>>> word.index('p', 1, 3)
1
>>> word.index('p', 2, 3)
2
>>> word.index('p', 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'p' is not in list
```



## More On Lists

- List methods: **pop**/**remove** methods
  - **remove** removes the first occurrence of the value
  - **pop** “pops” the item at the index and returns it

```
>>> word = ['a', 'p', 'p', 'l', 'e']
>>> word.remove('p')
>>> word
['a', 'p', 'l', 'e']
>>> word.pop(1)
'p'
>>> word
['a', 'l', 'e']
```



## More On Lists

- List methods: **sort** method

- Sorts the list

```
>>> word = ['a', 'p', 'p', 'l', 'e']
>>> word.sort()
>>> word
['a', 'e', 'l', 'p', 'p']
>>> word.sort(reverse=True)
>>> word
['p', 'p', 'l', 'e', 'a']
```



## More On Lists

- List methods: **reverse** method
  - Reverses the order of the list

```
>>> word = ['a', 'p', 'p', 'l', 'e']
>>> word.reverse()
>>> word
['e', 'l', 'p', 'p', 'a']
>>> word.reverse()
>>> word
['a', 'p', 'p', 'l', 'e']
```



## More On Lists

- List methods: **copy** method
  - Creates a copy of the list.
  - But why don't we use the assignment operator = instead of **copy**? Let's talk about it..

```
>>> word1 = ['a', 'p', 'p', 'l', 'e']
>>> word2 = word1
>>> word2
['a', 'p', 'p', 'l', 'e']
>>> word2.append('a')
>>> word2
['a', 'p', 'p', 'l', 'e', 'a']
>>> word1
['a', 'p', 'p', 'l', 'e', 'a']
```

- Although we have appended an 'a' to word2, word1 has also been affected.
- This is why you need to create a copy of the list in case you want two separate word lists.



## More On Lists

- List methods: **copy** method (cont.)
  - Creates a copy of the list.

```
>>> word1 = ['a', 'p', 'p', 'l', 'e']
>>> word2 = word1.copy()
>>> word2.append('a')
>>> word2
['a', 'p', 'p', 'l', 'e', 'a']
>>> word1
['a', 'p', 'p', 'l', 'e']
```



## More On Lists

- List methods: **copy** method (cont.)
  - Creates a copy of the list.

```
>>> word1 = ['a', 'p', 'p', 'l', 'e']
>>> word2 = word1[:]                                # alternative to word1.copy()
>>> word2.append('a')
>>> word2
['a', 'p', 'p', 'l', 'e', 'a']
>>> word1
['a', 'p', 'p', 'l', 'e']
```



## More On Lists

- List methods: **clear** method
  - Clears the list

```
>>> word = ['a', 'p', 'p', 'l', 'e']
>>> word
['a', 'p', 'p', 'l', 'e']
>>> word.clear()
>>> word
[]
```



## More On Lists

### ➤ **del** statement

- deletes the binding/connection between the variable and its object.
- You can use it to delete any object.
- The difference between **del** and **pop** on elements is that **del** does not return the element after it has been deleted.

```
>>> word = ['a', 'p', 'p', 'l', 'e']
>>> del word[1]
>>> word
['a', 'p', 'l', 'e']
```



## More On Lists

- **del** statement
  - deletes the binding/connection between the variable and its object.

```
>>> word1 = ['a', 'p', 'p', 'l', 'e']
>>> word2 = word1
>>> del word1
>>> word2
['a', 'p', 'p', 'l', 'e']
>>> word1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'word1' is not defined
```



## More On Lists

### ➤ **del** statement

- deletes the binding/connection between the variable and its object.
- You can use **del** when you have used a built-in function name as a variable name, and then you want to use that built-in function in your code.

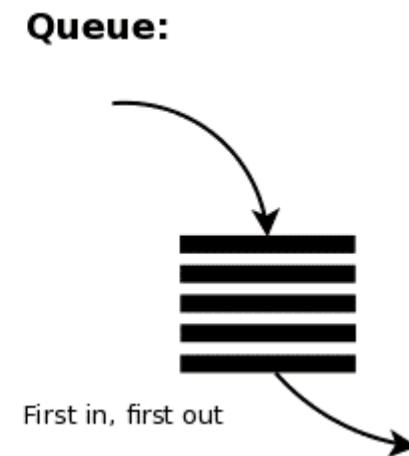
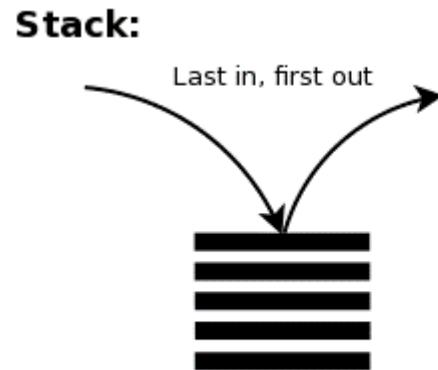
```
>>> result = ('a', 'letter')
>>> value, type = result
>>> print(value, type)
a letter
>>> print(type(value))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
>>> del type
>>> print(type(value))
<class 'str'>
```



## More On Lists

### ➤ Stacks & Queues

- **Stack:** LIFO (Last In, First Out).
- **Queue:** FIFO (First In, First Out).





# More On Lists

- **Stacks using Lists**
  - Lists are can easily be used to work as a stack

```
>>> stack = []
>>> stack.append("book1")
>>> stack.append("book2")
>>> stack.append("book3")
>>> stack
['book1', 'book2', 'book3']
>>> stack.pop()
'book3'
>>> stack.pop()
'book2'
>>> stack.append("book4")
>>> stack
['book1', 'book4']
```



SECURITY FLAME

# More On Lists

## ➤ Queues using Lists

- Traditional lists can also be used as queue, but they might not be efficient.
- We can use collections.deque that has been designed for this purpose

```
>>> from collections import deque
>>> queue = deque([])
>>> queue.append("book1")
>>> queue.append("book2")
>>> queue.append("book3")
>>> queue
deque(['book1', 'book2', 'book3'])
>>> queue.popleft()
'book1'
>>> queue.popleft()
'book2'
>>> queue.append("book4")
>>> queue
deque(['book3', 'book4'])
```



## More On Sets

- Set methods: **add/update** methods
  - **add** adds an item to the set
  - **update** takes items of iterables and adds them to the set

```
>>> s = {0, 1, 2}
>>> s.add(3)
>>> s
{0, 1, 2, 3}
>>> s.update([4, 5])
>>> s.update((6, 7), {8, 9})
>>> s
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```



## More On Sets

- Set methods: **discard/remove** methods
  - **discard** removes an item, and if the item is not present, it won't return anything.
  - **remove** removes an item, and if the item is not present, it will raise an error (KeyError).

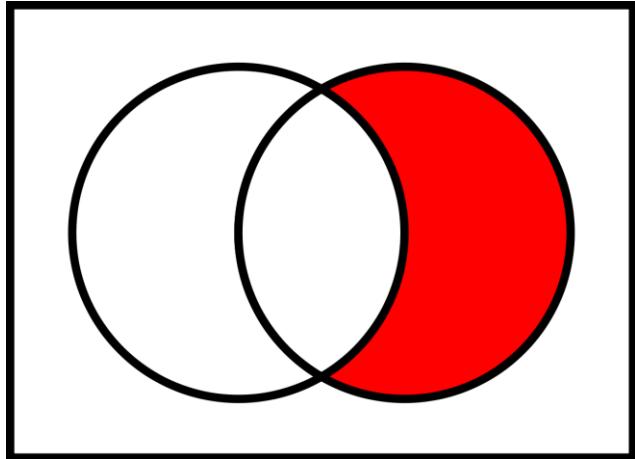
```
>>> s = {1, 2, 3, 4, 5, 6}
>>> s.discard(1)
>>> s
{2, 3, 4, 5, 6}
>>> s.discard(8)
>>> s.remove(2)
>>> s.remove(8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 8
>>> s
{3, 4, 5, 6}
```



SECURITY FLAME

## More On Sets

- Set methods: **difference/difference\_update** methods
  - **difference** returns the difference between two or more sets.
  - **difference\_update** removes all shared items with other sets.



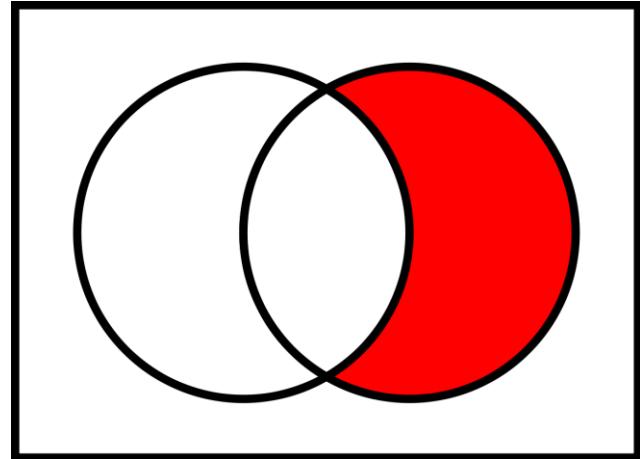
```
>>> s1 = {0, 1, 3}
>>> s2 = {4, 2, 3}
>>> s1.difference(s2) # equivalent to s1 - s2
{0, 1}
>>> s1 - s2
{0, 1}
>>> s2.difference(s1) # equivalent to s2 - s1
{2, 4}
>>> s2 - s1
{2, 4}
```



SECURITY FLAME

## More On Sets

- Set methods: **difference/difference\_update** methods (cont.)
  - **difference** returns the difference between two or more sets.
  - **difference\_update** removes all shared items with other sets.



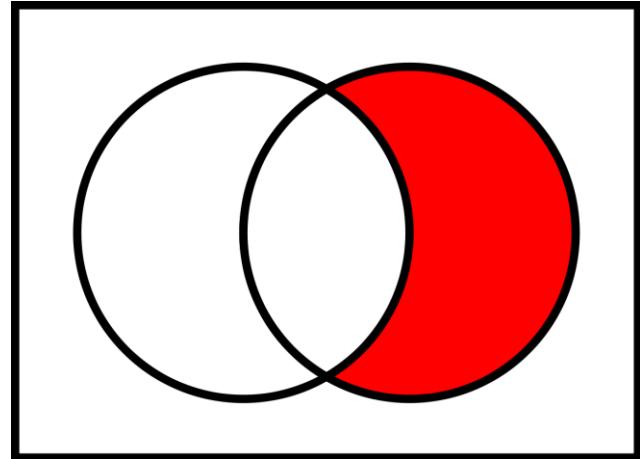
```
>>> s1 = {0, 1, 3}
>>> s2 = {4, 2, 3}
>>> s3 = {8, 1, 3}
>>> s1.difference(s2, s3)
{0}
>>> s2.difference(s1, s3)
{2, 4}
>>> s3.difference(s1, s2)
{8}
```



SECURITY FLAME

## More On Sets

- Set methods: **difference/difference\_update** methods (cont.)
  - **difference** returns the difference between two or more sets.
  - **difference\_update** removes all shared items with other sets.

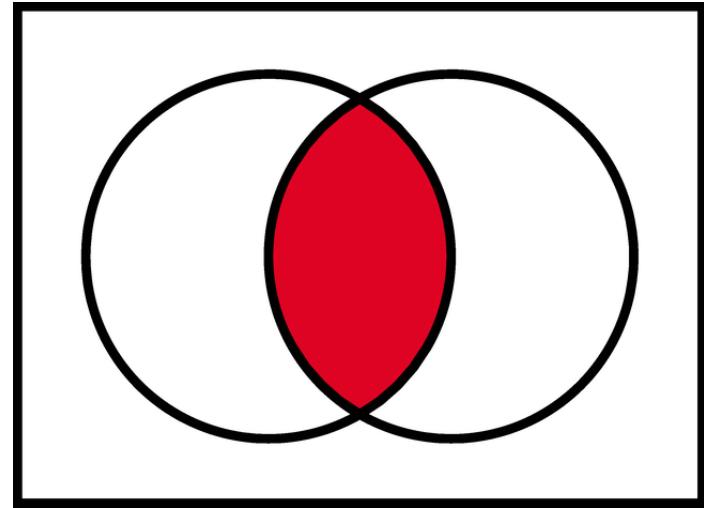


```
>>> s1 = {0, 1, 3}
>>> s2 = {4, 2, 3}
>>> s3 = {8, 1, 3}
>>> s2.difference_update(s1, s3)
>>> s2
{2, 4}
```



## More On Sets

- Set methods: **intersection/intersection\_update** methods
  - **intersection** returns the intersection of the sets.
  - **intersection\_update** updates the set with value of the intersection between the current set and other sets..



```
>>> s1 = {0, 1, 3}
>>> s2 = {4, 2, 3}
>>> s3 = {8, 1, 3}
>>> s1.intersection(s2)
{3}
>>> s1 & s2
{3}
>>> s1.intersection(s3)
{1, 3}
>>> s1 & s3
{1, 3}
```

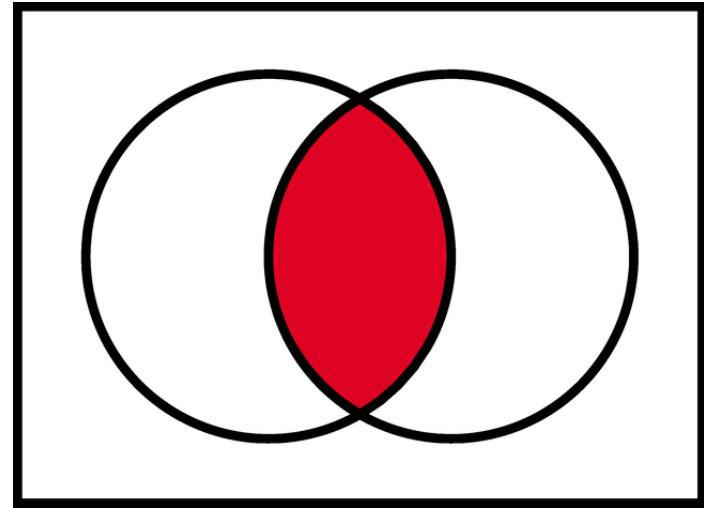


SECURITY FLAME

## More On Sets

- Set methods: **intersection/intersection\_update** methods  
(cont.)
  - **intersection** returns the intersection of the sets.
  - **intersection\_update** updates the set with value of the intersection between the current set and other sets..

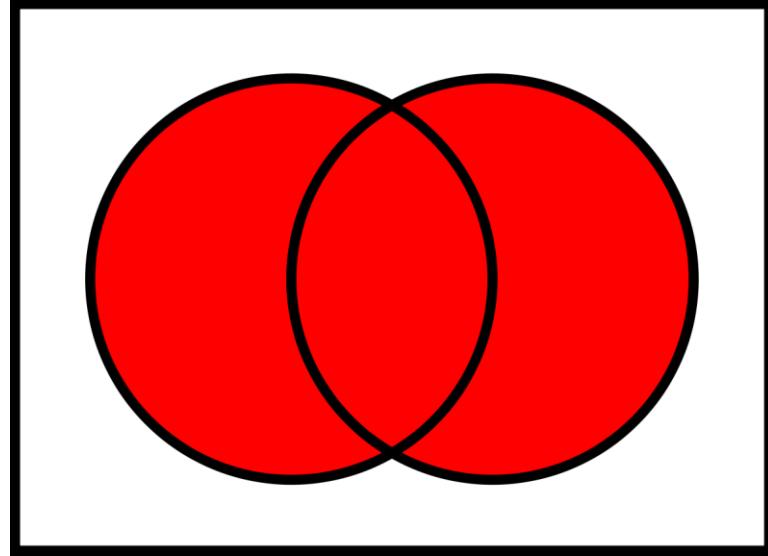
```
>>> s1 = {0, 1, 3}
>>> s2 = {4, 2, 3}
>>> s3 = {8, 1, 3}
>>> s1 & s2 & s3
{3}
>>> s1.intersection_update(s2, s3)
>>> s1
{3}
```





## More On Sets

- Set methods: **union** method
  - **union** return the union of sets as a new set.



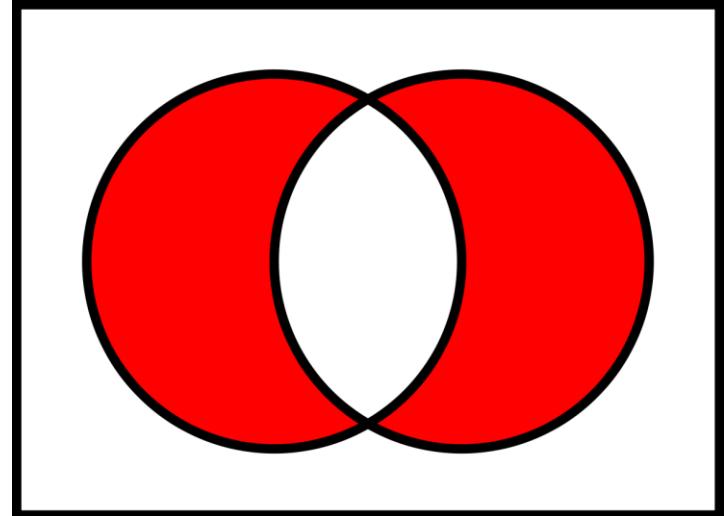
```
>>> s1 = {0, 1, 3}  
>>> s2 = {4, 2, 3}  
>>> s3 = {8, 1, 3}  
>>> s1.union(s2)    # equivalent to s1 | s2  
{0, 1, 2, 3, 4}  
>>> s1 | s2  
{0, 1, 2, 3, 4}  
>>> s1.union(s2, s3) # equivalent to s1 | s2 | s3  
{0, 1, 2, 3, 4, 8}  
>>> s1 | s2 | s3  
{0, 1, 2, 3, 4, 8}
```



SECURITY FLAME

## More On Sets

- Set methods: **symmetric\_difference** / **symmetric\_difference\_update** methods
  - **symmetric\_difference** returns the symmetric difference between two sets.
  - **symmetric\_difference\_update** updates the set to the value of the symmetric difference of itself and another



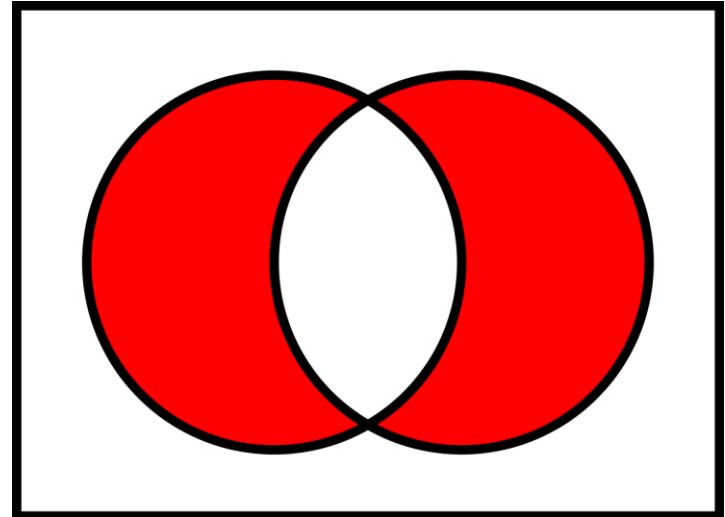
```
>>> s1 = {0, 1, 3}
>>> s2 = {4, 2, 3}
>>> s3 = {8, 1, 3}
>>> s1.symmetric_difference(s2) # equivalent to s1 ^ s2
{0, 1, 2, 4}
>>> s1 ^ s2
{0, 1, 2, 4}
>>> s1.symmetric_difference(s3) # equivalent to s2
{8, 0}
>>> s1 ^ s3
{8, 0}
```



SECURITY FLAME

## More On Sets

- Set methods: **symmetric\_difference** / **symmetric\_difference\_update** methods (cont.)
  - **symmetric\_difference** returns the symmetric difference between two sets.
  - **symmetric\_difference\_update** updates the set to the value of the symmetric difference of itself and another



```
>>> s1 = {0, 1, 3}
>>> s2 = {4, 2, 3}
>>> s3 = {8, 1, 3}
>>> s1.symmetric_difference_update(s2)
>>> s1
{0, 1, 2, 4}
>>> s1.symmetric_difference_update(s3)
>>> s1
{0, 2, 3, 4, 8}
```



## More On Sets

- Set methods: **pop** method
  - **pop** removes and returns a random/arbitrary element from the set. If the set is empty, it will raise an exception (**KeyError**).

```
>>> s = {5, 8, 2}
>>> s.pop()
8
>>> s
{2, 5}
>>> s.pop()
2
>>> s.pop()
5
>>> s = set()
>>> s.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
```



# Writing Modules

- A module is a collection of python code put together in a file (with .py extension).

```
# info.py

def get_info(d: dict):
    for name, number in d.items():
        print(name, number)

def get_names(d: dict):
    for name in d.keys():
        print(name)

def get_numbers(d: dict):
    for number in d.values():
        print(number)
```



# Writing Modules

- Your first module
  - Let's create a file called "info.py" and put the following python code into it

```
# info.py

def get_info(d: dict):
    for name, number in d.items():
        print(name, number)

def get_names(d: dict):
    for name in d.keys():
        print(name)

def get_numbers(d: dict):
    for number in d.values():
        print(number)
```



# Writing Modules

- Importing your module
  - Now you can launch the Python interpreter and import it
  - Make sure that you have opened the Python interpreter in same working directory as your newly created info.py module

```
>>> import info
>>> table = {"Mohammed": 90909090, "Khalid": 91919191, "Salim": 92929292}
>>> info.get_info(table)
Mohammed 90909090
Khalid 91919191
Salim 92929292
>>> info.get_numbers(table)
90909090
91919191
92929292
>>> info.get_names(table)
Mohammed
Khalid
Salim
```



# Writing Modules

- When you import a module called “info”, Python search for it in this order
  - It checks the current working directory if there’s “info” module.
  - If not, then Python checks if there’s a built-in module called “info”.
- You can import a specific function from a module:

```
>>> from info import get_names
>>> table = {"Mohammed": 90909090, "Khalid": 91919191, "Salim": 92929292}
>>> get_names(table)
Mohammed
Khalid
Salim
```



# Writing Modules

- You can import several functions from a module at once

```
>>> from info import get_names, get_numbers  
>>> ...
```

- You can import “everything” from a module

```
>>> from info import *  
>>> ...
```

- You can import the module with another name

```
>>> import info as ifo  
>>> table = {"Mohammed": 90909090, "Khalid": 91919191, "Salim": 92929292}  
>>> ifo.get_names(table)  
Mohammed  
Khalid  
Salim
```



# Writing Modules

- You can import a function from the module with another name

```
>>> from info import get_names as gnames
>>> table = {"Mohammed": 90909090, "Khalid": 91919191, "Salim": 92929292}
>>> gnames(table)
Mohammed
Khalid
Salim
```

- You can import several functions from the module with other names:

```
>>> from info import get_names as gnames, get_numbers as gnums
>>> gnames(table)
Mohammed
Khalid
Salim
>>> gnums(table)
90909090
91919191
92929292
```



# Writing Modules

- `__name__` property
  - The `__name__` property of a module is the module name.

```
>>> import info  
>>> info.__name__  
'info'
```

- The value of the main module's `__name__` is `__main__`

```
>>> __name__  
'__main__'
```

- Where can this be useful?



# Writing Modules

- We will modify our module to include some more code

```
# info.py

def get_info(d: dict):
    ...

if __name__ == "info":
    print("I have been imported!")
elif __name__ == "__main__":
    print("I have been run directly")
    d = {"Test": 12345678}
    get_info(d)
```

Add  
this

- Now if we try to import that module again:

```
>>> import info
I have been imported!
```



# Writing Modules

- If we try to run it directly

```
python info.py
I have been run directly
Test 12345678
```

- Many Python scripts use this way to define the behavior taken if the script is run directly (i.e., not imported):

```
def main():
    do_something()
    ...

if __name__ == "__main__":
    main()                      # start the script only if it's run directly.
```



# The `dir` Function

- `dir` is a built-in function that returns the names and attributes available in an object.
  - Without an argument, `dir` will list all available names in current scope (we'll talk about scopes later).
  - With an argument, `dir` will list all available names in the target scope.

```
>>> import info
I have been imported!
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'info']
>>> dir(info)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'get_info', 'get_names', 'get_numbers']
```



# Standard Modules

## ➤ sys

- You can get arguments supplied in command line using **sys** module

```
import sys
length_of_args = len(sys.argv)
if(length_of_args >= 3):
    arg0 = sys.argv[0] # name of file
    arg1 = sys.argv[1]
    arg2 = sys.argv[2]
    print(f"Name of file: {arg0}")
    print(f"Argument 1: {arg1}")
    print(f"Argument 2: {arg2}")
    print(f"Length of arguments: {length_of_args}")
else:
    print("Please supply some more arguments.)")
```

- If we run the program through command line

```
python test.py first_argument second_argument
Name of file: test.py
Argument 1: first_argument
Argument 2: second_argument
Length of arguments: 3
```



# Standard Modules

- **sys** (cont.)
  - You can also change Python interpreter prompts

```
>>> sys.ps1
'>>> '
>>> sys.ps2
'...'
>>> sys.ps1 = "-> "
-> print("Hello!")
Hello!
```



# Standard Modules

## ➤ **math**

- The **math** module can help you out when dealing with math problems

```
>>> import math
>>> math.cos(0)
1.0
>>> math.pi
3.141592653589793
>>> math.factorial(5) # 5!
120
```



# Standard Modules

## ➤ base64

- **base64** module contains functions that can help you encode/decode data to/from different encodings such as base16, base32, and base64 algorithms

```
>>> import base64  
>>> base64.b64encode(b"Hello there!")  
b'SGVsbG8gdGhlcml0eA=='  
>>> output = base64.b64encode(b"Hello there!")  
>>> base64.b64decode(output)  
b'Hello there!'
```



# Standard Modules

## ➤ **urllib**

- **urllib** module contains functions that can help you out when trying to make HTTP requests.

```
>>> import urllib.request
>>> res = urllib.request.urlopen("https://google.com")
>>> html = res.read()
>>> print(html[:15])
b'<!doctype html>'
>>> res.code
200
>>> res.msg
'OK'
>>> res.getheaders()
[('Set-Cookie', '1P_JAR=2021-11-28-17; ...)]
```



# Standard Modules

## ➤ json

- **json** is a format used to interchange data usually between a server and a client.
- **json** module contains functions that can help encode/decode Python data types to/from readable strings.

```
>>> import json
>>> d = {"Name": "Mohammed", "Age": 25}
>>> json.dumps(d)
'{"Name": "Mohammed", "Age": 25}'
>>> dump_str = json.dumps(d)
>>> json.loads(dump_str)
{'Name': 'Mohammed', 'Age': 25}
>>> load_d = json.loads(dump_str)
>>> type(load_d)
<class 'dict'>
```



# Standard Modules

## ➤ random

- **random** module can be helpful when you need randomness in your python script (not for security purposes).

```
>>> import random
>>> l = [1, 2, 3, 4, 5, 6]
>>> random.choice(l)
2
>>> random.choice(l)
5
>>> random.choice(l)
6
>>> random.randint(0, 500)
397
>>> random.randint(0, 500)
211
```



# Standard Modules

## ➤ secrets

- **secrets** module is used to generate cryptographically strong random numbers that can be used when dealing with passwords, tokens, and account authentication.

```
>>> import secrets
>>> l = [1, 2, 3, 4, 5, 6]
>>> secrets.choice(l)
4
>>> secrets.choice(l)
6
>>> secrets.token_urlsafe(32)
'F5SEGN_8eWqSUFh-nn0fnh18xgX1St1ge-2GobzDZNk'
```



# Standard Modules

## ➤ re

- re module provides regular expression matching operations.

```
>>> import re
>>> data = '<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.15.0/css/all.min.css"><a href="https://twitter.com/D4rkness_14" target="_blank" class="fab fa-twitter fa-2x mx-1 my-2"></a>'
>>> re.findall("href=['\"]?([^\'\"]]+)", data)
['https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.15.0/css/all.min.css', 'https://twitter.com/D4rkness_14']
```



# Standard Modules

- **string**
  - **string** module provide string constants and useful string operations

```
>>> import string
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.digits
'0123456789'
>>> string.punctuation
'!#$%&\'()*+,-./:;=>?@[\\]^_`{|}~'
>>> t = string.Template("$name will buy this for $$100")
>>> t.substitute(name="Khalid")
'Khalid will buy this for $100'
```



# Standard Modules

- **subprocess**
  - **subprocess** allows to execute commands & launch new processes.

```
>>> import subprocess  
>>> output = subprocess.check_output("calc")
```

- On windows, this will pop up the calculator.



# Creating Packages

## ➤ Packages

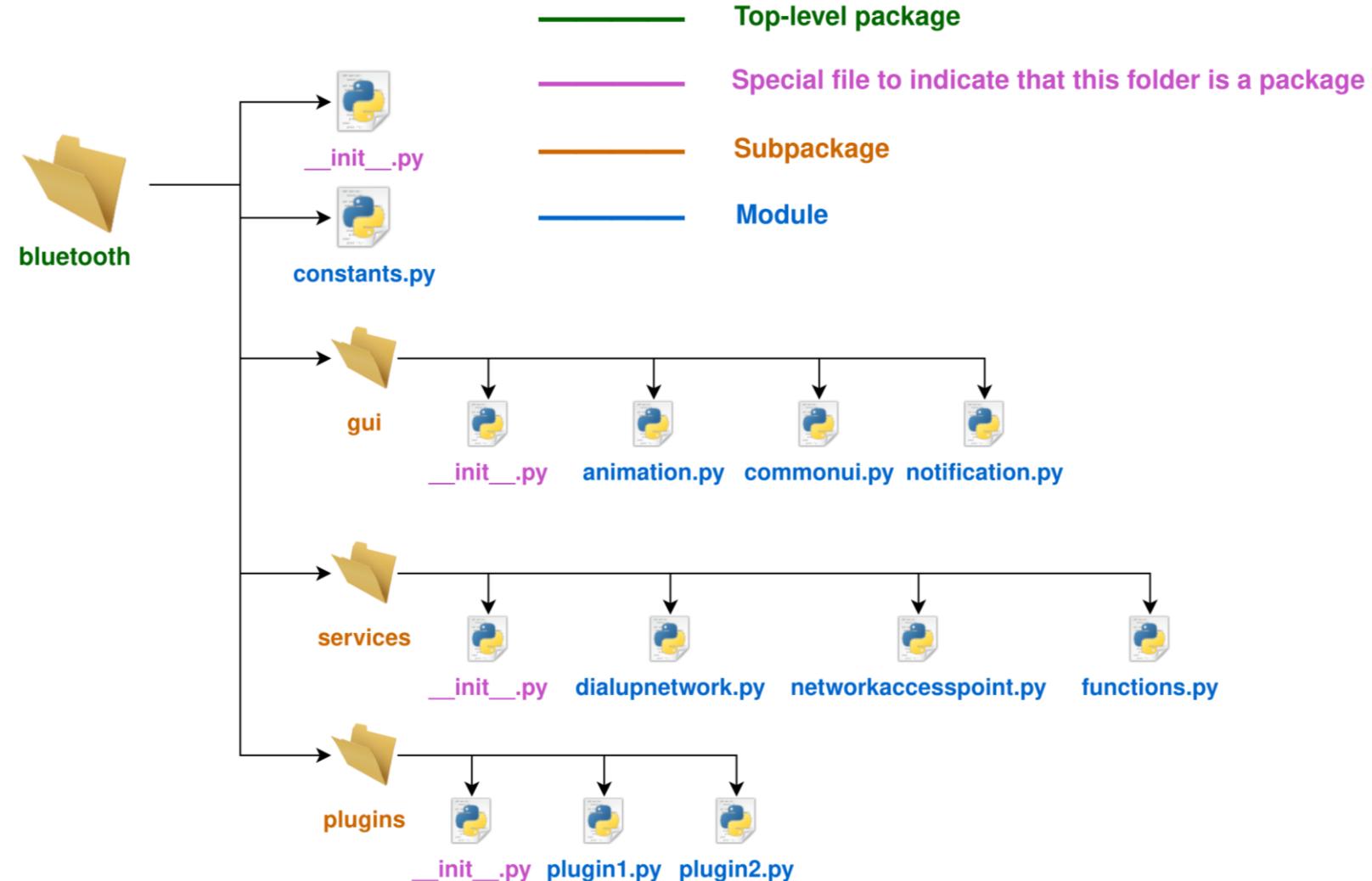
- Packages are a way of organizing modules.
- Modules can be accessed using “dotted module names”.

```
bluetooth
    __init__.py
    constants.py
    gui
        __init__.py
        animation.py
        commonui.py
        notification.py
    plugins
        __init__.py
        plugin1.py
        plugin2.py
    services
        dialupnetwork.py
        functions.py
        __init__.py
        networkaccesspoint.py
```



# Creating Packages

- Packages (cont.)
  - [Download the fictitious Bluetooth package.](#)





SECURITY FLAME

# Creating Packages

## ➤ `__init__.py` file

- This file is used to indicate that a folder is a package.
- It can be as simple as an empty file.
- It is used to initialize the code for the package or set the `__all__` variable as we'll see later.



# Creating Packages

- **Using our package**
  - We can use the package we've downloaded

```
>>> import bluetooth.constants
>>> bluetooth.constants.VERSION
'v0.1'
>>> import bluetooth.gui.animation
The animation module has been imported!
>>> bluetooth.gui.animation.animate1()
animate1 has been called.
>>> import bluetooth.gui.animation as animation
>>> animation.animate2()
animate2 has been called.
```



# Creating Packages

## ➤ Importing \* from a package

- When importing \* from a module, the functions that start with \_ are considered internal functions, so they're not “exported” by default.

```
>>> from bluetooth.gui.animation import *
The animation module has been imported!
>>> animate3()
_internal_animate has been called.
>>> animate4()
_internal_animate has been called.
>>> _internal_animate()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_internal_animate' is not defined
```



# Creating Packages

## ➤ Importing \* from a package

- What happens when we import \* from bluetooth.gui subpackage?

```
>>> from bluetooth.gui import *
>>> animation.animate1()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'animation' is not defined
>>> commonui.func1()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'commonui' is not defined
```

- We notice that submodules (animation, commonui, ...etc) have not been imported.



# Creating Packages

## ➤ `__all__` variable

- `__all__` list is used to define the submodules that will be loaded when trying to import all the packages as we did earlier
- `__all__` list is defined in `__init__.py`
- We will update `bluetooth/gui/__init__.py` to include `__all__` list

```
# bluetooth/gui/__init__.py
__all__ = ["animation", "commonui"]
```



# Creating Packages

- \_\_all\_\_ variable (cont.)
  - Now, when importing \* from **gui** subpackage, only the **animation** and **commonui** modules will be loaded.

```
>>> from bluetooth.gui import *
The animation module has been imported!
The commonui module has been imported!
>>> animation.animate2()
animate2 has been called.
>>> commonui.func1()
commonui.func1 has been called!
>>> notification.push_notification()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'notification' is not defined
```

- When \_\_all\_\_ is not defined, the submodules are not imported when trying to use import \* from a package.
- from package import \* is generally considered to be a bad practice in production environments.
- It's recommended to specify the module you want to use (i.e., from package import module\_x.)



SECURITY FLAME

# Creating Packages

## ➤ Intra-package References

- You can use a module/function from any subpackage.
- Say you need to use the following in “**bluetooth.gui.commonui**” module:
  - The module **bluetooth.gui.animation**
  - The module **bluetooth.plugins.plugin1**
  - The function **bluetooth.services.dialupnetwork.dialup**

```
# bluetooth/gui/commonui.py

from . import animation          # Importing a module
from ..plugins import plugin1    # Importing a module
from ..services.dialupnetwork import dialup # Importing a function

...
```



# Playing With Files

- You can use **open** built-in function to read/write files
- **open** takes several arguments, and we'll talk about two essential ones
  - ❖ First argument is the **file path** to open
  - ❖ Second argument is the **mode** which the file will be opened with (consists of mode, type, modifier)
    - There are several primary modes:
      - **r** open for reading (default)
      - **w** open for writing (after deleting/truncating its content if the file already exists)
      - **x** open for exclusive creation (will fail if the file already exists)
      - **a** appending to file (create a file if it doesn't exist)
    - Types:
      - **t** open in text mode (default)
      - **b** open in binary mode.
    - Modifier
      - **+** (reading and writing)
  - ❖ You can combine one of primary modes, types, the modifier to form a mode argument.
  - ❖ Mode **r** is assumed in case no mode is supplied.
  - ❖ Type **t** is assumed in case no type is supplied.



## Playing With Files (cont.)

- Writing to a file

```
>>> f = open("hello.txt", "w")
>>> f.write("Heloooooooo")
11
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
io.UnsupportedOperation: not readable
>>> f.close()
```



## Playing With Files (cont.)

- Opening a file for reading

```
>>> f = open("hello.txt") # default "rt"
>>> f.read()
'Heloooooooo'
>>> f.write("Hi")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
io.UnsupportedOperation: not writable
>>> f.close()
```



## Playing With Files (cont.)

- Opening a file for reading & writing in text mode
- Good practice to use **with** keyword (the file gets closed automatically).

```
>>> with open("hello.txt", "rt+") as f:  
...     original_output = f.read()  
...     f.write("This is another string..")  
...     f.seek(0)      # go to the begining of the opened file.  
...     current_output = f.read()  
...  
24  
0  
>>> print(original_output)  
Heloooooooo  
>>> print(current_output)  
HelooooooooThis is another string..
```



SECURITY FLAME

## Playing With Files (cont.)

- Opening a file that already exists with mode “x”

```
>>> f = open("hello.txt", "x")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'hello.txt'
```



SECURITY FLAME

## Playing With Files (cont.)

- Opening a file in binary mode

```
>>> with open("hello.txt", "rb") as f:  
...     print(f.read())  
...  
b'HelloooooooThis is another string..'
```



## Playing With Files (cont.)

- File Object methods: **write** method
  - writes to the file stream.

```
>>> f = open("test.txt", "w")
>>> f.write("This is the first line\nThis is the second line\nThis is the third line")
69
>>> f.close()
```



## Playing With Files (cont.)

- File Object methods: **read** method
  - reads the file stream (optionally providing the number of bytes/characters to read)

```
>>> f = open("test.txt", "r")
>>> f.read()
'This is the first line\nThis is the second line\nThis is the third line'
>>> f.close()
```



## Playing With Files (cont.)

- File Object methods: **readline** method
  - reads a line at a time.

```
>>> f = open("test.txt", "r")
>>> f.readline()
'This is the first line\n'
>>> f.readline()
'This is the second line\n'
>>> f.readline()
'This is the third line'
>>> f.readline()
''

>>> f.close()
```



## Playing With Files (cont.)

- File Object methods: **seek** method
  - goes to the nth byte in the file stream.
  - **seek** definition in Python: `seek(cookie, whence=0, /)`
- **cookie** is the offset to which you want to “seek”
- **whence**
  - 0 – start of stream (default)
  - 1 – current stream position (offset can be negative). This is only supported when reading in binary mode.
  - 2 – end of stream (offset usually negative). This is only supported when reading in binary mode.

```
>>> f = open("test.txt", "rb")
>>> f.seek(-4, 2) # go to the 4th byte before the end
65
>>> f.read()
b'line'
>> f.close()
```



## Playing With Files (cont.)

- File Object methods: **tell** method
  - “Tells” you at which offset you currently are.

```
>> f = open("test.txt", "rb")
>>> f.tell()
0
>>> f.seek(-4, 2) # go to the 4th byte before the end
65
>>> f.tell()
65
>>> f.read(1)      # reads only one byte.
b'1'
>>> f.tell()
66
>>> f.close()
```



# Exceptions

- What are exceptions?
  - Errors that occur during execution are called exceptions.
- When do Exceptions occur?
  - When some code results in an unexpected error, an exception is raised.

```
>>> result = 5 + count
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'count' is not defined
>>>
>>> s = "My favorite number is: " + 1234
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>>
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```



## Exceptions (cont.)

- What are exceptions?
  - Errors that occur during execution are called exceptions.
- When do Exceptions occur?
  - When some code results in an unexpected error, an exception is raised.

```
>>> for i in range(-2, 3):  
...     print(8/i)  
...  
-4.0  
-8.0  
Traceback (most recent call last):  
  File "<stdin>", line 2, in <module>  
ZeroDivisionError: division by zero
```



# Exceptions (cont.)

## Handling Exceptions

- Exceptions can be handled through **try** and **except** statements
- Handling the exception raised by division by zero.

```
>>> for i in range(-2, 3):
...     try:
...         print(8/i)
...     except ZeroDivisionError:
...         print("Cannot divide by zero!")
...
-4.0
-8.0
Cannot divide by zero!
8.0
4.0
```



# Exceptions (cont.)

## Handling Exceptions

- Handling the exception raised by trying to read a non-existent file.

```
file_name = input("Enter your file name to read: ")
print("Trying to read the file:", file_name)
try:
    with open(file_name, "r") as f:
        print(f.read())
except FileNotFoundError:
    print(f"File {file_name} was not found!")
```

## Output

```
> python test.py
Enter your file name to read: aaaaa.txt
Trying to read the file: aaaaa.txt
File aaaaa.txt was not found!
```



# Exceptions (cont.)

## Handling Exceptions

- Handling several exceptions:

```
file_name = input("Enter your file name to read: ")
print("Trying to read the file:", file_name)
try:
    with open(file_name, "r") as f:
        print(f.read())
except (FileNotFoundException, PermissionError) as e:
    print(f"File {file_name} cannot be read due to: {e}")
except:
    print("An error occurred while reading the file.")
```

Output

```
$ python test.py
Enter your file name to read: aaaa.txt
Trying to read the file: aaaa.txt
File aaaa.txt cannot be read due to: [Errno 2] No such file or directory: 'aaaa.txt'
```



# Exceptions (cont.)

## Raising Exceptions

- Exceptions can be raised for custom reasons using the **raise** keyword

```
file_name = input("Enter your file name to read: ")
print("Trying to read the file:", file_name)
try:
    if(file_name == "secret.txt"):
        raise PermissionError("You cannot read the secret file!")
    with open(file_name, "r") as f:
        print(f.read())
except (FileNotFoundException, PermissionError) as e:
    print(f"File {file_name} cannot be read due to: {e}")
except:
    print("An error occurred while reading the file.")
```

```
> python test.py
Enter your file name to read: secret.txt
Trying to read the file: secret.txt
File secret.txt cannot be read due to: You cannot read the secret file!
```



# Exceptions (cont.)

## User-defined Exceptions

- You can define your own exception which you can later raise when an unintended event happens
- This defines InvalidWithStandards exception which is raised when an incompatible device (coffee maker) is processed.

```
class InvalidWithStandards(Exception):  
    def __init__(self, ref, expected, received):  
        self.item_reference = ref  
        self.expected_value = expected  
        self.received_value = received  
  
    def __str__():  
        return "ref={}, expected_value={}, received_value={}".format(  
            self.item_reference,  
            self.expected_value,  
            self.received_value)  
  
try:  
    item_ref = "COFFEE-MAKER-1234512345"  
    received_voltage = 120  
    expected_voltage = 240  
  
    if(received_voltage != expected_voltage):  
        raise InvalidWithStandards(item_ref, expected_voltage, received_voltage)  
  
except InvalidWithStandards as e:  
    print("Received an item that's incompatible with our company standards:")  
    print(e)  
  
except:  
    print("Something else happened")
```



SECURITY FLAME

## User-defined Exceptions (cont.)

```
class InvalidWithStandards(Exception):
    def __init__(self, ref, expected, received):
        self.item_reference = ref
        self.expected_value = expected
        self.received_value = received

    def __str__(self):
        return "ref={}, expected_value={}, received_value={}".format(
            self.item_reference,
            self.expected_value,
            self.received_value)

try:
    item_ref = "COFFEE-MAKER-1234512345"
    received_voltage = 120
    expected_voltage = 240

    if(received_voltage != expected_voltage):
        raise InvalidWithStandards(item_ref, expected_voltage, received_voltage)

except InvalidWithStandards as e:
    print("Received an item that's incompatible with our company standards:")
    print(e)

except:
    print("Something else happened")
```

Output

```
Received an item that's incompatible with our company standards:
ref=COFFEE-MAKER-1234512345, expected_value=240, received_value=120
```



# Exceptions (cont.)

## Clean-up after Exceptions

- You can clean-up after the exception using **finally** keyword.
- **finally** block will be executed regardless whether the exception is taken or not.

```
file_name = input("Enter your file name to read: ")
print("Trying to read the file:", file_name)
try:
    with open(file_name, "r") as f:
        print(f.read())
except (FileNotFoundException, PermissionError) as e:
    print(f"File {file_name} cannot be read due to: {e}")
except:
    print("An error occurred while reading the file.")
finally:
    print("Process completed.")
```



# Exceptions (cont.)

## Clean-up after Exceptions (cont.)

- Output (entering a valid file test.txt)

```
Enter your file name to read: test.txt
Trying to read the file: test.txt

Hello there.
I like to program in Python.
Python is both powerful & easy.

Process completed.
```

- Output (entering a non-existing file)

```
Enter your file name to read: aaaa.txt
Trying to read the file: aaaa.txt

File aaaa.txt cannot be read due to: [Errno 2] No such file or directory:
'aaaa.txt'

Process completed.
```



# Namespaces & Scopes

## Namespaces

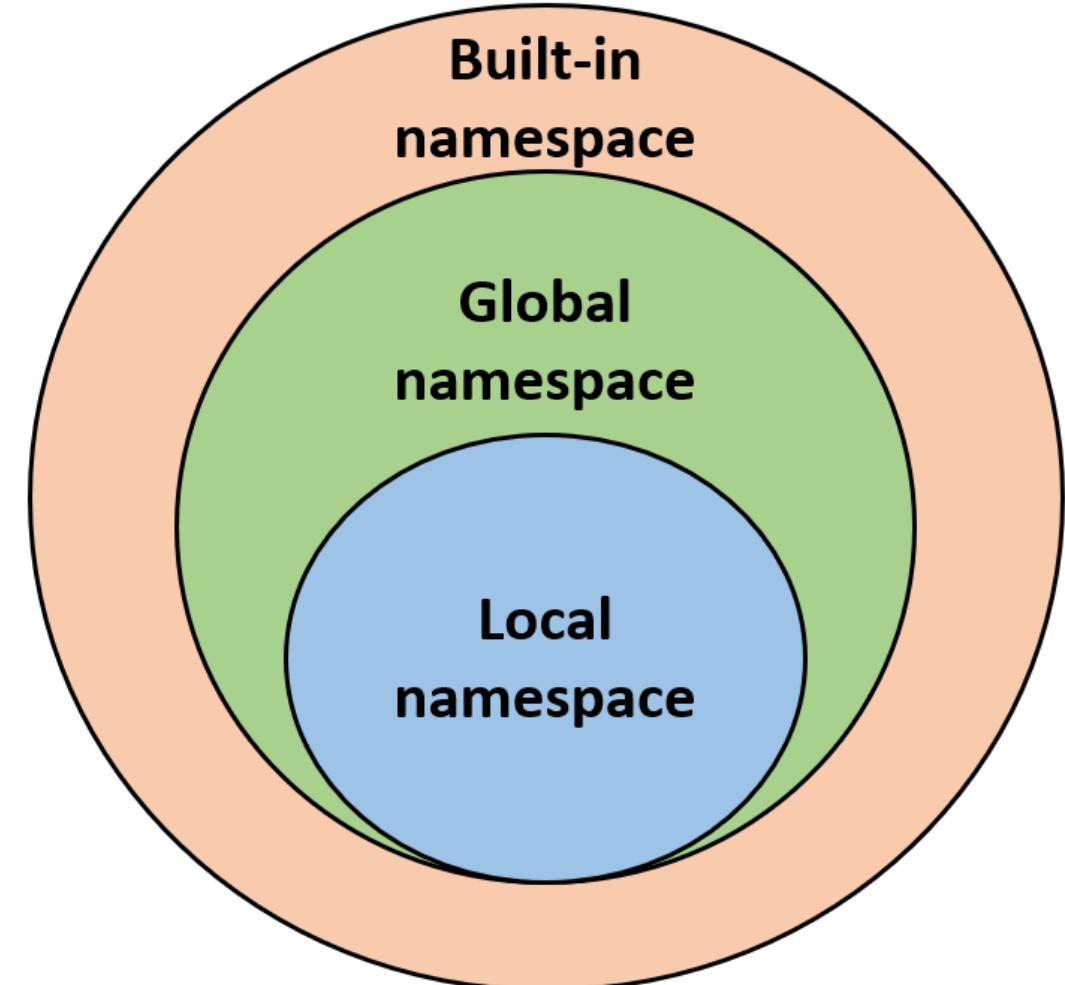
- A “namespace is a mapping from names to objects.”
- Examples

- Built-in namespace
  - Print and other built-in functions
  - Built-in exception names such as **KeyboardInterrupt**
- Global namespace
  - Any global variables such as **file\_name**

```
# test.py
file_name = "myfile.txt"
print(file_name)
```

- Local namespace
  - Any local names to a function such as **x** and **y**

```
def test():
    x = 8
    y = "hi"
    print(x, y)
```



Type of Namespaces

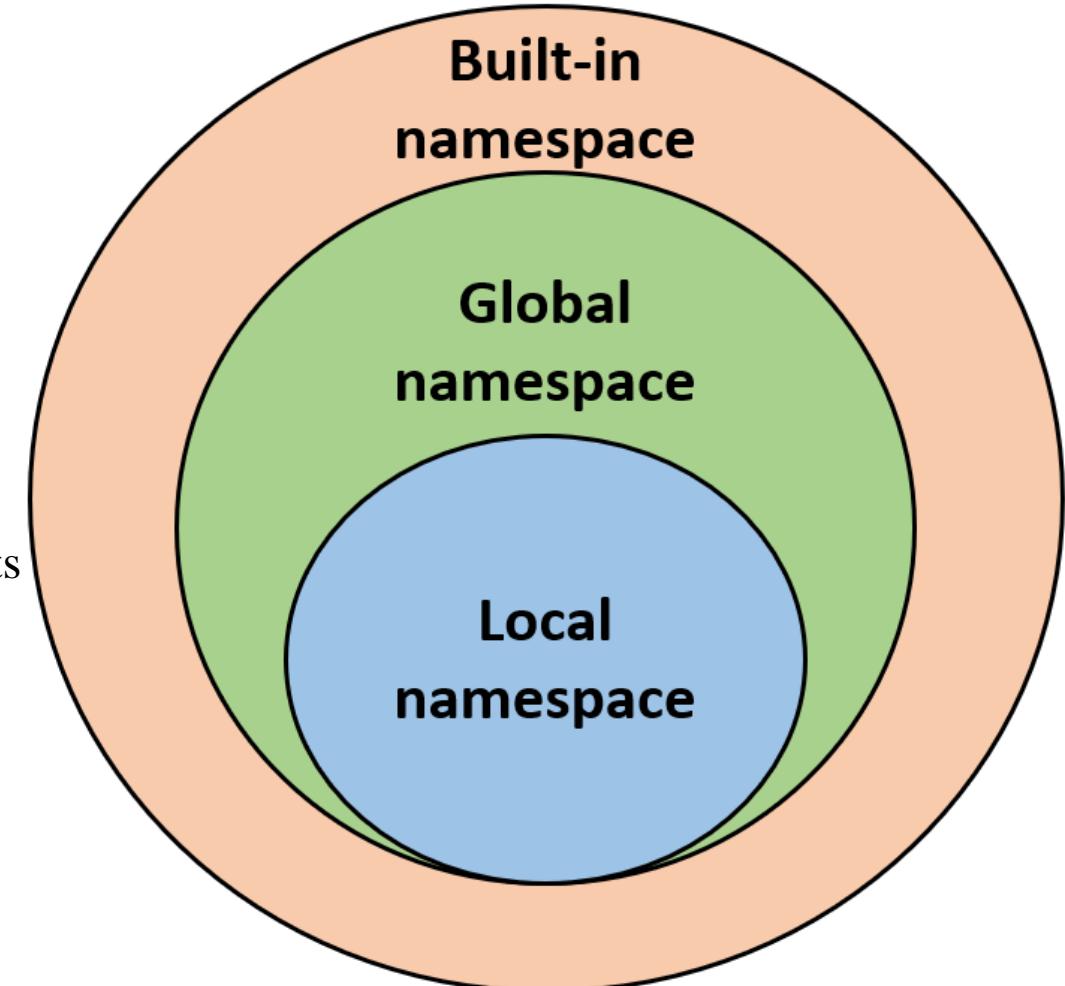


SECURITY FLAME

# Namespaces & Scopes (cont.)

## Namespaces (cont.)

- “Namespaces are created at different moments and have different lifetimes.”
  - Built-in namespace is created when Python starts and is never deleted
  - Global namespace is created when the module starts/imported and is normally deleted when Python quits
  - Local namespace is created when the function is called and is deleted when the function ends through either return or raising an unhandled exception.



Type of Namespaces



# Namespaces & Scopes (cont.)

## Scopes

- A scope is “a textual region of a Python program where namespace is directly accessible.”
  - The scope concept defines the way that Python follows to access names such as: variables, functions, objects, ... etc.
  - If Python doesn’t find a variable in the current scope, it will try to look for it in other scopes (above it)

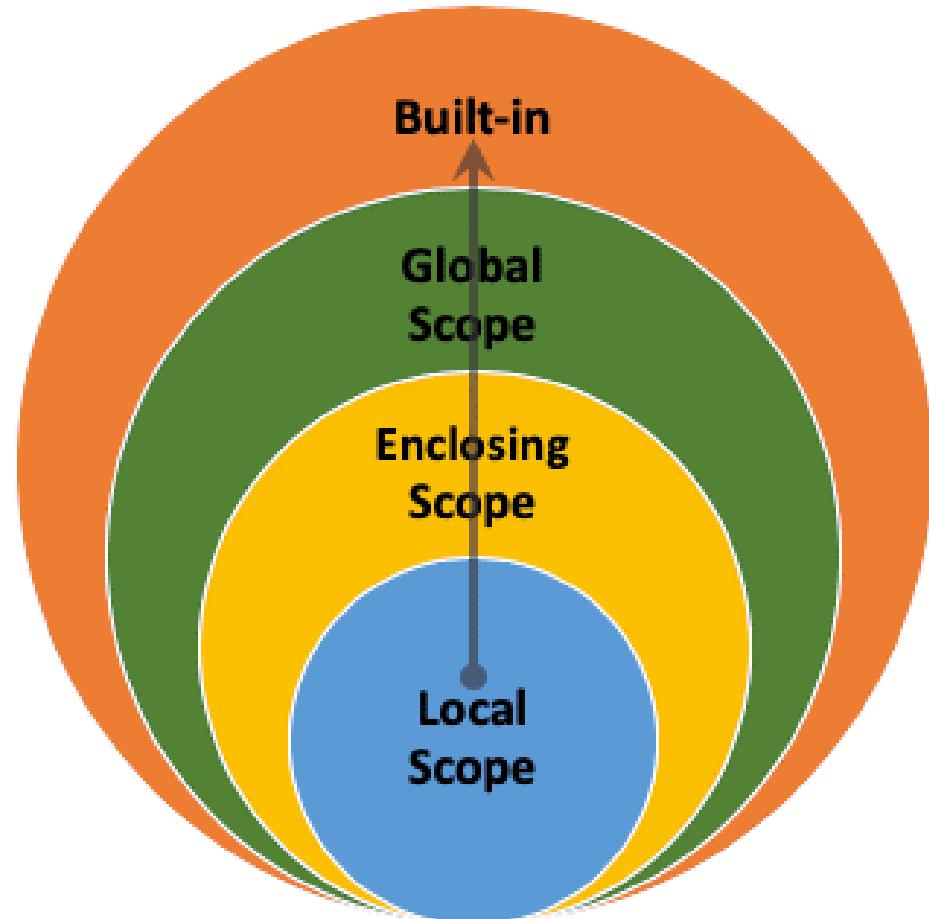
```
name = "Rashid"

def test():
    name = "Mohammed"

test()
print(name)
```

Output ?

Rashid





SECURITY FLAME

# Namespaces & Scopes (cont.)

## Scopes (cont.)

- **global** keyword binds the name to the global scope

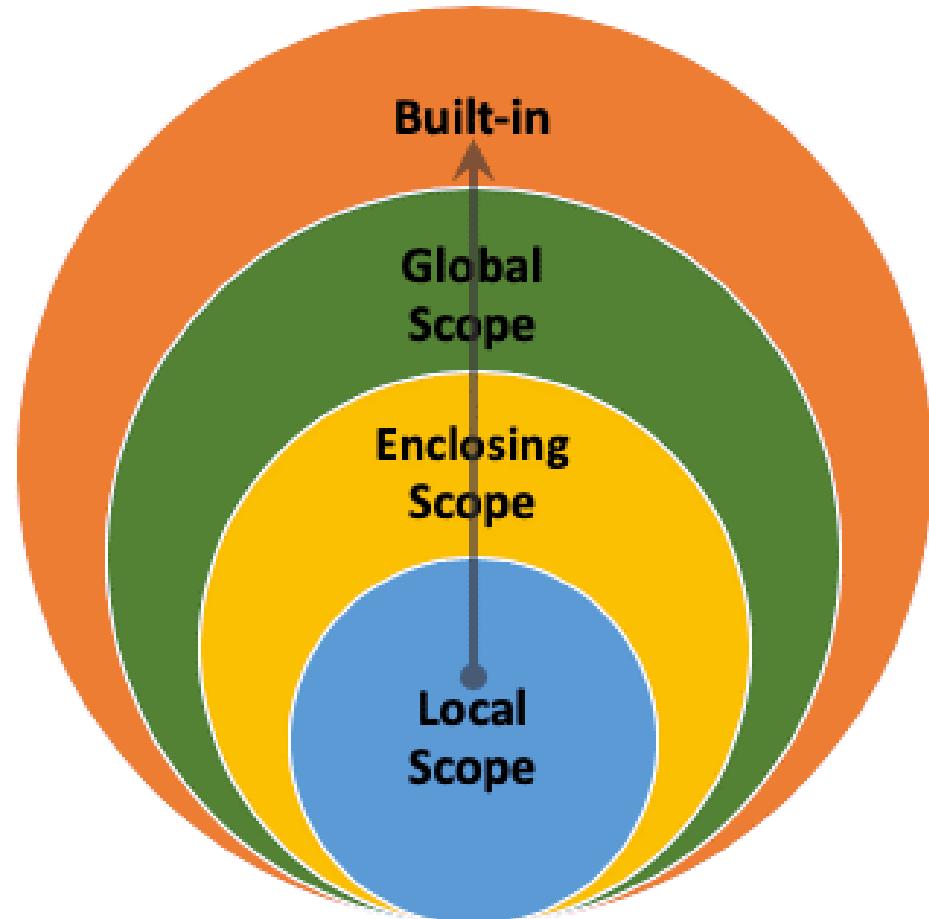
```
name = "Rashid"

def test():
    global name
    name = "Mohammed"

test()
print(name)
```

Output ?

Mohammed





SECURITY FLAME

## Scopes (cont.)

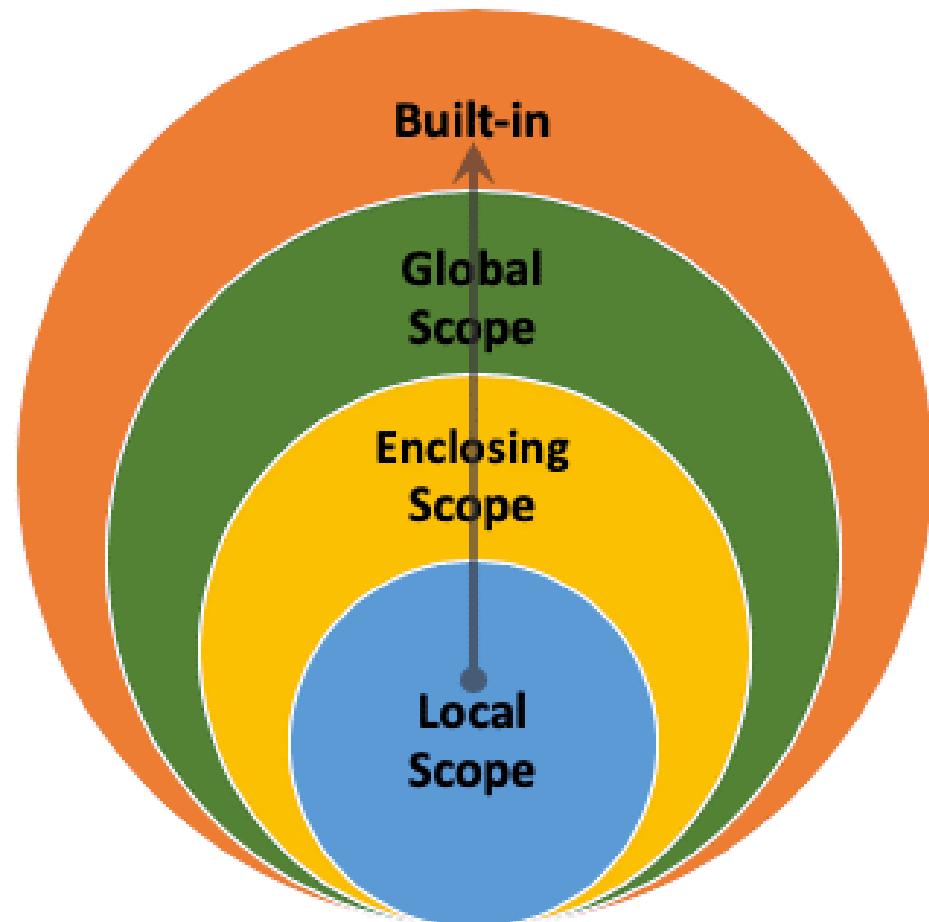
```
def scope_test():
    name = "Rashid"
    def name_local():
        name = "Khalid"

    def name_nonlocal():
        nonlocal name
        name = "Mohammed"

    def name_global():
        global name
        name = "Tariq"

        name_local()
        print("My name after name_local is", name)
        name_nonlocal()
        print("My name after name_nonlocal is", name)
        name_global()
        print("My name after name_global is", name)

    scope_test()
    print("My global name is", name)
```





# Namespaces & Scopes (cont.)

## Scopes (cont.)

```
def scope_test():
    name = "Rashid"
    def name_local():
        name = "Khalid"

    def name_nonlocal():
        nonlocal name
        name = "Mohammed"

    def name_global():
        global name
        name = "Tariq"

    name_local()
    print("My name after name_local is", name)      # My name after name_local is Rashid
    name_nonlocal()
    print("My name after name_nonlocal is", name)    # My name after name_nonlocal is Mohammed
    name_global()
    print("My name after name_global is", name)      # My name after name_global is Mohammed

scope_test()
print("My global name is", name)                  # My global name is Tariq
```

- “To rebind variables found outside of the innermost scope, the **nonlocal** statement can be used”



# Namespaces & Scopes (cont.)

## Scopes summary

Action	Global Code	Local Code	Nested Function Code
Access or reference names that live in the global scope	Yes	Yes	Yes
Modify or update names that live in the global scope	Yes	No (unless declared global)	No (unless declared global)
Access or reference names that live in a local scope	No	Yes (its own local scope), No (other local scope)	Yes (its own local scope), No (other local scope)
Override names in the built-in scope	Yes	Yes (during function execution)	Yes (during function execution)
Access or reference names that live in their enclosing scope	N/A	N/A	Yes
Modify or update names that live in their enclosing scope	N/A	N/A	No (unless declared nonlocal)



# Classes

- Classes are an essential part paradigm OOP (Object-Oriented Programming).
- Classes can be helpful in
  - Code organization
  - Tracking the state of an object
  - Inheritance
  - Code reusability
- In Python, a class is essentially a namespace.
- Class definition
  - A simple empty class can be written

```
class EmptyClass:  
    pass
```



# Classes (cont.)

## ➤ Class Objects

- Creating objects from our empty class

```
class EmptyClass:  
    pass  
  
first_object = EmptyClass()  
second_object = EmptyClass()  
  
first_object.value = 5  
second_object.value = 22  
  
print(first_object.value)      # 5  
print(second_object.value)    # 22
```



# Classes (cont.)

## ➤ Object Methods

- Object methods are functions within an instantiated class.

```
class TestClass:  
    def say_hello(self):  
        print("Hello!")  
  
obj = TestClass()  
  
obj.say_hello() # Hello!
```

- **TestClass.say\_hello** is called a class function
- **obj.say\_hello** is called an object method

- Functions that belong to an object are called methods.
- Class functions have an extra argument (conventionally named **self**).
- **self** argument is used to pass the object itself.

```
...  
obj = TestClass()  
  
obj.say_hello()          # Hello!  
TestClass.say_hello(obj) # Hello!  
  
# obj.say_hello() is equivalent to TestClass.say_hello(obj)
```



SECURITY FLAME

## Classes (cont.)

### ➤ Object Special Methods: `__init__` method

- `__init__` method is used to initialize the object when it's first instantiated

```
class Person:  
    def __init__(self, name):  
        self.name = name  
    def print_my_name(self):  
        print("My name is", self.name)  
  
person1 = Person("Khalid")  
person2 = Person("Mohammed")  
person1.print_my_name()          # My name is Khalid  
person2.print_my_name()          # My name is Mohammed
```

- Here, `__init__` is used to initialize the name that belongs to the object `self` with the argument it receives when `Person` is instantiated



# Classes (cont.)

- **Object Special Methods: `__str__` method**
  - `__str__` method is used to return a string that will represent the object

```
class Person:  
    def __init__(self, name):  
        self.name = name  
    def __str__(self):  
        return f"My name is {self.name}"  
  
person1 = Person("Khalid")  
print(person1)                  # My name is Khalid  
person2 = Person("Mohammed")  
print(person2)                  # My name is Mohammed
```



SECURITY FLAME

## Classes (cont.)

- **Object Special Methods:** `__getitem__`, `__setitem__` methods
  - `__getitem__` is used when you try to index the object
  - `__setitem__` is used when you try to index the object to set it to a different value

```
class Person:  
    def __init__(self, name):  
        self.name = name  
    def __setitem__(self, n, val):  
        self.name = list(self.name)  
        self.name[n] = val  
        self.name = ''.join(self.name)  
    def __getitem__(self, n):  
        return self.name[n]  
    def __str__(self):  
        return self.name  
  
person = Person("Khalid")  
print(person[2])                      # __getitem__ will be called  
person[2] = 's'                         # __setitem__ will be called  
print(person)
```

Output

```
a  
Khslid
```



SECURITY FLAME

# Classes (cont.)

## ➤ Object Special Methods

- There are many more special methods, you can look them up [here](#).



# Classes (cont.)

## ➤ Class variables & Instance variables

- Sometimes you want to share a variable between all objects

```
class Person:  
    country = "Oman"          # shared by all objects/persons  
    def __init__(self, name):  
        self.name = name        # specific to each object  
  
person1 = Person("Khalid")  
print(person1.country)      # Oman  
person2 = Person("Mohammed")  
print(person2.country)      # Oman
```



# Classes (cont.)

## ➤ Inheritance

- You can make your class inherit from another class
- Inheritance from another class makes attributes found in base class/s available in your class.

```
class MalePerson:  
    gender = "Male"  
  
class Person(MalePerson):  
    def __init__(self, name):  
        self.name = name  
  
person = Person("Khalid")  
print(person.gender)          # Male
```



# Classes (cont.)

## ➤ Inheritance (cont.)

- Sometimes you want to extend the functionality of a specific function in the base class

```
class MalePerson:  
    gender = "Male"  
    def get_name_len(self):  
        return len(self.name) # returns an int output  
  
class Person(MalePerson):  
    def __init__(self, name):  
        self.name = name  
    def get_name_len(self):  
        output = super().get_name_len()  
        # returns a string result including int output  
        return f"The length of the name is {output}"  
  
person = Person("Khalid")  
print(person.get_name_len())                      # The length of the name is 6
```



SECURITY FLAME

# Classes (cont.)

## ➤ Multiple Inheritance

- Multiple inheritance can be thought of as inheriting attributes from parent/base classes (depth-first) from left to right.

```
class MyClass(A, B, C):  
    ...
```

- If an attribute is not found in **MyClass**, it'll be searched in **A** and all its base classes. If it's not found there, it'll be searched in **B** and all its base classes, ... and so on.



# Classes (cont.)

## ➤ Multiple Inheritance (cont.)

```
class A:  
    name = "AAAA"  
    def say_hi(self):  
        print("Hi!")  
  
class B:  
    letter = 'B'  
    def say_hi(self):  
        print("Hi!!!!")  
  
class C(A, B):  
    pass  
  
c = C()  
print(c.name, c.letter)      # AAAA B  
c.say_hi()                  # Hi!
```



# Classes (cont.)

## ➤ Multiple Inheritance (cont.)

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

class MalePerson(Person):
    gender = "Male"

class CollegeStudent(Person):
    def __init__(self, name, college):
        self.name = name
        self.college = college

class MaleStudent(CollegeStudent, MalePerson):
    pass

male_student = MaleStudent("Khalid", "SQU")
print(male_student.name, male_student.college, male_student.gender)      # Khalid SQU Male
```



# Decorators

- You might notice the @ syntax in some Python code

```
class Person:  
    def __init__(self, name):  
        self.name = name  
    @property  
    def name_length(self):  
        return len(self.name)  
  
person = Person("Mohammed")  
print(person.name_length)      # output: 8
```

- **property** is a built-in function.
- **property** in the above example is called a decorator (“@” syntax).
- “A decorator in Python is a function that takes another function as its argument, and [usually] returns yet another function.”
- However, In the above example, **name\_length** is no longer a function, but a property.
- A property can be set through a setter, get through a getter, and delete through a deleter.
- **property** definition:

```
property(fget=None, fset=None, fdel=None, doc=None)
```



SECURITY FLAME

## Decorators (cont.)

- property definition:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

- The following code is equivalent to the previous example

```
class Person:  
    def __init__(self, name):  
        self.name = name  
    # @property  
    def name_length(self):  
        return len(self.name)  
    name_length = property(name_length) # equivalent to @property  
  
person = Person("Mohammed")  
print(person.name_length)          # output: 8
```



# Decorators (cont.)

- The following two code snippets are equivalent

```
def call_three_times(func):
    def inner():
        func()
        func()
        func()
    return inner

def say_hi():
    print("Hi!")

say_hi = call_three_times(say_hi)
say_hi()
```

- Without syntactic sugar

```
def call_three_times(func):
    def inner():
        func()
        func()
        func()
    return inner

@call_three_times
def say_hi():
    print("Hi!")

say_hi()
```

- With syntactic sugar

Output

```
Hi!
Hi!
Hi!
```



## Decorators (cont.)

- `time_this_func` is a decorator that calculates the execution time of a target function.

```
from timeit import default_timer as timer

def time_this_func(func):
    def inner():
        start = timer()
        func()
        end = timer()
        elapsed_time = end - start
        print(func.__name__, "took", elapsed_time, "to execute.")
    return inner

@time_this_func
def say_hi():
    print("Hi!")

say_hi()
```

Output

```
Hi!
say_hi took 5.083996802568436e-05 to execute.
```



SECURITY FLAME

# Iterators

- Lists, tuples, strings, ...etc are iterable using the “iterator protocol”.

```
>>> s = "Oman"
>>> for c in s:
...     print(c)
...
0
m
a
n
```

- How does this work?
  - for** loops through the iterator and calls the special method `__next__` over each iteration.



SECURITY FLAME

## Iterators (cont.)

```
>>> s = "Oman"
>>> for c in s:
...     print(c)
...
0
m
a
n
```

```
>>> s = "Oman"
>>> s_iter = iter(s)
>>> s_iter
<str_iterator object at 0x7fdfc699de20>
>>> next(s_iter)
'0'
>>> next(s_iter)
'm'
>>> next(s_iter)
'a'
>>> next(s_iter)
'n'
>>> next(s_iter)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```



# Iterators (cont.)

- We can create a class that implements the “iterator protocol”

```
class Reverse:  
    def __init__(self, data):  
        self.data = data  
        self.index = len(data)  
    def __iter__(self):  
        return self  
    def __next__(self):  
        if(self.index == 0):  
            raise StopIteration  
        self.index = self.index - 1  
        return self.data[self.index]  
  
reverse_word = Reverse("Oman")  
for c in reverse_word:  
    print(c)
```

Output

```
n  
a  
m  
o
```



# Generators

- Generators are a powerful way to create iterators.
- A generator is essentially a function that has a **yield** keyword instead of **return**.
- Previous **Reverse** example can be written as follows

```
def reverse(data):
    for i in range(len(data)-1, -1, -1):
        yield data[i]

for c in reverse("Oman"):
    print(c)
```

- In generators, **\_\_iter\_\_** and **\_\_next\_\_** are created automatically.



SECURITY FLAME

## Generators (cont.)

- **yield** isn't really like a **return**

```
def my_range(upto_n):
    index = 0
    while index < upto_n:
        yield index
        index += 1

for i in my_range(5):
    print(i, end=" ")
print() # 0 1 2 3 4
```



SECURITY FLAME

## Generators (cont.)

- **yield from** is used to yield values from a generator.

```
def my_range(upto_n):
    index = 0
    while index < upto_n:
        yield index
        index += 1

def wrapper():
    yield from my_range(3)
    yield from [ 'a', 'b', 'c' ]
    yield from range(3, 6)
    yield from ( 'd', 'e', 'f' )

for i in wrapper():
    print(i, end=" ") # 0 1 2 a b c 3 4 5 d e f
print()
```



SECURITY FLAME

## Generators (cont.)

- Generators/iterators can save up memory when processing a large set of values since they calculate values when they're requested for.

```
>>> values = list(range(5, 8000000000))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
>>> values = iter(range(5, 8000000000))
```



# Generator Expressions

- Generators can be written in a linear form using parentheses ( and )

```
>>> my_genexpr = (i for i in range(500))
>>> my_genexpr
<generator object <genexpr> at 0x7fdfc68b97b0>
>>>
```

```
>>> word = "Oman"
>>> my_genexpr = (word[i] for i in range(len(word)-1, -1, -1))
>>> ''.join(my_genexpr)
'namO'
```

```
>>> list(zip(range(1, 4), range(4, 7)))
[(1, 4), (2, 5), (3, 6)]
>>> list_of_tuples = list(zip(range(1, 4), range(4, 7)))
>>> my_genexpr = (i*j for i, j in list_of_tuples)
>>> my_genexpr
<generator object <genexpr> at 0x7f087a808740>
>>> sum(my_genexpr)
32
```



# Writing Tests

- Writing tests is crucial during the development process to verify that the code runs as expected.
- **doctest** module
  - The **doctest** module is simple & effortless.
  - Creating a test is as simple as copy-and-pasting the function call along with its results into the docstring (function documentation).
  - In case there's a failure, a clear message will be provided to which test has failed along with expected/got values.

```
import doctest

def square_list(l: list) -> list:
    """Computes the square of each item in the list.

    >>> square_list([4, 5, 6])
    [16, 25, 36]
    """
    for i in range(len(l)):
        l[i] **= 2
    return l

doctest.testmod()
```



SECURITY FLAME

# Writing Tests (cont.)

## ➤ **unittest** module

- **unittest** is not as simple as **doctest**, but it has more advanced capabilities for testing code.
- Many Python programmers use **unittest** to write their tests.

```
import unittest
from mycode import square_list

class MyCodeTest(unittest.TestCase):
    def test_square_list(self):
        self.assertEqual(square_list([4, 5, 6]), [16, 25, 36])
        self.assertEqual(square_list([1.2, 3.8, 5.3]), [1.44, 14.44, 28.09])
        self.assertEqual(square_list([0]), [0])
        with self.assertRaises(TypeError):
            square_list(4, 5)

unittest.main()
```

Output

```
.
-----
Ran 1 test in 0.000s

OK
```

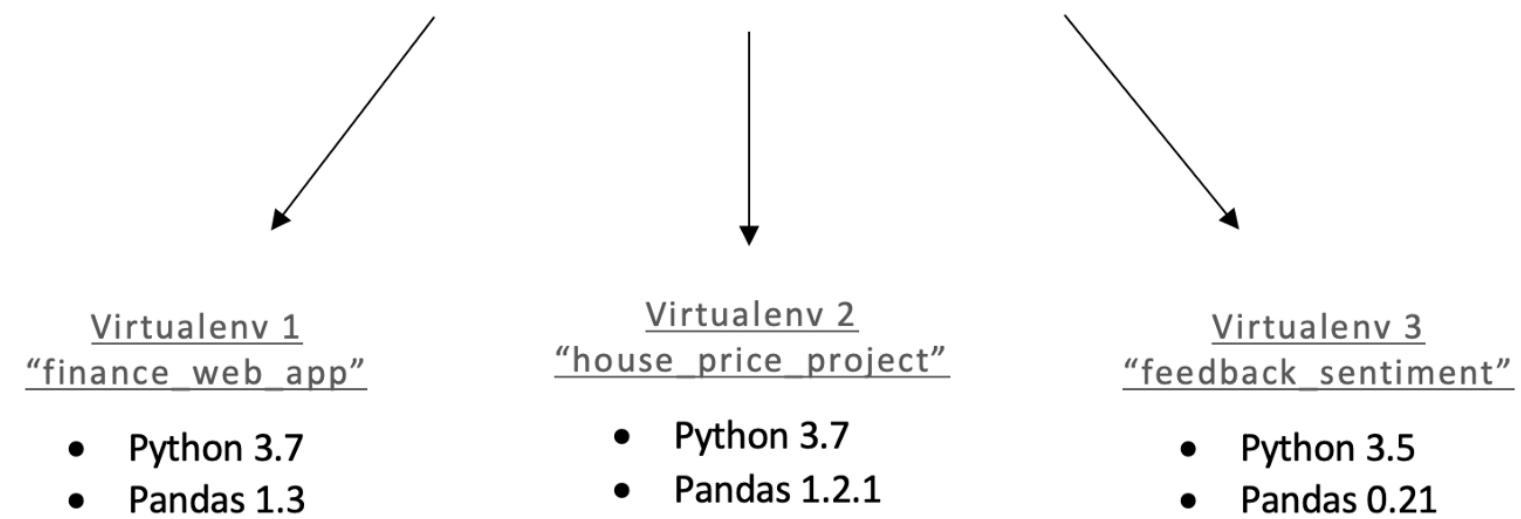


SECURITY FLAME

# Virtual Environments

- A virtual environment is “a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages”.
- Virtual Environments are used to create isolated environments for Python projects.
- Different projects can use different virtual environments that contain different versions of different packages/modules.

# Python Projects





# Virtual Environments (cont.)

## ➤ Creating Virtual Environments

- To create a virtual environment, we can use the **venv** module which is a part of the standard library.

```
python -m venv first-env
```

- This will create a directory “first-env”, which will contain necessary files and a copy of the Python interpreter.
- To work within the virtual environment, you need to activate it.
- On Windows, execute (make sure you’re not running within PowerShell):

```
PS C:\Users\user> cmd
Microsoft Windows [Version 10.0.19043.1165]
(c) Microsoft Corporation. All rights reserved.

C:\Users\user>.\first-env\Scripts\activate.bat

(first-env) C:\Users\user>
```

- On Unix-like, execute:

```
bash-5.1$ source first-env/bin/activate
(first-env) bash-5.1$
```



SECURITY FLAME

# Virtual Environments (cont.)

## ➤ Managing Packages Within The Virtual Environment

- **pip** is used to install, upgrade, and remove packages.
- **pip** will install packages from [pypi.org](https://pypi.org).
- You can use it to install any package to the virtual environment.
- Packages installed within the virtual environment will not be available system-wide.
- Say you want to install a package called **impacket** in the Virtual Environment:

```
(first-env) C:\Users\user>pip install impacket
Collecting impacket
...
    Running setup.py install for impacket ... done
Successfully installed ... impacket-0.9.24 ...


```

```
(first-env) C:\Users\user>python
>>> import impacket
>>>
```

- You can uninstall it

```
(first-env) C:\Users\user>pip uninstall impacket
Found existing installation: impacket 0.9.24
Uninstalling impacket-0.9.24:
...
Proceed (Y/n)? y
Successfully uninstalled impacket-0.9.24
```



SECURITY FLAME

# Virtual Environments (cont.)

## ➤ Managing Packages Within The Virtual Environment (cont.)

- You can install a specific version of impacket:

```
(first-env) C:\Users\user>pip install impacket==0.9.23
Collecting impacket==0.9.23
...
      Running setup.py install for impacket ... done
Successfully installed impacket-0.9.23
```

- You can upgrade it to the latest version:

```
(first-env) C:\Users\user>pip install --upgrade impacket
Requirement already satisfied: impacket in c:\users\user\first-
env\lib\site-packages (0.9.23)
Collecting impacket
...
Installing collected packages: impacket
  Attempting uninstall: impacket
    Found existing installation: impacket 0.9.23
    Uninstalling impacket-0.9.23:
      Successfully uninstalled impacket-0.9.23
    Running setup.py install for impacket ... done
Successfully installed impacket-0.9.24
```



SECURITY FLAME

# Virtual Environments (cont.)

## ➤ Managing Packages Within The Virtual Environment (cont.)

- To show information about a specific package

```
(first-env) C:\Users\user>pip show impacket
Name: impacket
Version: 0.9.24
Summary: Network protocols Constructors and Dissectors
Home-page: https://www.secureauth.com/labs/open-source-tools/impacket
Author: SecureAuth Corporation
Author-email: oss@secureauth.com
License: Apache modified
Location: c:\users\user\first-env\lib\site-packages
Requires: pyasn1, pycryptodomex, pyOpenSSL, six, ldap3, ldapdomaindump,
flask, future, chardet
```

- To show all installed packages in the virtual environment

```
(first-env) C:\Users\user>pip list
Package           Version
-----
...
impacket          0.9.24
pip               21.2.3
setuptools        57.4.0
```



SECURITY FLAME

# Virtual Environments (cont.)

## ➤ Managing Packages Within The Virtual Environment (cont.)

- To show all installed packages in an output suitable for **requirements.txt** (file used to install requirements for projects):

```
(first-env) C:\Users\user>pip freeze
...
impacket==0.9.24
MarkupSafe==2.0.1
pyasn1==0.4.8
pycparser==2.21
...
(first-env) C:\Users\user>pip freeze > requirements.txt
```

- In case you want to install packages from a **requirements.txt** file:

```
(first-env) C:\Users\user>pip install -r requirements.txt
...
Requirement already satisfied: impacket==0.9.24 in c:\users\user\first-
env\lib\site-packages (from -r requirements.txt (line 9)) (0.9.24)
Requirement already satisfied: ldap3==2.9.1 in c:\users\user\first-env\lib\site-
packages (from -r requirements.txt (line 12)) (2.9.1)
...
```



SECURITY FLAME

# Virtual Environments (cont.)

## ➤ Managing Packages Within The Virtual Environment (cont.)

- To deactivate the environment:

```
(first-env) C:\Users\user>deactivate  
C:\Users\user>
```



# Pycache Files

- You might notice sometimes Python creates a folder `__pycache__` in which it creates `.pyc` files.
- Those are Python cache files, which are used to speed up loading modules.
- It caches the modules in the following pattern:

```
__pycache__/<module_name>.cpython-<python_version>.pyc
```

- For example, a module called `info` loaded in Python 3.10 will be cached as follows:

```
__pycache__/info.cpython-310.pyc
```



SECURITY FLAME

# Multiprocessing vs Threading vs Asyncio

- [multiprocessing](#) is a module that can be used to start Python code as another process.
- [threading](#) is a module that can be used to start Python code as a thread.
- [asyncio](#) is a module that can be used to create/start Python code as a coroutine. (added since Python 3.4)



SECURITY FLAME

## Multiprocessing vs Threading vs Asyncio (cont.)

### ➤ Global Interpreter Lock (GIL)

- A Python process **can only utilize one physical thread of your CPU**. It also **cannot use multiple CPU cores**.
- This is because of something called the [Global Interpreter Lock \(GIL\)](#).
- The GIL **prevents threads from causing race conditions** and **ensures thread safety**.
- It is also used to prevent memory leaks when executing several threads since there's only one thread that is effectively executing.
- In essence, the GIL is used to prevent possible problems caused by multithreading.

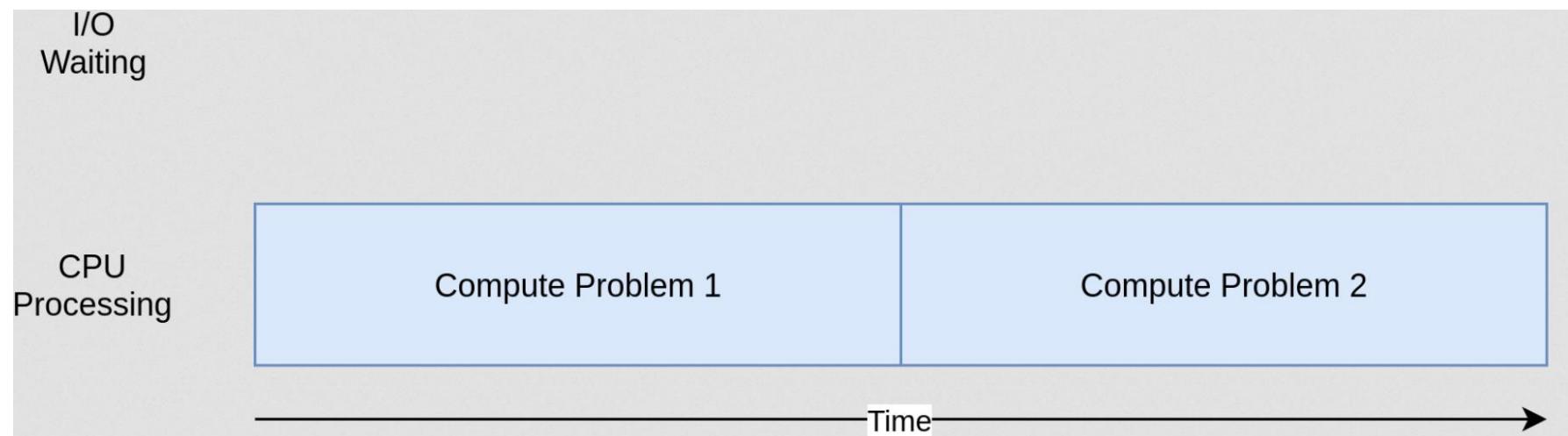


SECURITY FLAME

## Multiprocessing vs Threading vs Asyncio (cont.)

### ➤ CPU-bound vs I/O-bound

A CPU-bound process refers to **the process at which the CPU's most time is spent processing intensively** (i.e., for intensive calculation).



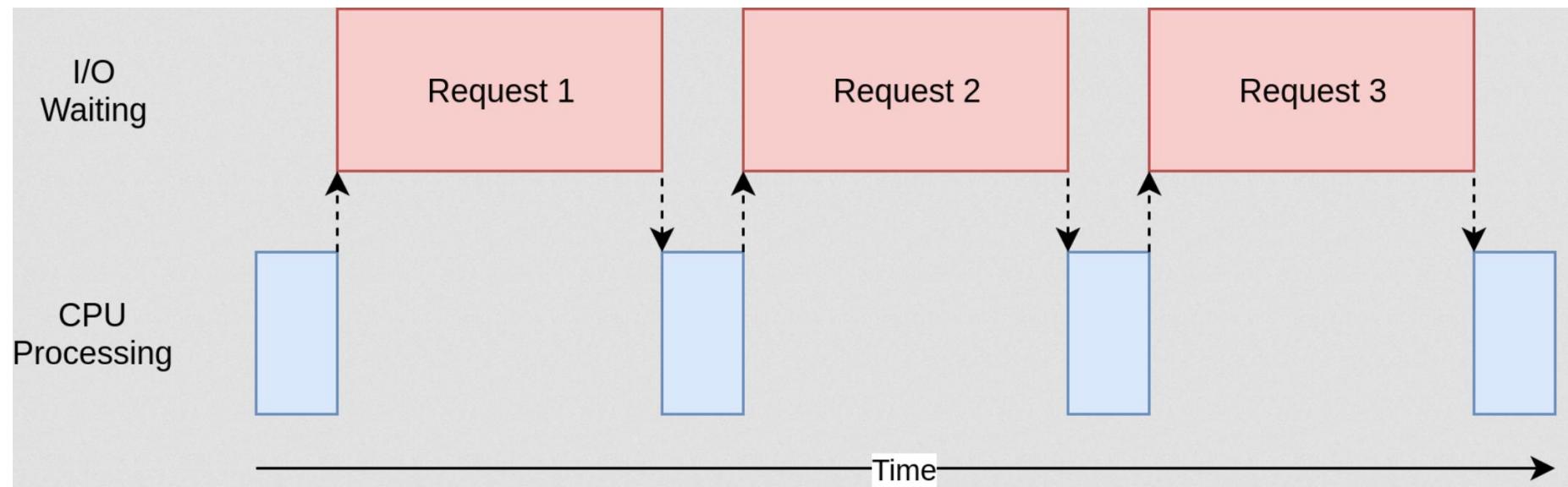


SECURITY FLAME

## Multiprocessing vs Threading vs Asyncio (cont.)

### ➤ CPU-bound vs I/O-bound (cont.)

An I/O-bound process refers to **the process at which the CPU's most time is spent idle waiting for some input to be received** (i.e., waiting for an HTTP response).





SECURITY FLAME

# Multiprocessing vs Threading vs Asyncio (cont.)

## ➤ When to use which?

- For a CPU-bound Python process, the best candidate to use is **multiprocessing** since it starts new processes taking advantage of CPU cores.
- For an I/O bound Python process, the best candidate(s) to use is either **threading** or **asyncio**.
  - In case you have a **fast I/O and a small number of tasks**, you can use **threading**. In **threading**, the OS determines when switch contexts between threads. For a large number of threads, there can be a thread-management overhead.
  - In case you know exactly when to switch your program's execution, you better use **asyncio**. **asyncio** gives you the ability to decide when/where to switch contexts between coroutines. All **asyncio** coroutines run as one thread. Programming with **asyncio** can be more challenging than if it's done with **threading**.



# Multiprocessing vs Threading vs Asyncio (cont.)

## ➤ Summary

Module	multiprocessing	threading	asyncio
Purpose	Create processes	Create threads	Create coroutines
Use case	CPU-bound	Fast I/O-bound (limited number of connections)	Slow I/O-bound (many connections)
Number of threads	As many as each process starts	As many as you create	One thread
Number of threads effectively executing at a time	One thread per process	One thread	One thread



SECURITY FLAME

# Multiprocessing vs Threading vs Asyncio (cont.)

## ➤ multiprocessing

- Executing Python code as three different processes

```
import multiprocessing
import time

def task(name):
    print(f"{name} started")
    time.sleep(1)
    print(f"{name} finished")

def main():
    p1 = multiprocessing.Process(target=task, args=("first task",))
    p2 = multiprocessing.Process(target=task, args=("second task",))
    p3 = multiprocessing.Process(target=task, args=("third task",))
    p1.start(); p2.start(); p3.start()
    p1.join(); p2.join(); p3.join()

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



SECURITY FLAME

# Multiprocessing vs Threading vs Asyncio (cont.)

## ➤ threading

- Executing Python code as three different threads

```
import threading
import time

def task(name):
    print(f"{name} started")
    time.sleep(1)
    print(f"{name} finished")

def main():
    t1 = threading.Thread(target=task, args=("first task",))
    t2 = threading.Thread(target=task, args=("second task",))
    t3 = threading.Thread(target=task, args=("third task",))
    t1.start(); t2.start(); t3.start()
    t1.join(); t2.join(); t3.join()

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



SECURITY FLAME

# Multiprocessing vs Threading vs Asyncio (cont.)

## ➤ **asyncio**

- Executing Python code as three coroutines

```
import asyncio

async def task(name):
    print(f'{name} started')
    await asyncio.sleep(1)
    print(f'{name} finished')

async def main():
    await asyncio.gather(
        task('first task'),
        task('second task'),
        task('third task'),
    )

if __name__ == '__main__':
    asyncio.run(main())
```



# Extra Stuff

## ➤ Python “-i” option

- If have written several lines of code and you made a mistake, it might be tedious to rewrite the whole thing in the Python interpreter.
- You can create a file and run it interactively as follows:

```
l = [1, 2, 3, 4, 5]
n = 5

def multiply_list(l, n):
    for i in range(len(l)):
        l[i] = l[i] * n
    return l
```

- You can create a file and run it **interactively** as follows:

```
python -i test1.py

>>> multiply_list(l, n)
[5, 10, 15, 20, 25]
>>>
```



SECURITY FLAME

# Extra Stuff

## ➤ Python “-c” option

- You can execute Python code through command line as follows:

```
python -c 'name = input("Enter your name: "); print("Hello,", name)'
```

- Make sure that you have the Python code enclosed by quotes.



# Extra Stuff

## ➤ Python “-m” option

- You can run modules directly from command line as we've done when creating virtual environments using **venv** module
- **compileall** module can be used to compile modules to pycache

```
python -m compileall mymodule.py
Compiling 'mymodule.py'...
```

- **cProfile** module is used to calculate the total running time and total function calls

```
python -m cProfile mymodule.py
    13 function calls in 0.000 seconds
```

```
Ordered by: standard name
...
...
```

- **http.server** module is used to calculate the total running time and total function calls

```
python3 -m http.server --bind 127.0.0.1
Serving HTTP on 127.0.0.1 port 8000 (http://127.0.0.1:8000/) ...
```



SECURITY FLAME

# Extra Stuff

## ➤ Walrus operator

- := (since Python 3.8)
- parentheses are mandatory.



```
x = 0
while (x := x+1) < 5:
    print(f"{x} is less than 5")
```

Output

```
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
```



SECURITY FLAME

## Extra Stuff

- **Walrus operator (cont.)**
  - := (since Python 3.8)
  - parentheses are mandatory.



```
with open("data.bin", "rb") as f:  
    while (chunk := f.read(100)):  
        print(chunk)
```



SECURITY FLAME

# Extra Stuff

## ➤ **repr** function

- When debugging code, you probably want to use **repr** within the print.
- Shows you more info about a value and its type.

```
>>> text = "Test\n\nValue"
>>> print(text)
Test

Value
>>> print(repr(text))
'Test\n\nValue'
```



SECURITY FLAME

# Coding Styles

- Use spaces around operators and after commas, but not directly inside bracketing constructs

```
a = f(1, 2) + g(3, 4)
```

- “Use docstrings”
- “Wrap lines so that they don’t exceed 79 characters.”
- “Name your classes and functions consistently; the convention is to use **UpperCamelCase** for classes and **lowercase\_with\_underscores** for functions and methods. Always use self as the name for the first method argument.”
- Rest of them here - you don’t have to follow every single thing ⓘ



SECURITY FLAME

## What's next?

- Pick an idea and start your first project in Python.
  - Hint: there are many frameworks that can help you do cool stuff 😊
- If you'd like more practice, try to join competitive programming platforms such as the following:
  - [LeetCode](#)
  - [HackerRank](#)
  - [Codewars](#)

I hope you enjoyed the ride 😊

If you'd like to contact me for whatever reason, you can find me  
on Twitter [@D4rkness\\_14](#)