PYTHON

PYTHON OVERVIEW

- Preliminaries
- IPython Basics
- Python Language Basics
- Built-in Data Structures
- Functions & Files

CASE STUDY

- Algorithm
- Real cases

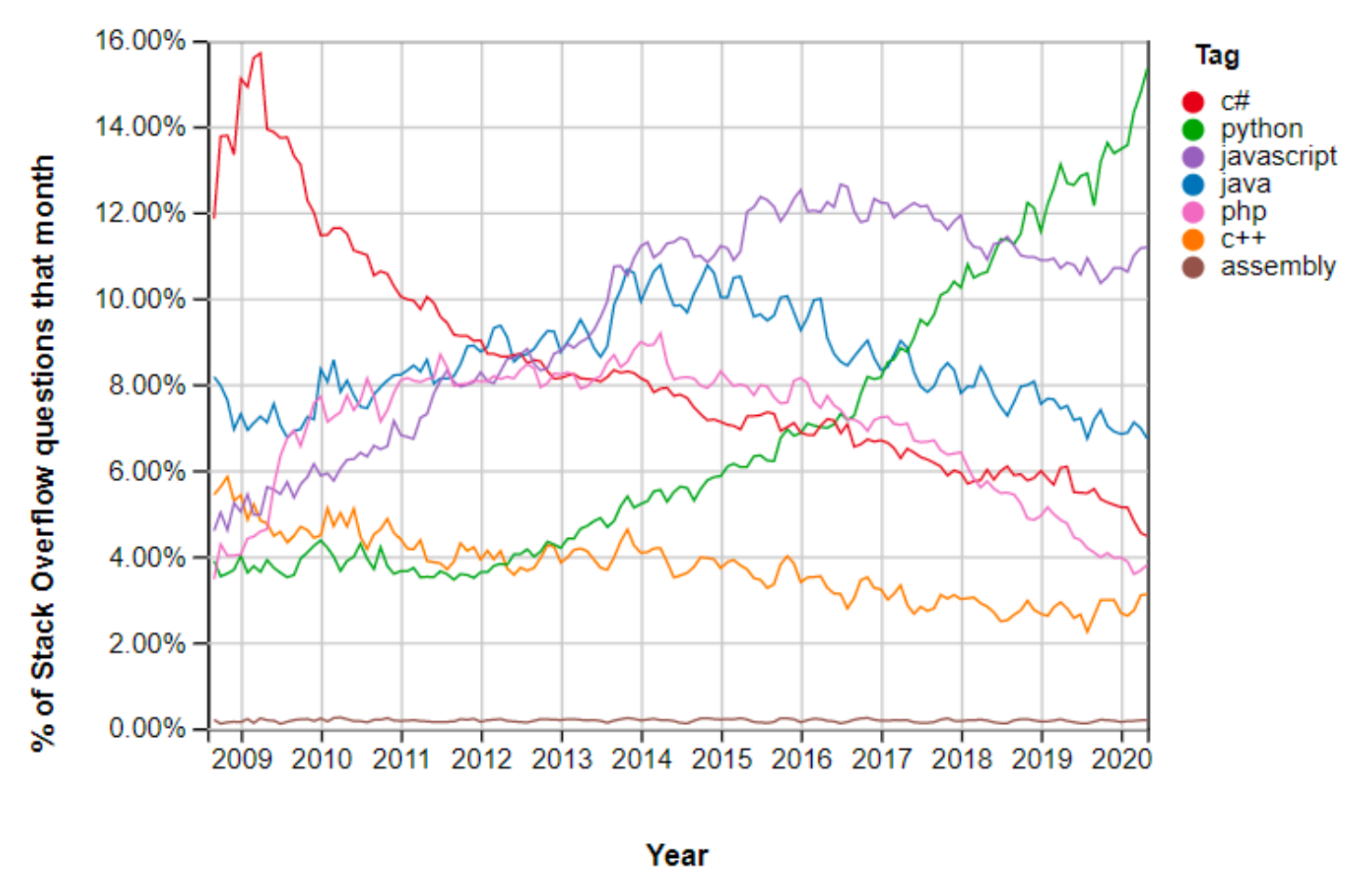
HOMEWORK

- Assignment 1
- Assignment 2
- Assignment 3

PYTHON OVERVIEW

Python is popular and widely supported

Python is quickly ascending to the forefront of the most popular programming languages in the world. The incredible growth of Python is shown very clearly by StackOverflow:

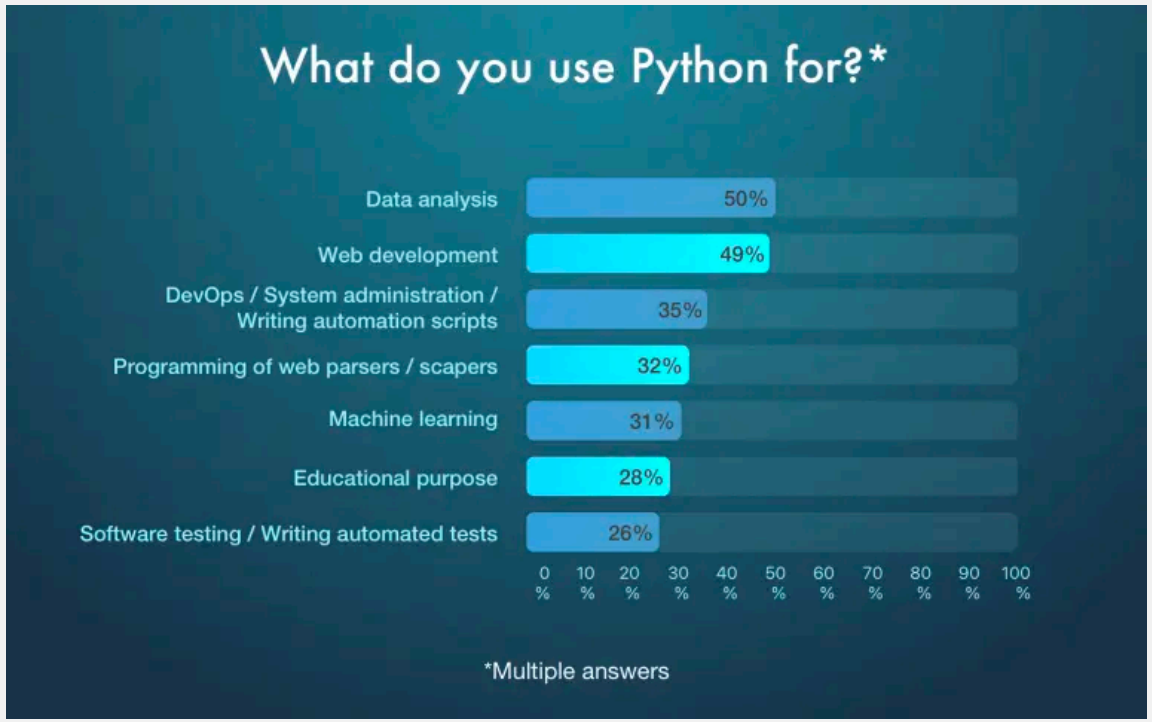


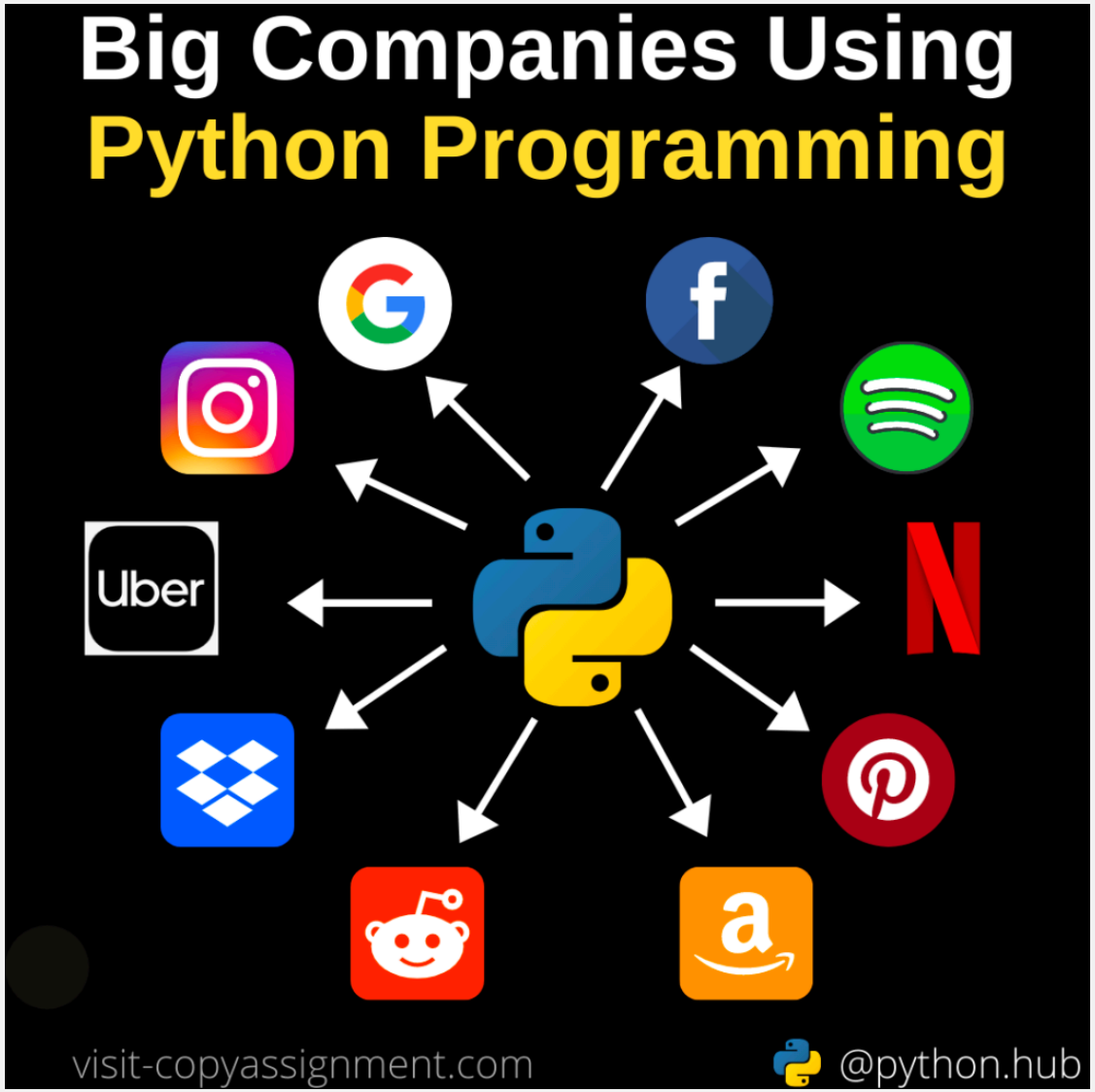
Python – fast growing

Python – easy to learn

Python – large community

Python applications





PYTHON: HISTORY & FEATURE

*Created by **Guido van Rossum in 1991** and further developed by the Python Software Foundation*

Easy Language, Readable

Designed to make developers life easy. Reading a Python code is like reading an English sentence.

Interpreted Language

Executes and displays the output of one line at a time.
Displays errors while you're running a line and displays the entire stack trace for the error

Dynamically-Typed Language

Don't need to declare data type while defining a variable

Popular and Large Community Support

Large Standard Library

Don't need to write it from scratch

High-level Language

Easy to understand and closer to the user.
Don't need to remember system architecture or manage the memory.


```c
#include <stdio.h>
int main(void)
{
    printf("Hello, world!");
}
```

C

```cpp
#include <iostream.h>
int main()
{
    std::cout << "Hello, world! ";
    return 0;
}
```

C++

```java
class HelloWorld {
    public static void main(String[]
args) {
        System.out.println("Hello,
World!");
    }
}
```

Java

```python
print "Hello, world!"
```

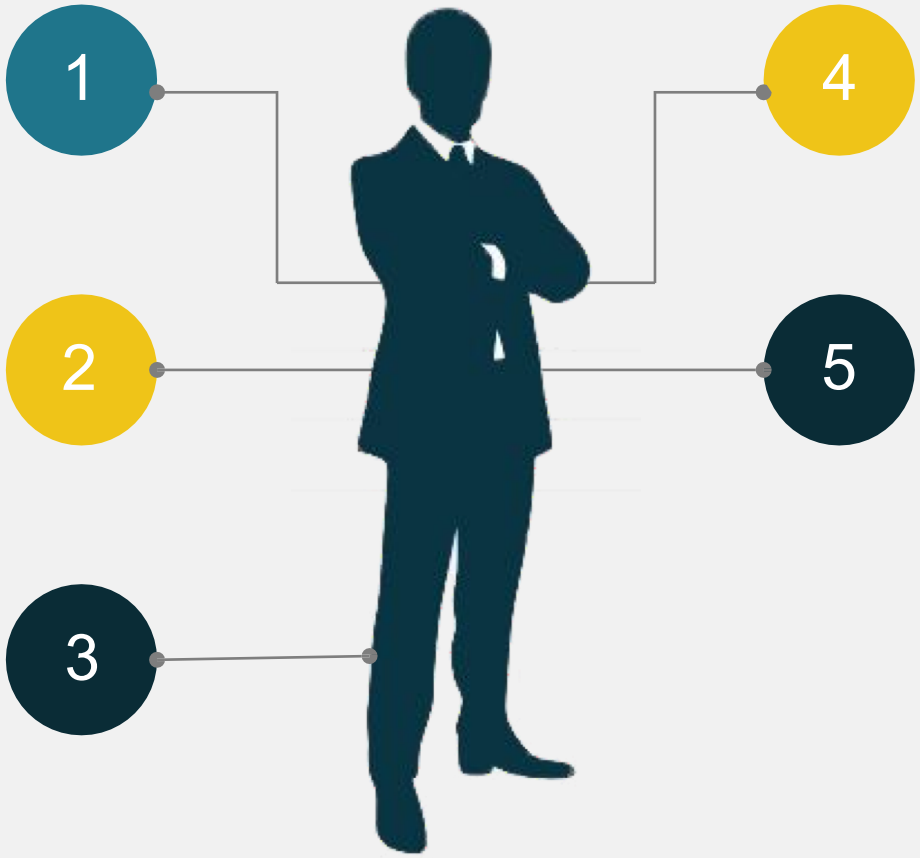
Python

THE ZEN OF PYTHON

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.



Complex is better than complicated.

Readability counts.

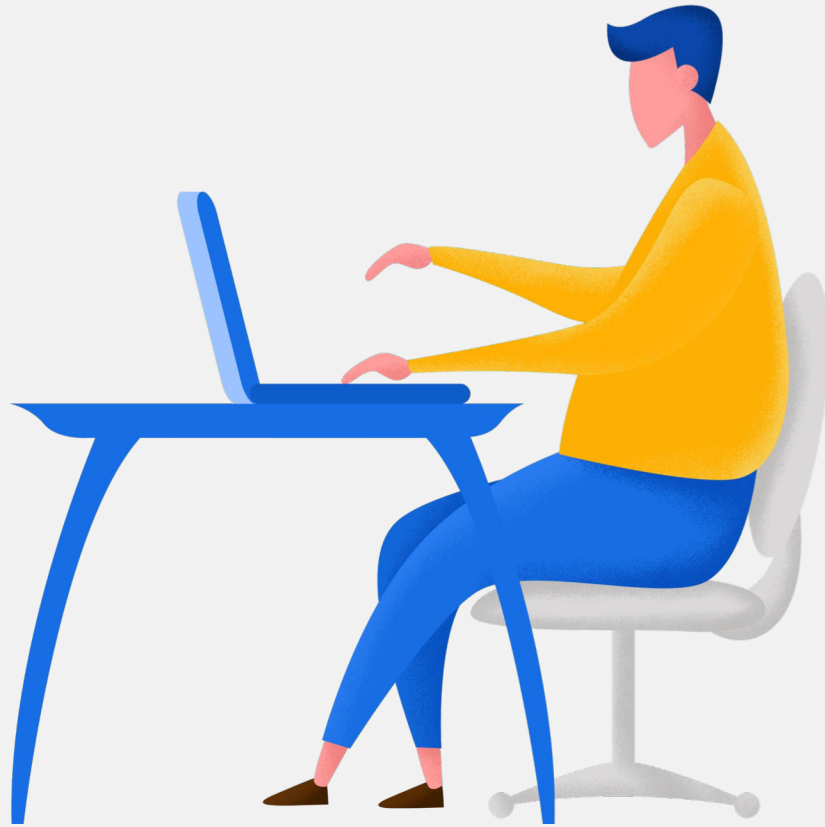
PYTHON'S IDEOLOGY

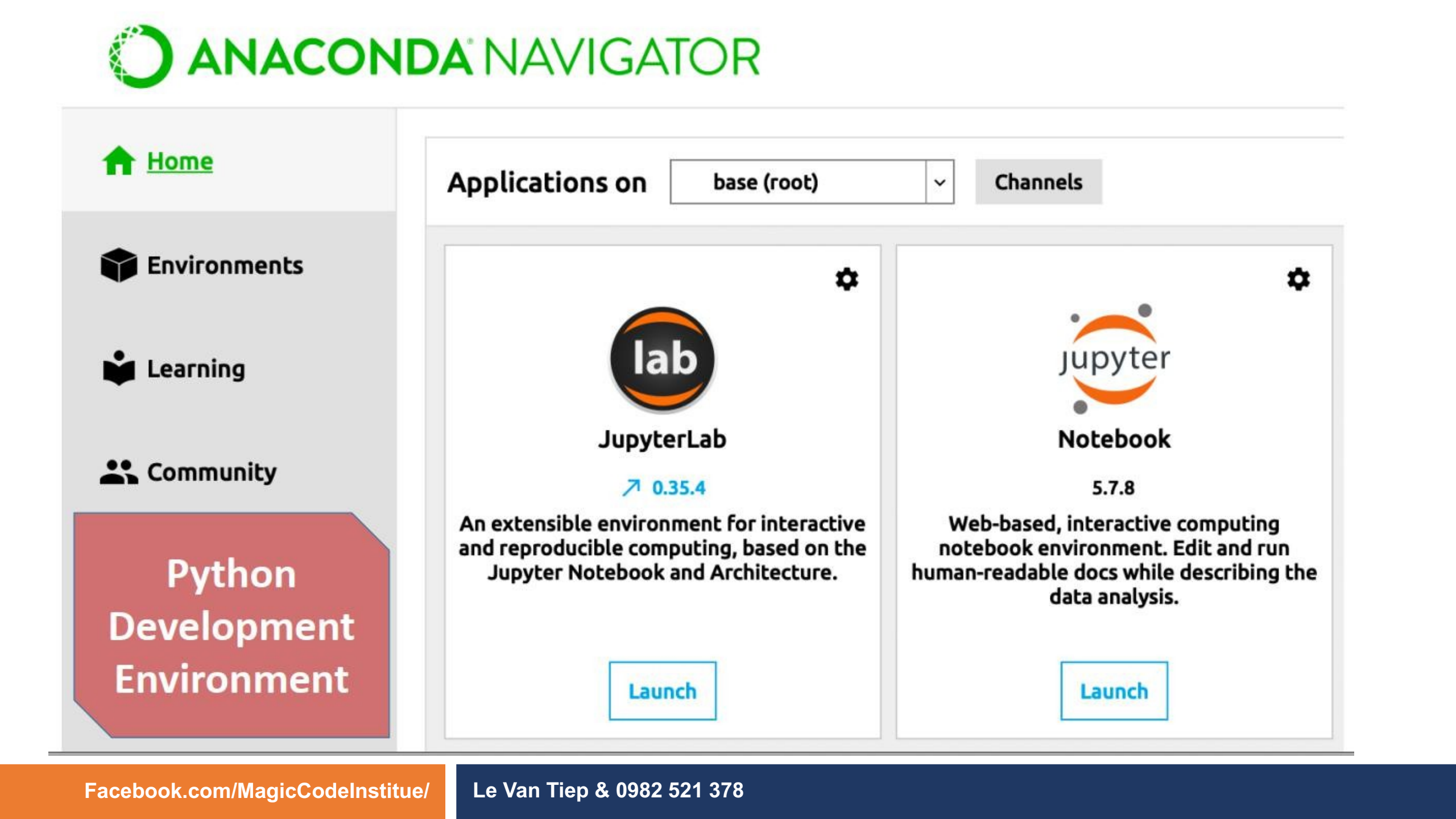
Perl:

- There is more than one way to do it

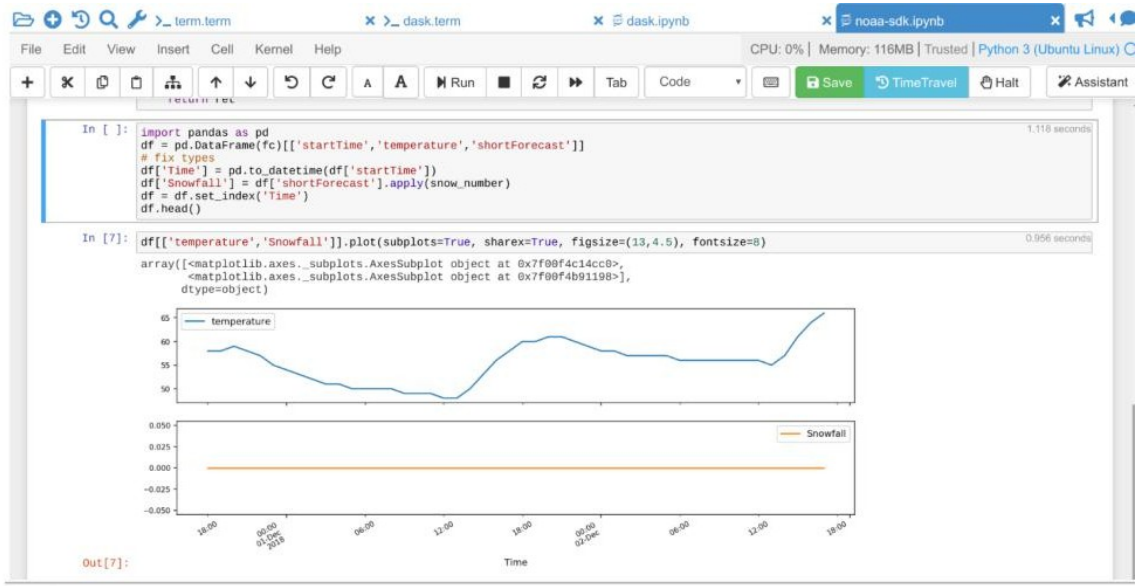
Python:

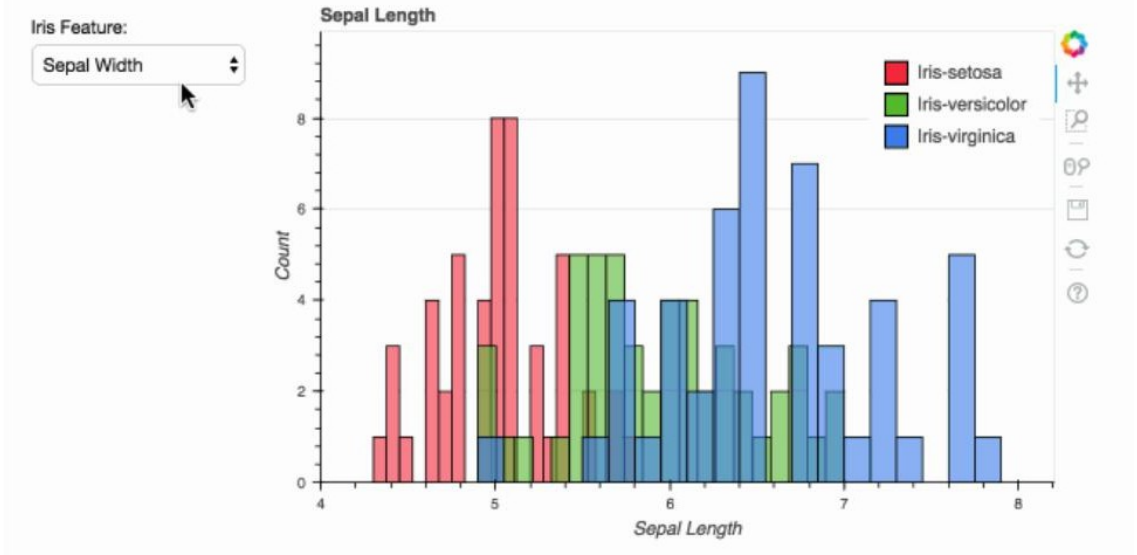
- There should be one – and preferably one – obvious way to do it




ANACONDA NAVIGATOR
Home
Environments
Learning
Community
Python Development Environment
Applications on
base (root)
Channels
lab
JupyterLab
0.35.4
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.
Launch
jupyter
Notebook
5.7.8
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.
Launch

- Jupyter = Julia + Python + R
- Multi-Language
- Packed with Anaconda
- Cross-Platform
- Web-Based
- Code & Visualize Data
- Among the most popular Python working environment for Data Scientists





2

IPYTHON BASICS

- The Python Interpreter
- Running the IPython Shell
- Running the Jupyter Notebook
- Tab Completion
- Introspection
- The %run command
- Interrupting Running Code
- Terminal Keyboard Shortcuts
- Magic Commands

2

IPYTHON BASICS

The Python Interpreter

Python is an *interpreted* language. The Python interpreter runs a program by executing one statement at a time. The standard interactive Python interpreter can be invoked on the command line with the python command:

```
$ python
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:17:12)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> print(a)
5
```

Hello_world.py

You can run it by executing the following command (the *hello_world.py* file must be in your current working terminal directory):

```
$ python hello_world.py
Hello world
```

2

IPYTHON BASICS

The Python Interpreter

When you use the `%run` command, IPython executes the code in the specified file in the same process, enabling you to explore the results interactively when it's done:

```
$ ipython
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:17:12)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: %run hello_world.py
Hello world

In [2]:
```

The default IPython prompt adopts the numbered `In [2]:` style compared with the standard `>>>` prompt.

2

IPYTHON BASICS

Running the IPython Shell

You can launch the IPython shell on the command line just like launching the regular Python interpreter except with the ipython command.

```
In [5]: import numpy as np

In [6]: data = {i : np.random.randn() for i in range(7)}

In [7]: data
Out[7]:
{0: -0.20470765948471295,
 1: 0.47894333805754824,
 2: -0.5194387150567381,
 3: -0.55573030434749,
 4: 1.9657805725027142,
 5: 1.3934058329729904,
 6: 0.09290787674371767}
```

→ Import numpy package

→ Create a variable named data

→ Print the value of data in the console

Be aware of current Keyboard Language mode: ENG – VIE → change it to ENG

IPYTHON BASICS

Running the Jupyter Notebook

One of the major components of the Jupyter project is the *notebook*, a type of interactive document for code, text (with or without markup), data visualizations, and other output. The Jupyter notebook interacts with *kernels*, which are implementations of the Jupyter interactive computing protocol in any number of programming lan guages. Python's Jupyter kernel uses the IPython system for its underlying behavior.

To start up Jupyter, run the command `jupyter notebook` in a terminal:

```
$ jupyter notebook
[I 15:20:52.739 NotebookApp] Serving notebooks from local directory:
/home/wesm/code/pydata-book
[I 15:20:52.739 NotebookApp] 0 active kernels
[I 15:20:52.739 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/
[I 15:20:52.740 NotebookApp] Use Control-C to stop this server and shut down
all kernels (twice to skip confirmation).
Created new window in existing browser session.
```

2

IPYTHON BASICS

Running the Jupyter Notebook

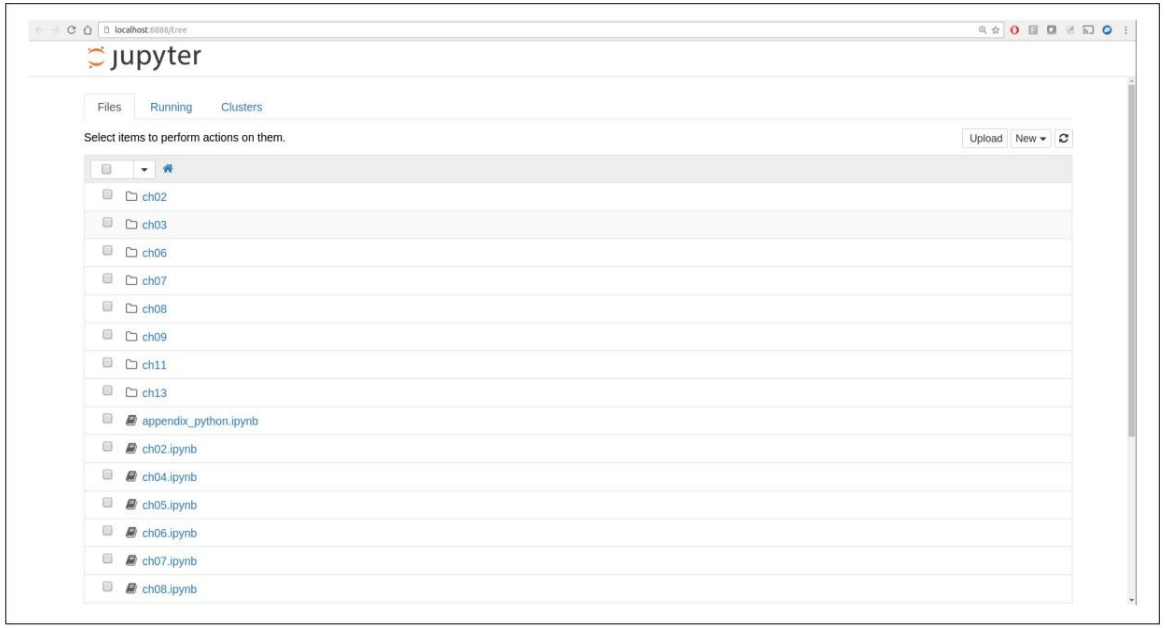


Figure 2-1. Jupyter notebook landing page

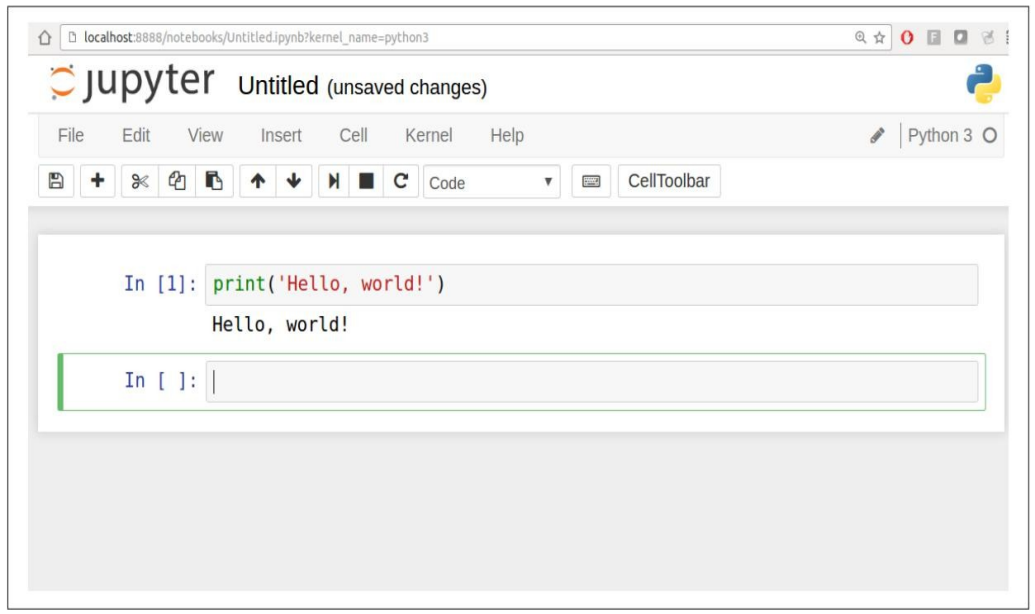


Figure 2-2. Jupyter new notebook view

2 IPYTHON BASICS

Tab Completion

On the surface, the IPython shell looks like a cosmetically different version of the standard terminal Python interpreter (invoked with python). One of the major improvements over the standard Python shell is *tab completion*, found in many IDEs or other interactive computing analysis environments. While entering expressions in the shell, pressing the **Tab** key will search the namespace for any variables (objects, functions, etc.) matching the characters you have typed so far:

```
In [1]: an_apple = 27

In [2]: an_example = 42

In [3]: an<Tab>
an_apple    and         an_example  any
```

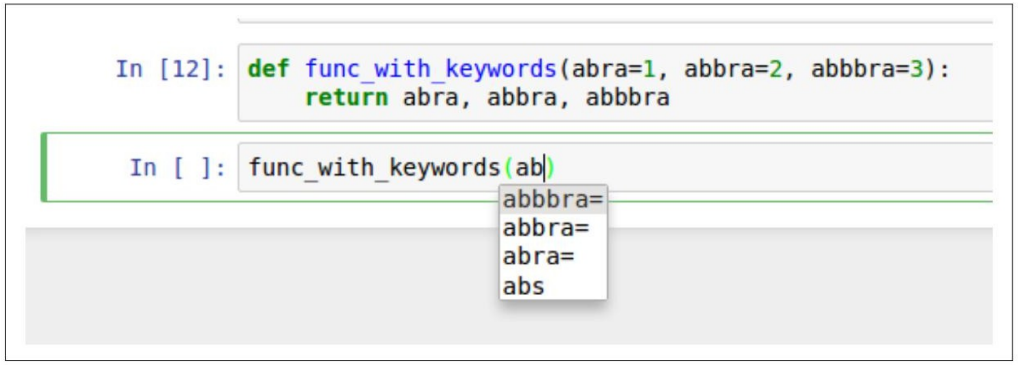


Figure 2-4. Autocomplete function keywords in Jupyter notebook

IPYTHON BASICS

Introspection

Using a question mark (?) before or after a variable will display some general information about the object:

```
In [8]: b = [1, 2, 3]

In [9]: b?
Type:        list
String Form:[1, 2, 3]
Length:      3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

IPYTHON BASICS

The %run Command

You can run any file as a Python program inside the environment of your IPython session using the `%run` command. Suppose you had the following simple script stored in *ipython_script_test.py*:

```
def f(x, y, z):
    return (x + y) / z

a = 5
b = 6
c = 7.5

result = f(a, b, c)
```

In the Jupyter notebook, you may also use the related `%load` magic function, which imports a script into a code cell:

```
>>> %load ipython_script_test.py
```

Interrupting running code → Pressing Ctrl-C while any code is running

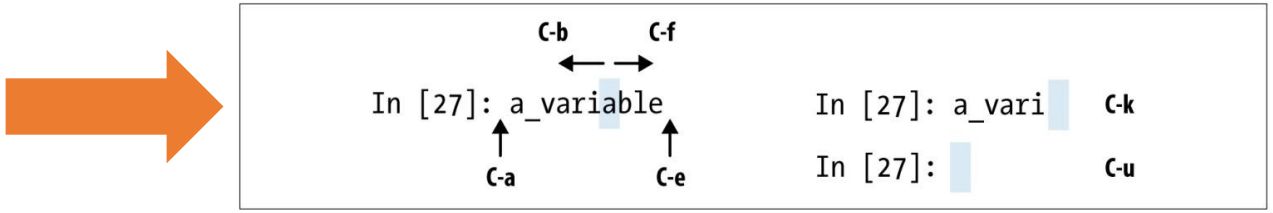
2

IPYTHON BASICS

Terminal Keyboard Shortcuts

Table 2-1. Standard IPython keyboard shortcuts

Keyboard shortcut	Description
Ctrl-P or up-arrow	Search backward in command history for commands starting with currently entered text
Ctrl-N or down-arrow	Search forward in command history for commands starting with currently entered text
Ctrl-R	Readline-style reverse history search (partial matching)
Ctrl-Shift-V	Paste text from clipboard
Ctrl-C	Interrupt currently executing code
Ctrl-A	Move cursor to beginning of line
Ctrl-E	Move cursor to end of line
Ctrl-K	Delete text from cursor until end of line
Ctrl-U	Discard all text on current line
Ctrl-F	Move cursor forward one character
Ctrl-B	Move cursor back one character
Ctrl-L	Clear screen



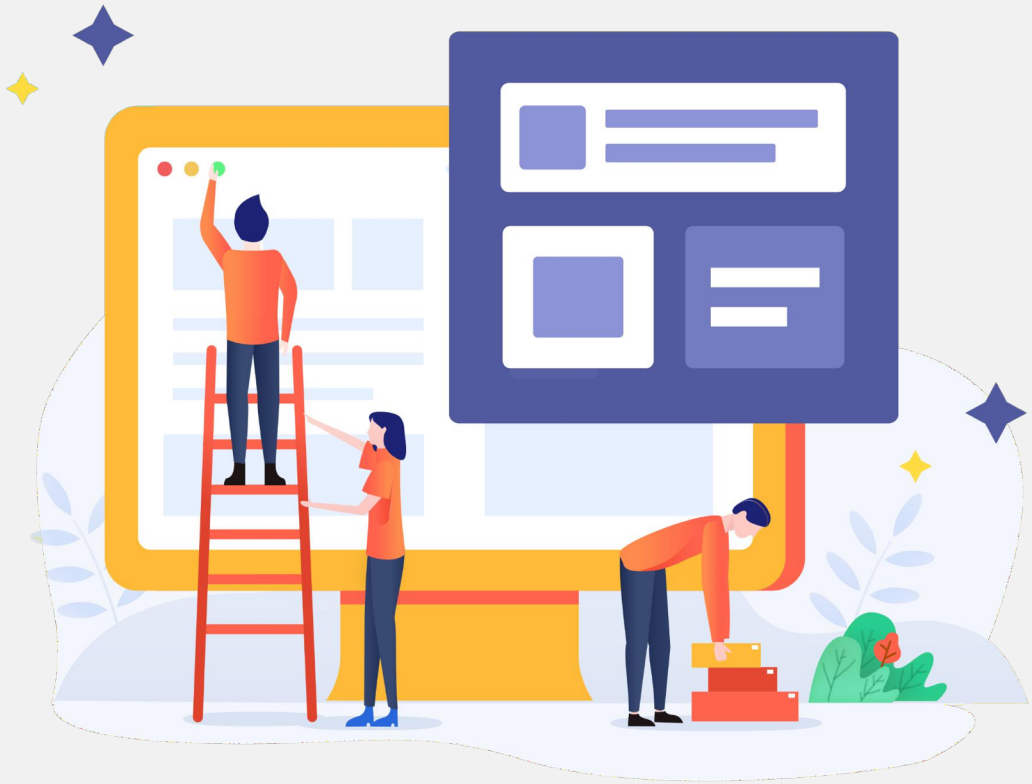
Magic Commands

Table 2-2. Some frequently used IPython magic commands

Command	Description
`%quickref`	Display the IPython Quick Reference Card
`%magic`	Display detailed documentation for all of the available magic commands
`%debug`	Enter the interactive debugger at the bottom of the last exception traceback
`%hist`	Print command input (and optionally output) history
`%pdb`	Automatically enter debugger after any exception
`%paste`	Execute preformatted Python code from clipboard
`%cpaste`	Open a special prompt for manually pasting Python code to be executed
`%reset`	Delete all variables/names defined in interactive namespace
`%page` *OBJECT*	Pretty-print the object and display it through a pager
`%run` *script.py*	Run a Python script inside IPython
`%prun` *statement*	Execute *statement* with `cProfile` and report the profiler output
`%time` *statement*	Report the execution time of a single statement
`%timeit` *statement*	Run a statement multiple times to compute an ensemble average execution time; useful for timing code with very short execution time
`%who, %who_ls, %whos`	Display variables defined in interactive namespace, with varying levels of information/ verbosity
`%xdel` *variable*	Delete a variable and attempt to clear any references to the object in the IPython internals

PYTHON LANGUAGE BASICS

- Language Semantics
- Scalar Types
- Control Flow



PYTHON LANGUAGE BASICS

Language Semantics

Indentation, not braces

Python uses whitespace (tabs or spaces) to structure code instead of using braces as in many other languages like R, C++, Java, and Perl. Consider a `for` loop from a sorting algorithm:

```
for x in array:
    if x < pivot:
        less.append(x)
    else:
        greater.append(x)
```

```
a = 5; b = 6; c = 7
```

3

PYTHON LANGUAGE BASICS

Everything is an object

An important characteristic of the Python language is the consistency of its object model. Every number, string, data structure, function, class, module, and so on exists in the Python interpreter in its own "box," which is referred to as a Python object. Each object has an associated type (e.g., string or function) and internal data. In practice this makes the language very flexible, as even functions can be treated like any other object.

Comments

Any text preceded by the hash mark (pound sign) # is ignored by the Python interpreter. This is often used to add comments to code. At times you may also want to exclude certain blocks of code without deleting them. An easy solution is to comment out the code

```
results = []
for line in file_handle:
    # keep the empty lines for now
    # if len(line) == 0:
    #   continue
    results.append(line.replace('foo', 'bar'))
```

```
print("Reached this line")  # Simple status report
```

PYTHON LANGUAGE BASICS

Function and object method calls

```
result = f(x, y, z)
g()
```

Variables and argument passing

```
In [8]: a = [1, 2, 3]

In [9]: b = a

In [10]: a.append(4)

In [11]: b
Out[11]: [1, 2, 3, 4]
```

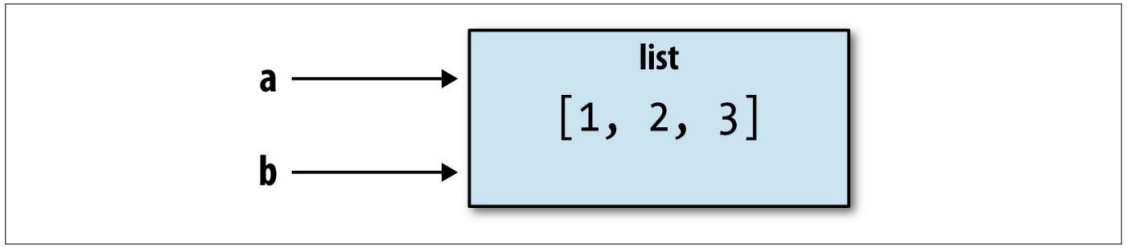


Figure 2-7. Two references for the same object

3

PYTHON LANGUAGE BASICS

Dynamic references, strong types

Attributes and methods

Duck typing

Imports

Binary operators and comparisons

Mutable and immutable objects

Table 2-3. Binary operators

Operation	Description
`a + b`	Add a and b
`a - b`	Subtract b from a
`a * b`	Multiply a by b
`a / b`	Divide a by b
`a // b`	Floor-divide a by b, dropping any fractional remainder
`a ** b`	Raise a to the b power
`a & b`	`True` if both a and b are `True`; for integers, take the bitwise AND
`a | b`	`True` if either a or b is `True`; for integers, take the bitwise OR
`a ^ b`	For booleans, `True` if a or b is `True`, but not both; for integers, take the bitwise `EXCLUSIVE-OR`

Operation	Description
`a == b`	`True` if a equals b
`a != b`	`True` if a is not equal to b
`a <= b, a < b`	`True` if a is less than (less than or equal) to b
`a > b, a >= b`	`True` if a is greater than (greater than or equal) to b
`a is b`	`True` if a and b reference the same Python object
`a is not b`	`True` if a and b reference different Python objects

PYTHON LANGUAGE BASICS

Scalar Types

Python support some data types by 7 groups as below:

- Text type: suitable for text format: **str**
- Numeric types: number format: **int**, **float**, complex
- Sequence types: for data chain: **list**, **tuple**, range
- Mapping type: for reference (dữ liệu ánh xạ): **dict**
- Set types: dữ liệu tập hợp: **set**, frozenset
- Boolean type: dữ liệu chân lý, đúng sai: **bool**
- Binary types: dữ liệu nhị phân: bytes, bytearray, memoryview

3 Python Data type examples

PYTHON OVERVIEW

```
# Số nguyên
a = 1
```

```
# Số thực
a = 1.5
```

```
# xâu
a = "abcdefgh"
```

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

```
# danh sách
a = [1, 2, '3', 4.5]
b = a
b.append('foo')
print(a)
```

```
[1, 2, '3', 4.5, 'foo']
```

```
print(a[0])
```

```
1
```

```
print(a[5])
```

```
---------------------------
IndexError
<ipython-input-29-8f63bedb8e
----> 1 print(a[5])

IndexError: list index out o
```

```
a[1] = 3
print(a)
```

```
[1, 3, '3', 4.5, 'foo']
```

```
# tuple
a = (1, 2, 3)
print(a)
```

```
(1, 2, 3)
```

```
print(a[0])
```

```
1
```

```
print(a[3])
```

```
---------------------------
IndexError
<ipython-input-35-5451d47c444b
----> 1 print(a[3])

IndexError: tuple index out of
```

```
a[0] = 0
```

```
---------------------------
TypeError
<ipython-input-36-7d02990d0abc
----> 1 a[0] = 0

TypeError: 'tuple' object does
```

```
# tập hợp
a = set()
a.add(1)
a.add(2)
a.add(2)
print(a)
```

```
{1, 2}
```

```
b = a
b.add(3)
print(a)
```

```
{1, 2, 3}
```

```
# từ điển
a = {}
a['name'] = 'foo'
a['age'] = 18
a[1] = 2
print(a)
```

```
{'name': 'foo', 'age': 18, 1: 2}
```

```
b = a
b['address'] = 'Hanoi'
print(a)
```

```
{'name': 'foo', 'age': 18, 1: 2, 'address': 'Hanoi'}
```

```
print(a['name'])
print(a[1])
```

```
foo
2
```

```
print(a['bar'])
```

```
---------------------------------------------------------
KeyError                                  Traceback (
<ipython-input-6-a500a0a32466> in <module>
----> 1 print(a['bar'])

KeyError: 'bar'
```

Casting type

- Casting type allows change data type to expected data type as below

```
a = "1"
```

```
type(a)
```

```
str
```

```
a = int(a)
```

```
type(a)
```

```
int
```

```
a = [1, 1, 2, 3, 4, 5, 6]
```

```
a = tuple(a)
```

```
type(a)
```

```
tuple
```

```
a = str(1)
```

```
type(a)
```

```
str
```

```
a = [1, 1, 2, 3, 4, 5, 6]
a = set(a)
print(a)
```

```
{1, 2, 3, 4, 5, 6}
```

```
a = [1, 1, 2, 3, 4, 5, 6]
a = dict(a)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-b383c7ee338b> in <module>
      1 a = [1, 1, 2, 3, 4, 5, 6]
----> 2 a = dict(a)

TypeError: cannot convert dictionary update sequence element #0 to a sequence
```

PYTHON LANGUAGE BASICS

Control Flow

Python has several built-in keywords for conditional logic, loops, and other standard *control flow* concepts found in other programming languages.

3

Condition (if...elif...else)

- Condition programming allows to run next depending on previous condition

```
a = 1
b = 2
```

```
if a < b:
    print("a nhỏ hơn b")
```

```
a nhỏ hơn b
```

```
a = 2
b = 1
```

```
if a < b:
    print("a nhỏ hơn b")
```

```
a = 2
b = 1
```

```
if a < b:
    print("a nhỏ hơn b")
elif a == 2:
    print("a bằng 2")
else:
    print("a không nhỏ hơn b")
```

```
a bằng 2
```

```
a = 2
b = 1
```

```
if a < b:
    print("a nhỏ hơn b")
else:
    print("a không nhỏ hơn b")
```

```
a không nhỏ hơn b
```

3 Loop (for, while)

- Loop allows to run a line of code multiple times
- 02 keywords frequently used in a loop

➢ `continue: go next another loop time`

➢ `break: go out of loop`

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

```
0
1
2
3
4
```

```
a = 0
while a < 5:
    a += 1
    print(a)
```

```
1
2
3
4
5
```

```
a = [1, 3, 5, 7, 9]
for e in a:
    print(e)
```

```
1
3
5
7
9
```

```
for i in range(10):
    if i % 2:
        continue
    print(i)
```

```
0
2
4
6
8
```

```
for i in range(10):
    if i == 4:
        break
    print(i)
```

```
0
1
2
3
```

4 BUILT-IN DATA STRUCTURES

- Strings
- Lists
- Tuple
- Dictionary

4

BUILT-IN DATA STRUCTURES - Strings

All of the following lines are strings being assigned to a variable:

```
a = "Hey"
b = "Hey there!"
c = "742 Evergreen Terrace"
d = "1234"
e = "'How long is a piece of string?' he asked"
f = "'!$*#@ you!' she replied"
```

So if we print those out it would look like this:

RESULT

```
Hey
Hey there!
742 Evergreen Terrace
1234
'How long is a piece of string?' he asked
'!$*#@ you!' she replied
```

4 BUILT-IN DATA STRUCTURES - Strings

Return a String's Length

You can use the `len()` function to return a string's length. To do this, simply pass the string in as an argument to the function.

```
a = "Hey"
print(len(a))
```

RESULT

```
3
```

4 BUILT-IN DATA STRUCTURES - Strings

There are a couple of different ways to make a string span multiple lines.

Escape Sequence

Just as we used the backslash to escape the double quotes, we can use an escape sequence to force a string to span multiple lines.

```
msg = "ATTENTION!\rFor those about to rock, we salute you!"
print(msg)
```

RESULT

```
ATTENTION!
For those about to rock, we salute you!
```

BUILT-IN DATA STRUCTURES - Strings

Triple Quotes

You can also use triple quotes to enclose strings that span multiple lines. This is especially handy if you have many lines, as it saves you from having to put a backslash on the end of every line.

So you can use triple double quotes:

```
msg = """This string
spans
multiple lines"""
print(msg)
```

Or triple single quotes

```
msg = '''This string
spans
multiple lines'''
print(msg)
```

4

BUILT-IN DATA STRUCTURES - Lists

To create a list in Python, simply add any number of comma separated values between square brackets. Like this:

```
planets = ["Earth", "Mars", "Saturn", "Jupiter"]
```

In Python, a list can contain items of varying data types. For example, here's a list that contains a string, an integer, and another list.

```
mixedList = ["Hey", 123, ["Dog", "Cat", "Bird"]]
```

When we print both of those lists we get this:

RESULT

```
['Earth', 'Mars', 'Saturn', 'Jupiter']
['Hey', 123, ['Dog', 'Cat', 'Bird']]
```

BUILT-IN DATA STRUCTURES - Lists

Return the Number of items in a List

You can use the `len()` function to return the number of items in a list. Adding the `len()` function to the `print()` function will print that number out.

```
planets = ["Earth", "Mars", "Saturn", "Jupiter"]
print(len(planets))
```

Accessing a List Item

You can pick out a single list item by referring to its index. Remember, Python uses zero-based indexing, so start the count at `0` when doing this.

So to find out what the second item in the `planet` list is, do this:

```
planets = ["Earth", "Mars", "Saturn", "Jupiter"]
print(planets[1])
```

BUILT-IN DATA STRUCTURES - Lists

Update a List Item

```
# Set the initial list
planets = ["Earth", "Mars", "Saturn", "Jupiter"]
# Print the initial list
print(planets)
# Update the list item
planets[1] = "Mercury"
# Print the updated list
print(planets)
```

RESULT

```
['Earth', 'Mars', 'Saturn', 'Jupiter']
['Earth', 'Mercury', 'Saturn', 'Jupiter']
```

BUILT-IN DATA STRUCTURES - Lists

Append a List Item

You can add items to a list by using the `append()` function. Here's an example:

```
planets.append("Mercury")
```

Here's a demonstration:

```
# Set the initial list
planets = ["Earth", "Mars", "Saturn", "Jupiter"]
# Print the initial list
print(planets)
# Add the new list item
planets.append("Mercury")
# Print the updated list
print(planets)
```

RESULT

```
['Earth', 'Mars', 'Saturn', 'Jupiter']
['Earth', 'Mars', 'Saturn', 'Jupiter', 'Mercury']
```

BUILT-IN DATA STRUCTURES - Lists

Delete a List Item

```
# Set the initial list
planets = ["Earth", "Mars", "Saturn", "Jupiter"]
# Print the initial list
print(planets)
# Delete the third list item
del planets[2]
# Print the updated list
print(planets)
```

RESULT

```
['Earth', 'Mars', 'Saturn', 'Jupiter']
['Earth', 'Mars', 'Jupiter']
```

4 BUILT-IN DATA STRUCTURES - Tuple

Create a Tuple

```
# Set the tuples
weekdays = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
weekenddays = "Saturday", "Sunday"
# Print the tuples
print(weekdays)
print(weekenddays)
```

RESULT

```
('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
('Saturday', 'Sunday')
```

4 BUILT-IN DATA STRUCTURES - Tuple

Access the Values in a Tuple

You can access the values in a tuple the same way you can with lists — by appending the variable name with an index value enclosed in square brackets. Like this:

```
# Set the tuple
weekdays = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
# Print the second element of the tuple
print(weekdays[1])
```

RESULT

```
Tuesday
```

BUILT-IN DATA STRUCTURES - Tuple

Concatenate Tuples

You can also use a tuple as the basis for another tuple. For example, you can concatenate a tuple with another one to create a new a tuple that contains values from both tuples.

```
# Set two tuples
weekdays = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
weekenddays = ("Saturday", "Sunday")
# Set a third tuple to the value of the previous two
alldays = weekdays + weekenddays
# Print them all
print(weekdays)
print(weekenddays)
print(alldays)
```

RESULT

```
('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
('Saturday', 'Sunday')
('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday')
```

4 BUILT-IN DATA STRUCTURES - Dictionary

You can create a dictionary by enclosing comma separated `key: value` pairs within curly braces `{}`. Like this:

```
d = {"Key1": "Value1", "Key2": "Value2"}
```

Here's an example of creating a dictionary, then printing it out, along with its type:

```
# Create the dictionary
planet_size = {"Earth": 40075, "Saturn": 378675, "Jupiter": 439264}
# Print the dictionary
print(planet_size)
# Print the type
print(type(planet_size))
```

RESULT

```
{'Earth': 40075, 'Saturn': 378675, 'Jupiter': 439264}
<class 'dict'>
```

4 BUILT-IN DATA STRUCTURES - Dictionary

Access the Values in a Dictionary

```
# Create the dictionary
planet_size = {"Earth": 40075, "Saturn": 378675, "Jupiter": 439264}
# Square brackets
print(planet_size["Earth"])
# get() function
print(planet_size.get("Saturn"))
```

RESULT

```
40075
378675
```

4 BUILT-IN DATA STRUCTURES - Dictionary

Update an Existing Item

Here's an example that demonstrates updating an existing item:

```
# Create the dictionary and print
user = {"Name": "Christine", "Age": 23}
print(user)
# Update the user's age and print
user["Age"] = 24
print(user)
```

RESULT

```
{'Name': 'Christine', 'Age': 23}
{'Name': 'Christine', 'Age': 24}
```

4 BUILT-IN DATA STRUCTURES - Dictionary

Add a New Item

Here's an example that demonstrates adding a new item (i.e. updating a key that doesn't exist):

```
# Create the dictionary and print
user = {"Name": "Christine", "Age": 23}
print(user)
# Add a new item and print
user["Height"] = 154
print(user)
```

RESULT

```
{'Name': 'Christine', 'Age': 23}
{'Name': 'Christine', 'Age': 23, 'Height': 154}
```

4 BUILT-IN DATA STRUCTURES - Dictionary

Delete a Dictionary Item

You can use the `del` keyword to delete a given dictionary item. To do this, simply refer to the item's key. Like this:

```
# Create dictionary and print
planet_size = {"Earth": 40075, "Saturn": 378675, "Jupiter": 439264}
print(planet_size)
# Delete an item then print
del planet_size["Jupiter"]
print(planet_size)
```

RESULT

```
{'Earth': 40075, 'Saturn': 378675, 'Jupiter': 439264}
{'Earth': 40075, 'Saturn': 378675}
```

FUNCTIONS & FILES

- Built-in functions
- User-defined functions

FUNCTIONS & FILES - Functions

Built-In Python Functions

abs()	frozenset()	open()
all()	getattr()	ord()
any()	globals()	pow()
ascii()	hasattr()	print()
bin()	hash()	property()
bool()	help()	range()
bytearray()	hex()	repr()
bytes()	id()	reversed()

FUNCTIONS & FILES - Functions

Built-In Python Functions

`callable()`	`input()`	`round()`
`chr()`	`int()`	`set()`
`classmethod()`	`isinstance()`	`setattr()`
`compile()`	`issubclass()`	`slice()`
`complex()`	`iter()`	`sorted()`
`delattr()`	`len()`	`staticmethod()`
`dict()`	`list()`	`str()`
`dir()`	`locals()`	`sum()`

FUNCTIONS & FILES - Functions

Built-In Python Functions

`divmod()`	`map()`	`super()`
`enumerate()`	`max()`	`tuple()`
`eval()`	`memoryview()`	`type()`
`exec()`	`min()`	`vars()`
`filter()`	`next()`	`zip()`
`float()`	`object()`	`__import__()`
`format()`	`oct()`	

FUNCTIONS & FILES - Functions

User-Defined Functions

```
def multiplyMe(x):
    y = x * x
    return y
```

```
print(multiplyMe(10))
```

RESULT

```
100
```

```
a = multiplyMe(10)
b = multiplyMe(15)
c = a + b
print(c)
```

RESULT

```
325
```

FUNCTIONS & FILES - Functions

User-Defined Functions

Return Multiple Values

```
# Function to perform the four basic arithmetic operations
# against two numbers that are passed in as arguments.
def basicArithmetic(x, y):

    # Do the calulations and put each result into a variable
    sum = x + y
    product = x * y
    quotient = x / y
    difference = x - y

    # Return each variable
    return sum, product, quotient, difference


# Call the function and print the result
print(basicArithmetic(3, 30))
```

RESULT

```
(33, 90, 0.1, -27)
```

FUNCTIONS & FILES - Functions

Python Modules

A module is simply a file containing Python definitions and statements, and it has a file extension of `.py`. You can import modules into other modules in order to make use of the code that the imported modules contain.

Python also comes with a library of standard modules. Some modules are built into the Python interpreter, which means that you can use them immediately.

Here's an example, where we import the `random` module in order to generate a random number:

```
import random
print(random.randint(1,100))
```

RESULT

```
72
```

FUNCTIONS & FILES - Functions

Assign a Local Name

If you don't like prefixing functions with the module name, you can assign it a local name. This allows you to call that function without the prefix.

Here's how to do that:

```
import random
randint = random.randint
print(randint(1,100))
```

RESULT

```
82
```

FUNCTIONS & FILES - Functions

Variation of the `import` Statement

There's a variation of the `import` statement that allows you to name the functions to import from the module. When you do this, you can call the functions without having to qualify them with the module name.

It goes like this:

```
from random import randint
print(randint(1,100))
```

RESULT

```
93
```

FUNCTIONS & FILES - Files

Python Read Write Files

Create a File

Here's an example of creating a new file:

```
# Create the file in 'write' mode
f = open("hello.txt", "w")

# Write some text to the file
f.write("Hello World!")

# Close the file
f.close()
```

FUNCTIONS & FILES - Files

Read a File

Now that we've created a file and added some content to it, we can read it. Here's how:

```
# Open the file in 'read' mode
f = open("hello.txt", "r")

# Put the contents of the file into a variable
f_contents = f.read()

# Close the file
f.close()

# Print the file's contents
print(f_contents)
```

RESULT

```
Hello World!
```

II

CASE STUDY

1

CASE STUDY - 01

CASE STUDY

- Assuming (name = “John Smith”), what does name[1] return?
- What about name[-2]?
- What about name[1:-1]?
- How to get the length of name?

CASE STUDY - 02

Input	Output		
aaabbbcccddd111	abcd1	Case 1	Loại bỏ các giá trị trùng
aaabbbcccddd111	aaa bbb ccc ddd 111	Case 2	Tách các giá trị trùng ra

III

HOMEWORK

HOMEWORK

1 Assignment 01

- Write a function that returns the maximum of two numbers.

HOMEWORK

1 Assignment 02

- Write a function called **show_stars(rows).** If **rows** is 5, it should print the following:
  - *
  - **
  - ***
  - ****
  - *****

HOMEWORK

1

Assignment 03

- Write a function called **fizz_buzz** that takes a number.
  - If the number is divisible by 3, it should return “Fizz”.
  - If it is divisible by 5, it should return “Buzz”.
  - If it is divisible by both 3 and 5, it should return “FizzBuzz”.
  - Otherwise, it should return the same number

THANKS FOR LISTENING!!!