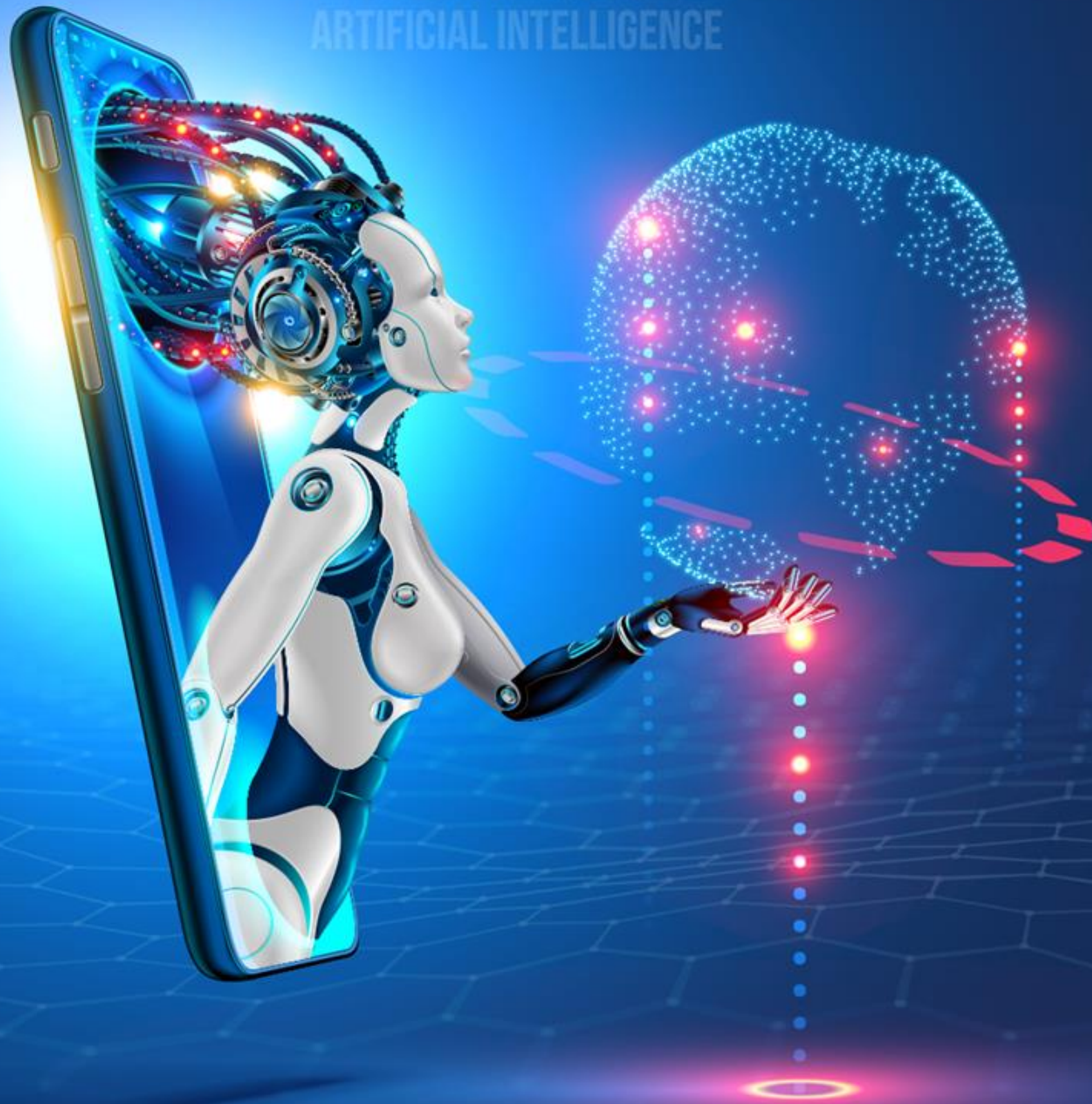


# DATA AND ARTIFICIAL INTELLIGENCE



Data Science with Python

# DATA AND ARTIFICIAL INTELLIGENCE



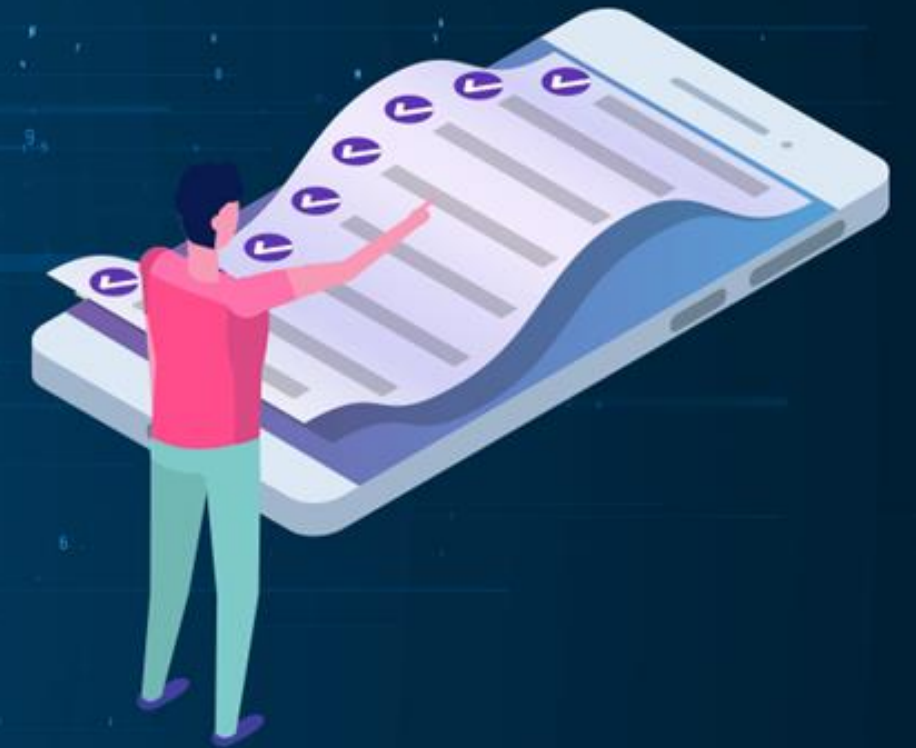
## Python: Environment Setup and Essentials



# Learning Objectives

By the end of this lesson, you will be able to:

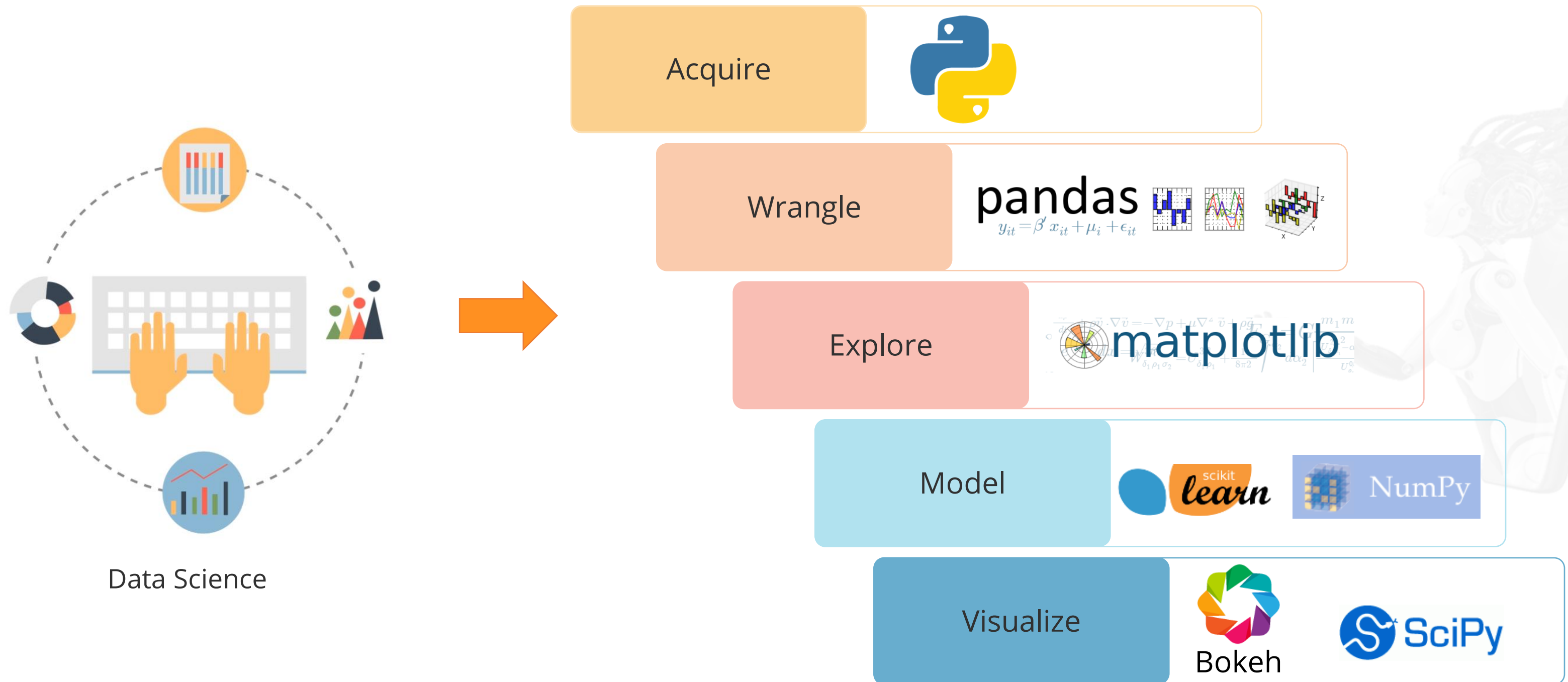
- 🕒 Explain Anaconda and Jupyter notebook installation
- 🕒 List the important data types supported by Python
- 🕒 Discuss data structures, such as lists, tuples, sets, and dicts
- 🕒 Explain slicing and accessing the four data structures
- 🕒 Discuss basic operators and functions
- 🕒 Outline the important control flow statements



## Anaconda: The World's Most Popular Data Science Platform

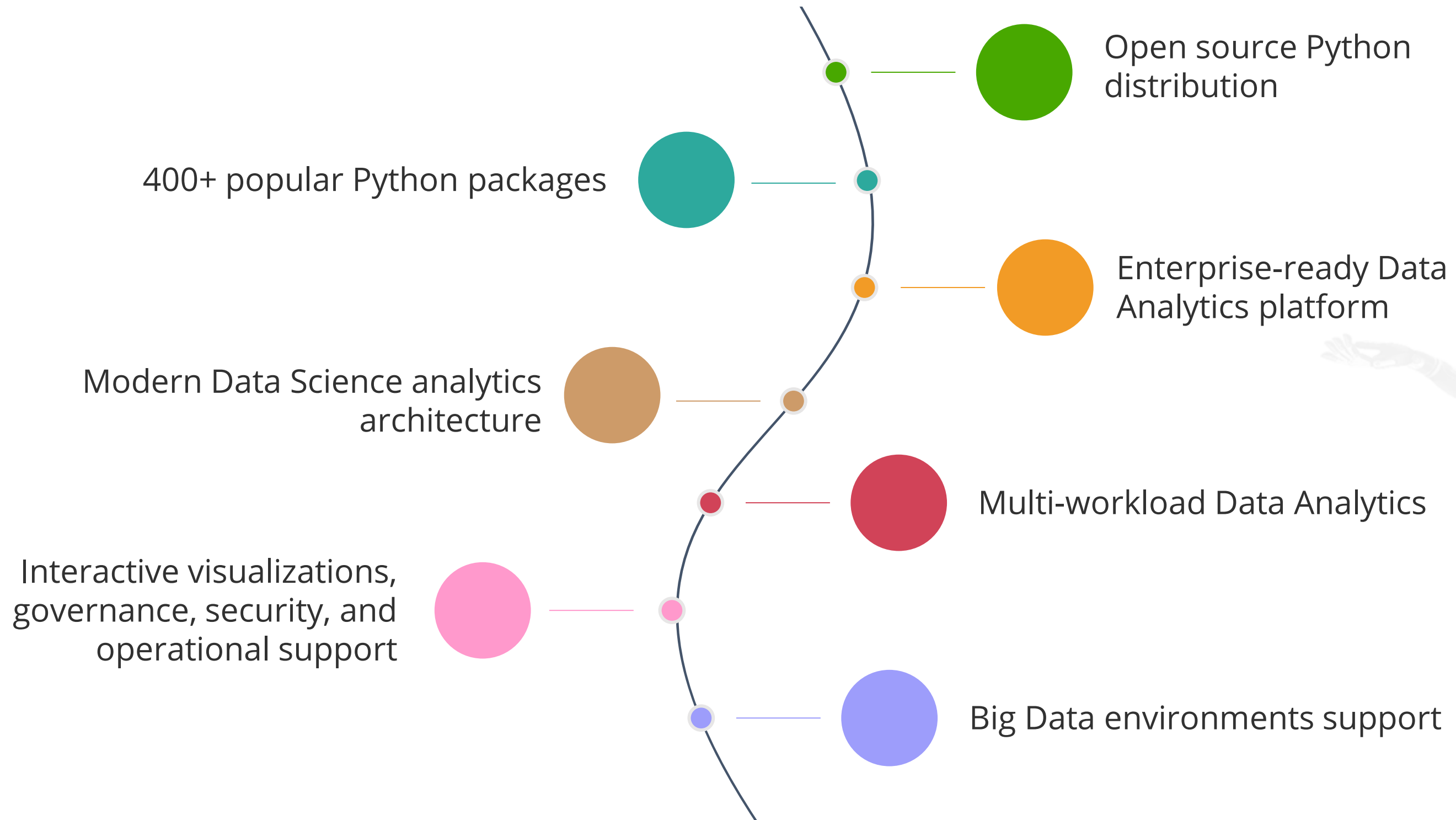
# Quick Recap: Python for Data Science

You know the importance of Python and its libraries in various aspects of Data Science.



# Why Anaconda

To use Python, we recommend that you download Anaconda. Following are some of the reasons why Anaconda is one of the best Data Science platforms:



# Installation of Anaconda Python Distribution

Currently, there are two versions of Python. You can download and use 3.7 version, as the course is designed based on the latest version.

## Python 3.7 version

64-Bit Graphical Installer (637 MB)  
64-Bit Command Line Installer (542 MB)

## Python 2.7 version

64-Bit Graphical Installer (624 MB)  
64-Bit Command Line Installer (530 MB )

# Installation of Anaconda Python Distribution

You can install and run the Anaconda Python distribution on different platforms.

## PYTHON 3.7

Windows

Mac OS

Linux

### Python 3.7 version

64-Bit Graphical Installer (637 MB)

64-Bit Command Line Installer (542 MB)



#### Website URL:

<https://www.continuum.io/downloads>

#### Graphical Installer

- Download the graphical installer.
- Double-click the .exe file to install Anaconda and follow the instructions on the screen.



# Installation of Anaconda Python Distribution

## PYTHON 3.7

Windows

Mac OS

Linux



Website URL:

<https://www.continuum.io/downloads>

### Graphical Installer

- Download the graphical installer.
- Double-click the downloaded .pkg file and follow the instructions.

### Command Line Installer

- Download the command line installer.
- In your terminal window, type the command listed below and follow the given instructions:

**Python 3.7:**

```
bash Anaconda2-4.0.0-MacOSX-x86_64.sh
```

# Installation of Anaconda Python Distribution

PYTHON 3.7

Windows

Mac OS

Linux

LINUX 64-BIT

392M

Linux 32-bit

332M



Website URL:

<https://www.continuum.io/downloads>

Command Line Installer

- Download the installer.
- In your terminal window, type the command line shown below and follow the instructions:

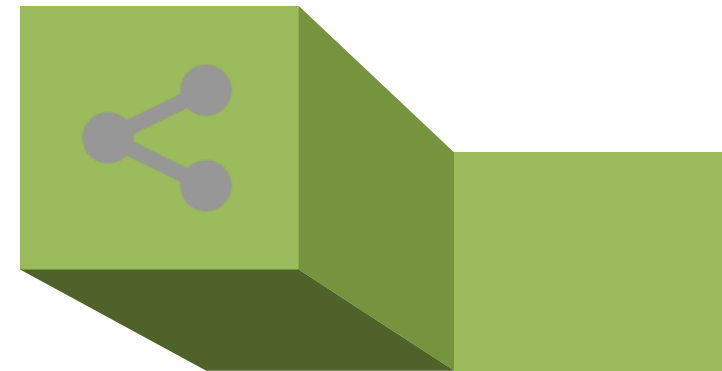
Python 3.7:

```
bash Anaconda2-4.0.0-Linux-x86_64.sh
```

# Jupyter Notebook

Jupyter is an open source and interactive web-based Python interface for Data Science and scientific computing.  
Some of the advantages are:

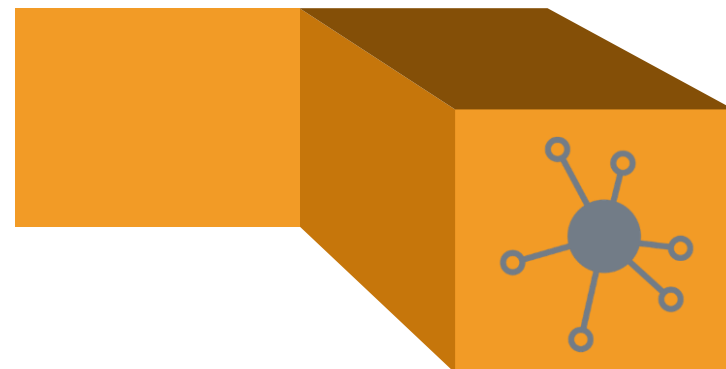
Python language  
support



Content sharing and  
contribution



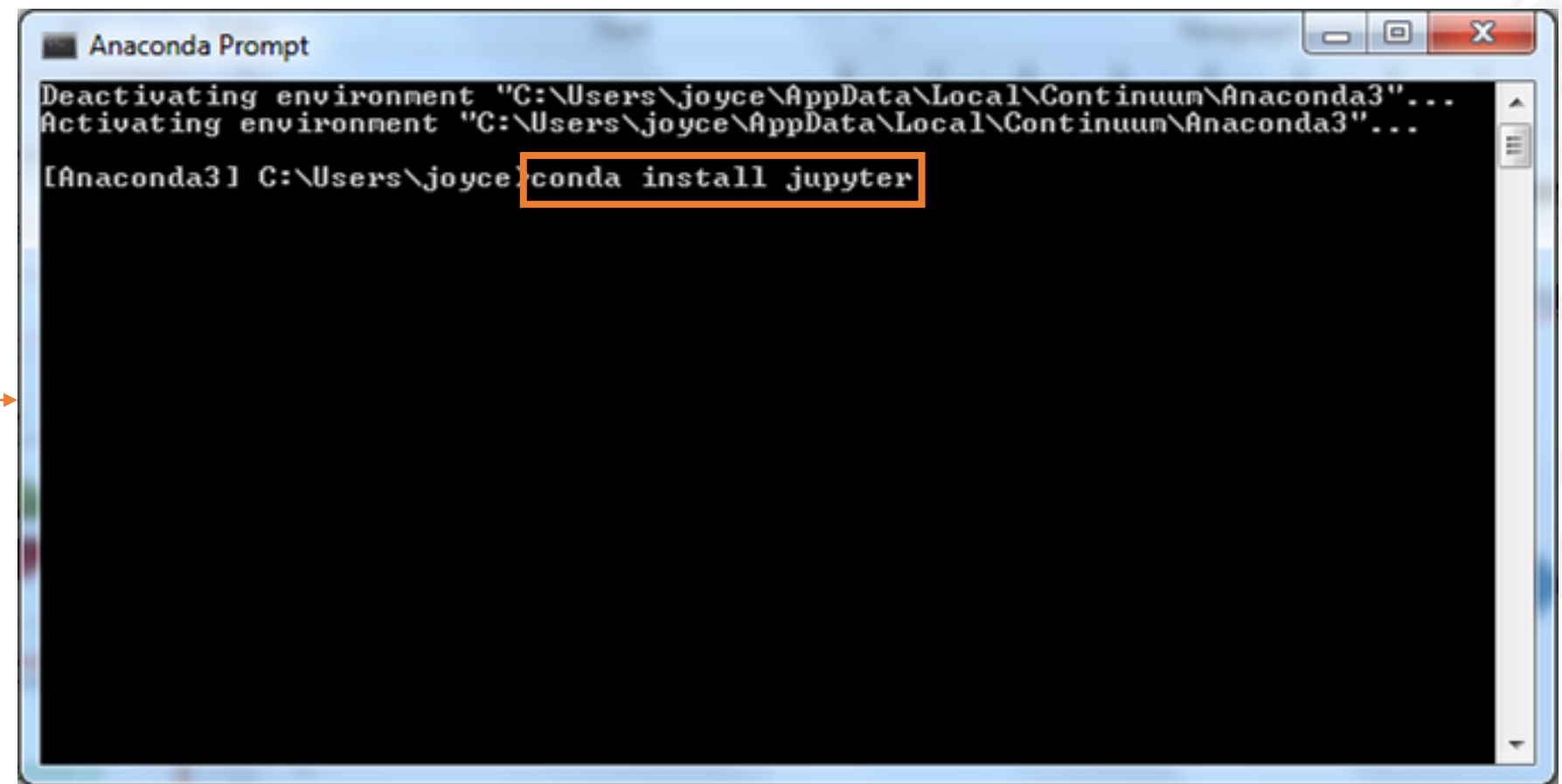
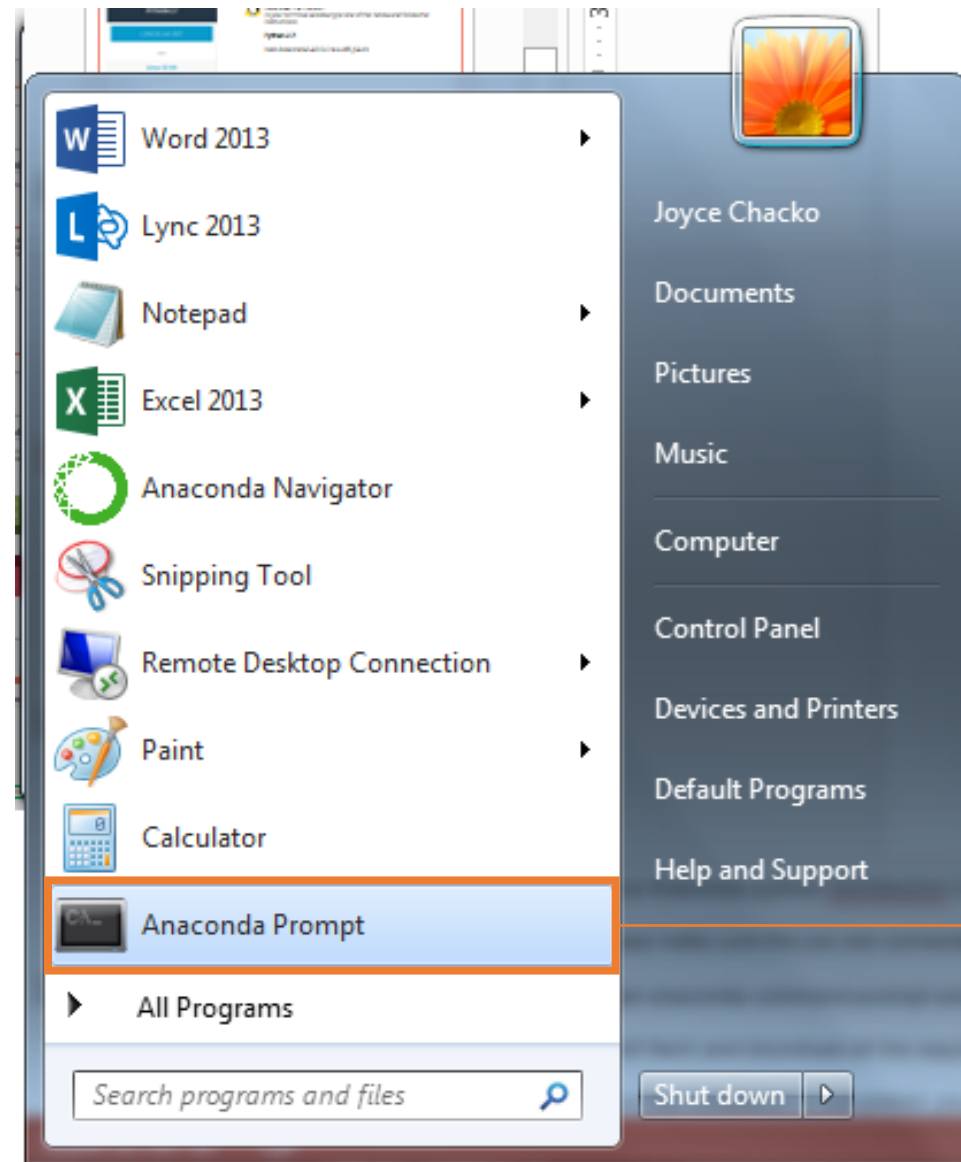
Big Data platform  
integration



Built-in interactive  
widgets

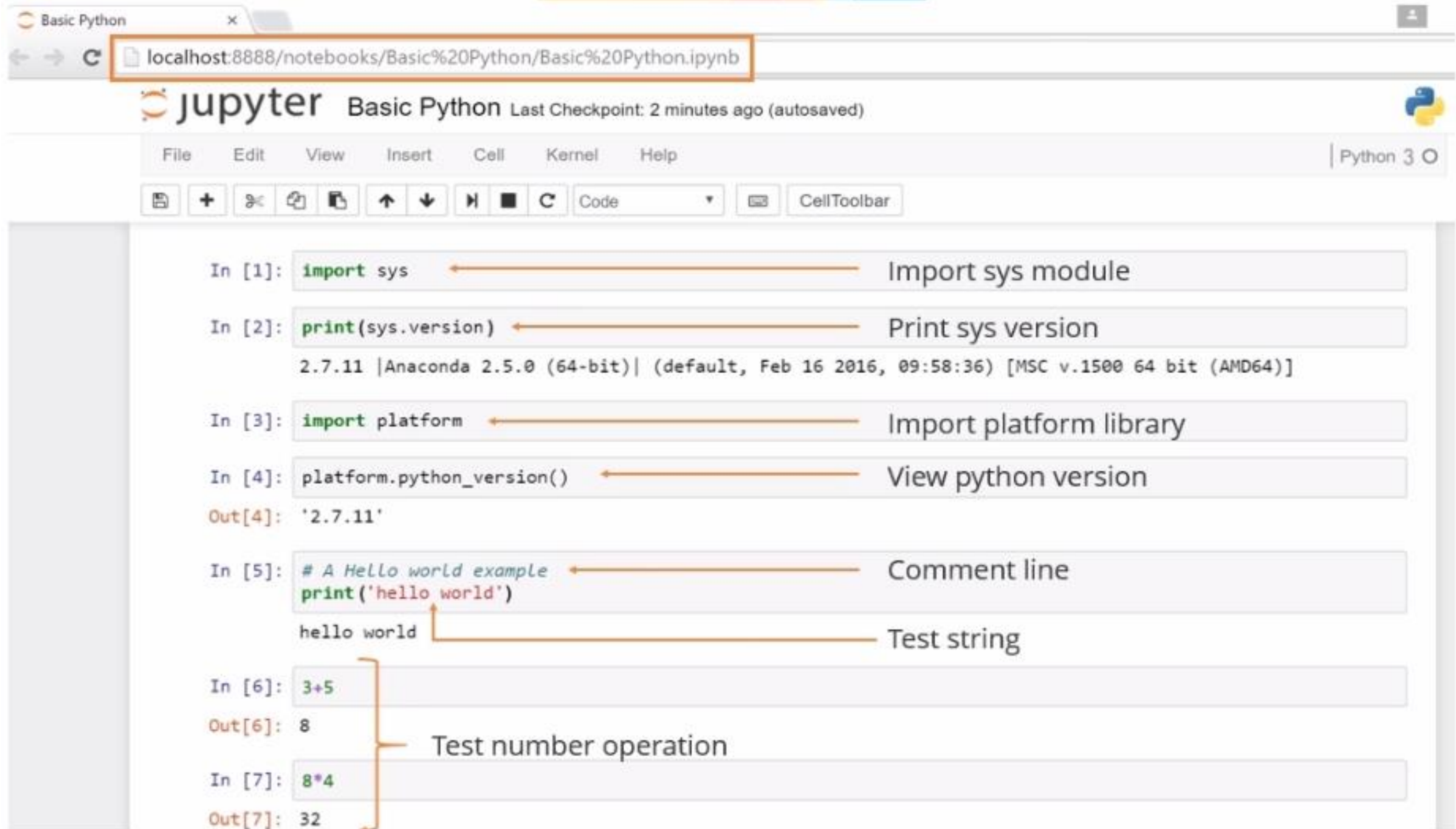
# Jupyter Notebook: Installation

To install Jupyter notebook on your system, type the command shown here on Anaconda prompt and press Enter to execute it.





# Getting Started



The screenshot displays a Jupyter Notebook titled "Basic Python" in a web browser. The address bar shows the URL `localhost:8888/notebooks/Basic%20Python/Basic%20Python.ipynb`. The notebook interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar with icons for file operations and code execution. The notebook content consists of several code cells, each with an input prompt (In [n]:) and an output (Out[n:]). Annotations with arrows point to specific parts of the code and output, explaining their purpose:

- In [1]:** `import sys` → Import sys module
- In [2]:** `print(sys.version)` → Print sys version  
Output: `2.7.11 |Anaconda 2.5.0 (64-bit)| (default, Feb 16 2016, 09:58:36) [MSC v.1500 64 bit (AMD64)]`
- In [3]:** `import platform` → Import platform library
- In [4]:** `platform.python_version()` → View python version  
Output: `'2.7.11'`
- In [5]:** `# A Hello world example` → Comment line  
`print('hello world')` → Test string  
Output: `hello world`
- In [6]:** `3+5` → Test number operation  
Output: `8`
- In [7]:** `8*4` → Test number operation  
Output: `32`

# Variables and Assignment

A variable can be assigned or bound to any value. Some of the characteristics of binding a variable in Python are listed here:

The variable refers to the memory location of the assigned value.

```
In [1]: x = 3  
        type(x)
```

Out[1]: int

The variable appears on the left, while the value appears on the right.

```
In [2]: y = 2.1  
        type(y)
```

Out[2]: float

The data type of the assigned value and the variable is the same.

```
In [3]: z = 'test'  
        type(z)
```

Out[3]: str

# Variables and Assignment: Example

Let us look at an example of how you can assign a value to a variable, and print it and its data type.

```
In [44]: first_string_variable = 'test'  
first_integer_variable = 123
```

Assignment

```
In [45]: print (first_string_variable)  
print (first_integer_variable)
```

test  
123

Variable data value

```
In [47]: print type (first_string_variable)  
print type (first_integer_variable)
```

<type 'str'>  
<type 'int'>

Data type of the object

# Multiple Assignments

You can access a variable only if it is defined. You can define multiple variables simultaneously.

In [48]: `number_example`

Access variable  
without assignment

```
-----  
NameError                                Traceback (most recent call last)  
<ipython-input-48-a856f233ae98> in <module>()  
----> 1 number_example  
  
NameError: name 'number_example' is not defined
```

In [49]: `number_example = 2`  
`number_example`

Access variable after  
assignment

Out[49]: 2

In [54]: `integer_x, integer_y = 5, 22`

In [55]: `integer_x`

Out[55]: 5

In [56]: `integer_y`

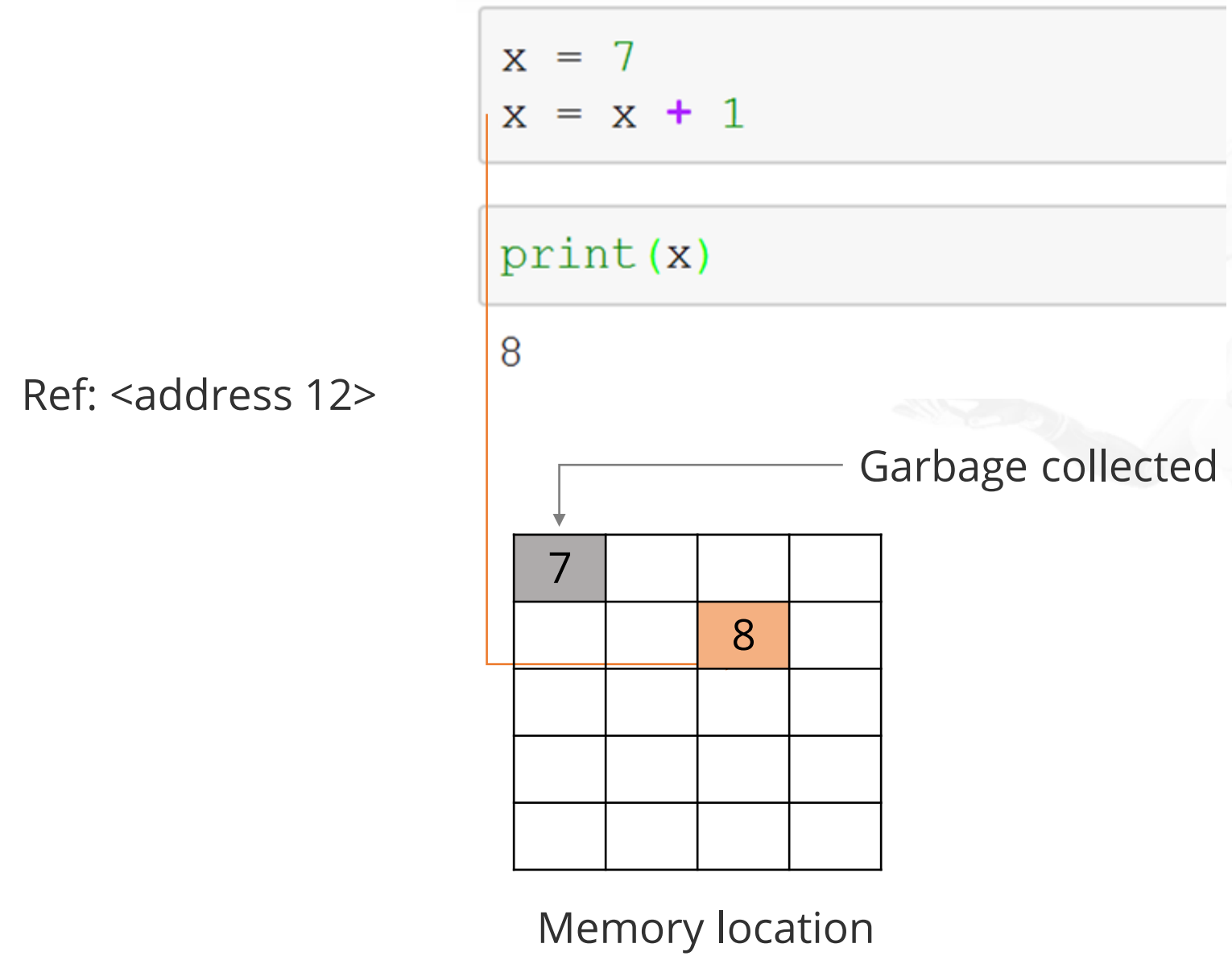
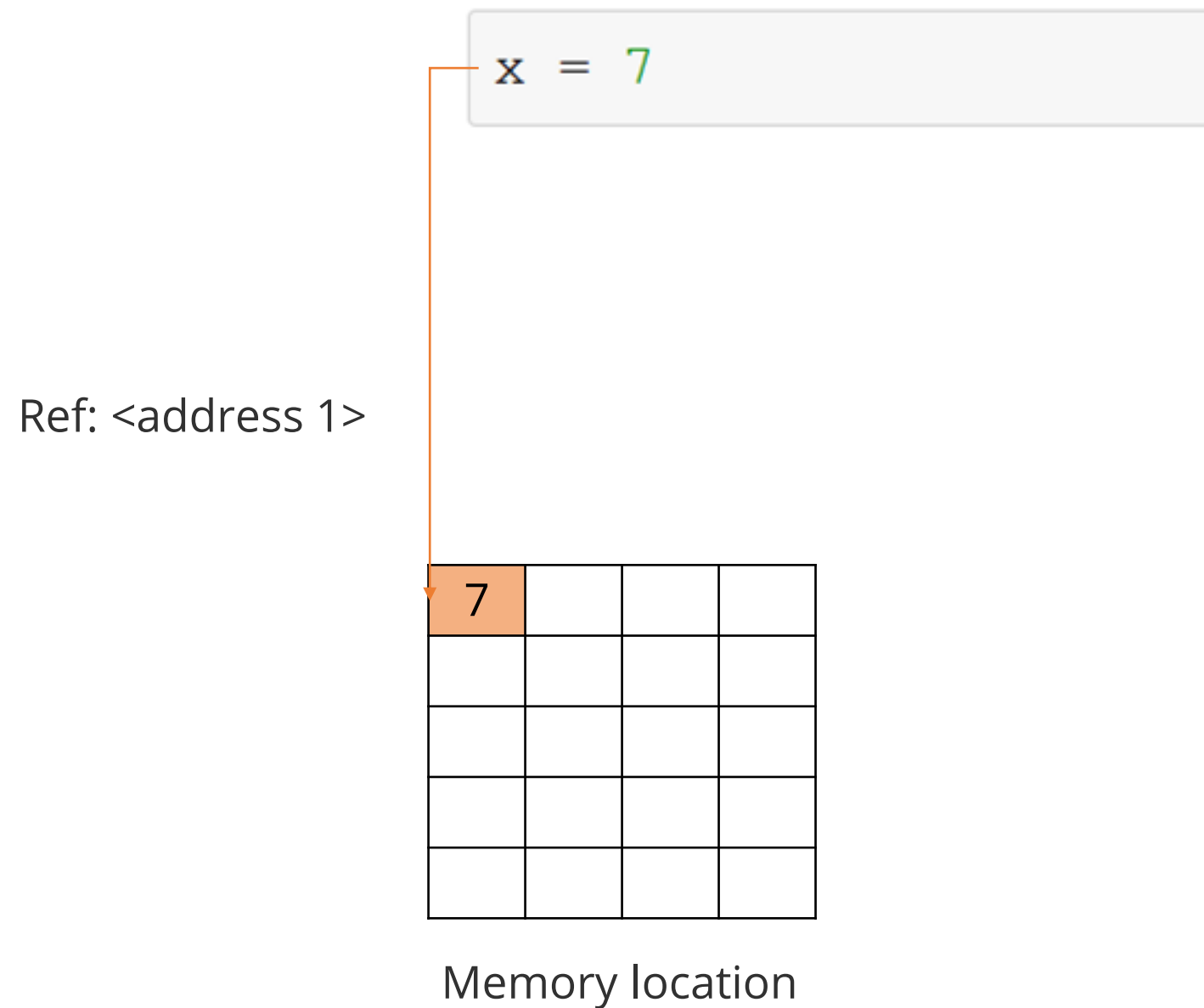
Out[56]: 22

Multiple assignments



# Assignment and Reference

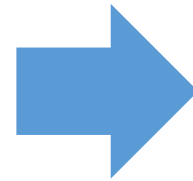
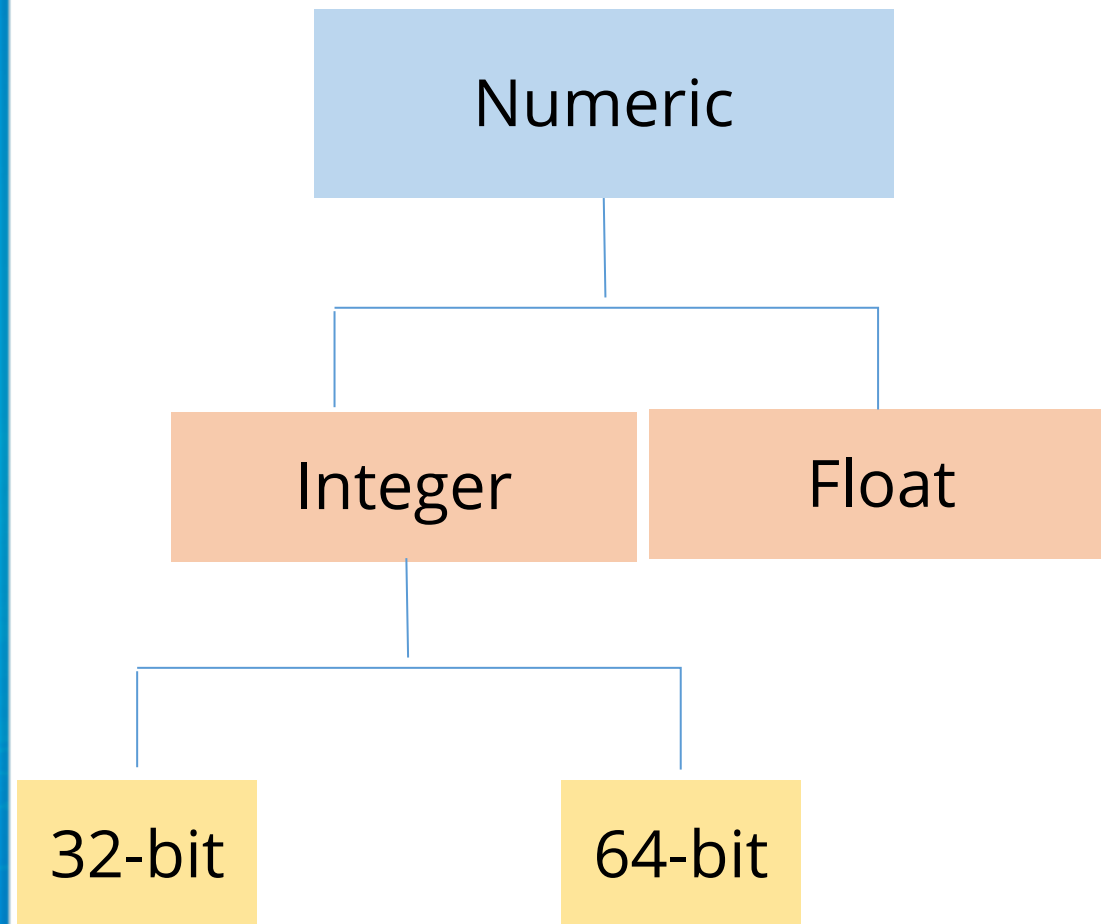
When a variable is assigned a value, it refers to the value's memory location or address. It does not equal the value itself.



## Data Types and Structures

# Basic Data Types: Integer and Float

Python supports various data types. There are two main numeric data types:



```
In [9]: int_number = 7/2  
int_number
```

Out[9]: 3 ← Integer value

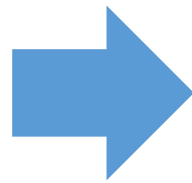
```
In [10]: float_number = 5.2/2  
float_number
```

Out[10]: 2.6 ← Float value

# Basic Data Types: String

Python has extremely powerful and flexible built-in string processing capabilities.

String



```
In [14]: string_one = 'first string'  
string_two = "second string"  
string_three = """third string"""
```

With single quote

With double quote

Three double quotes

```
In [15]: print (string_one)  
print (string_two)  
print (string_three)
```

Print string values

```
first string  
second string  
third string
```



# Basic Data Types: None and Boolean

Python also supports the **None** and Boolean data types.

In [102]: `num_x = None` ← Null value type  
`num_x is None`

Out[102]: `True` ← Boolean type

In [103]: `num_x = 10`  
`num_x is None`

Out[103]: `False` ← Boolean type

# Type Casting

You can change the data type of a number using type casting.

In [58]: `float_number = 3.6467` ← Float number

In [59]: `float_number`

Out[59]: 3.6467

In [60]: `int(float_number)` ← Type cast to integer

Out[60]: 3

In [61]: `str(float_number)` ← Type cast to string value

Out[61]: '3.6467'

# Data Structure: Tuple

A tuple is a one-dimensional, immutable ordered sequence of items which can be of mixed data types.

```
In [145]: first_tuple = (12, 'Jack', 45.6, 'new', (3, 2), 'test')
```

← Create a tuple

```
In [146]: first_tuple
```

```
Out[146]: (12, 'Jack', 45.6, 'new', (3, 2), 'test')
```

← View tuple

```
In [147]: first_tuple[1]
```

← Access the data at index value 1

```
Out[147]: 'Jack'
```

```
In [148]: first_tuple[1] = 'Mark'
```

← Try to modify the tuple

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-148-38afcb40e37> in <module>()  
----> 1 first_tuple[1] = 'Mark'
```

```
TypeError: 'tuple' object does not support item assignment
```

↑ Error: A tuple is immutable and can't be modified

# Data Structure: Accessing Tuples

You can access a tuple using indices.

```
In [1]: first_tuple = (12, 'Jack', 45.6, 'new', (3,2), 'test')
```

← Tuple

```
In [2]: #Accessing elements using a positive index  
#The index count starts from the left, with the first index being 0  
first_tuple[2]
```

Out [2]: 45.6

↑ Access with positive index

```
In [3]: #Accessing elements using a negative index  
#The index count starts from the right, with the first index being -1  
first_tuple[-3]
```

Out [3]: 'new'

↑ Access with negative index



# Data Structure: Slicing Tuples

You can also slice a range of elements by specifying the start and end indices of the desired range.

```
In [1]: first_tuple = (12, 'Jack', 45.6, 'new', (3,2), 'test') ← Tuple
```

```
In [4]: #Creating a subset/slice of the tuple  
#Specify the indices of the elements, separated by a colon  
#The first index is inclusive; the second index is exclusive  
first_tuple[1:4]
```

```
Out[4]: ('Jack', 45.6, 'new')
```

Count starts with the first index, but stops before the second index

```
In [5]: #You can use negative indices as well to slice a tuple  
#Count from the right, starting from -1, to specify the correct index  
first_tuple[1: -1]
```

```
Out[5]: ('Jack', 45.6, 'new', (3, 2))
```

Even for negative indices, the count stops before the second index

# Data Structure: List

A list is a one-dimensional, mutable ordered sequence of items which can be of mixed data types.

```
In [161]: first_list = ['Mark',101,23.6,'test',None,11] ← Create a list
```

```
In [162]: first_list ← View a list
```

```
Out[162]: ['Mark', 101, 23.6, 'test', None, 11]
```

```
In [163]: first_list.append('Jack') ← Modify a list: Add new items  
first_list
```

```
Out[163]: ['Mark', 101, 23.6, 'test', None, 11, 'Jack']
```

```
In [164]: first_list.remove('Mark') ← Modify a list: Remove items  
first_list
```

```
Out[164]: [101, 23.6, 'test', None, 11, 'Jack']
```

```
In [165]: first_list.pop(2) ← Access and remove list data using  
element indices
```

```
Out[165]: 'test'
```

```
In [166]: first_list.insert(1,'Smith') ← Modify a list: Insert a new item at a  
first_list          certain index
```

```
Out[166]: [101, 'Smith', 23.6, None, 11, 'Jack']
```

# Data Structure: Accessing Lists

Just like tuples, you can access elements in a list through indices.

```
In [5]: first_list
```

```
Out[5]: [101, 'Smith', 'Smith', 23.6, None, 11, 'Jack']
```

 ← New modified list

```
In [6]: #Accessing elements using a positive index  
#The index count starts from the left, with the first index being 0  
first_list[2]
```

```
Out[6]: 'Smith'
```

 ↑ Access with positive index

```
In [7]: #Accessing elements using a negative index  
#The index count starts from the right, with the first index being -1  
first_list[-2]
```

```
Out[7]: 11
```

 ↑ Access with negative index

# Data Structure: Slicing Lists

Similar to tuples, you can also slice lists through indices.

```
In [5]: first_list
```

```
Out[5]: [101, 'Smith', 'Smith', 23.6, None, 11, 'Jack']
```

 ← New modified list

```
In [8]: #Creating a subset/slice of the tuple  
#Specify the indices of the elements, separated by a colon  
#The first index is inclusive; the second index is exclusive  
first_list[1:4]
```

```
Out[8]: ['Smith', 'Smith', 23.6]
```

Count starts with the first index,  
but stops before the second index

```
In [9]: #You can use negative indices as well to slice a tuple  
#Count from the right, starting from -1, to specify the correct index  
first_list[1:-1]
```

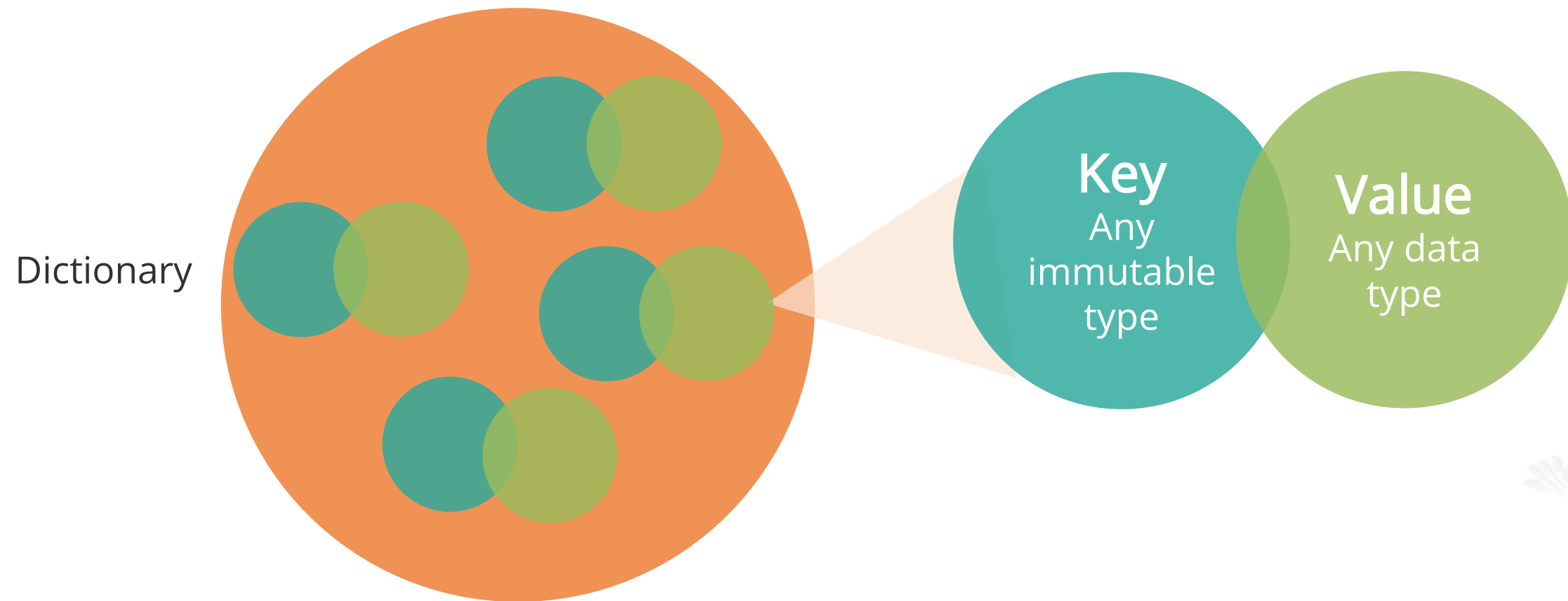
```
Out[9]: ['Smith', 'Smith', 23.6, None, 11]
```

Even for negative indices, the count  
stops before the second index



# Data Structure: Dictionary (dict)

Dictionaries store a mapping between a set of keys and a set of values.



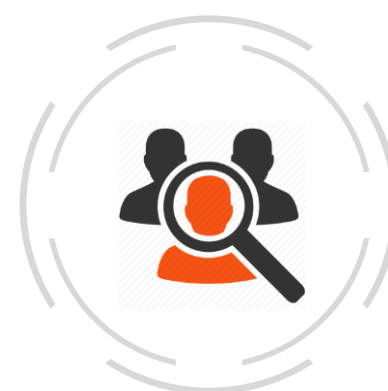
Define



Modify



View



Lookup



Delete



# Data Structure: View Dictionaries

You can view the keys and values in a dict, either separately or together, using the syntax shown here.



In [215]:	<code>first_dict = {'John': 'john@abc.com', 'Kelly': 'kelly@xyz.org', 'id': [23, 81]}</code>	← Create a dictionary
In [216]:	<code>first_dict</code>	← View entire dictionary
Out[216]:	<code>{'John': 'john@abc.com', 'Kelly': 'kelly@xyz.org', 'id': [23, 81]}</code>	
In [217]:	<code>first_dict.keys()</code>	← View only keys
Out[217]:	<code>['Kelly', 'John', 'id']</code>	
In [218]:	<code>first_dict.values()</code>	← View only values
Out[218]:	<code>['kelly@xyz.org', 'john@abc.com', [23, 81]]</code>	

# Data Structure: Access and Modify dict Elements

You can also access and modify individual elements in a dict.

```
In [219]: first_dict['Kelly']
```

```
Out[219]: 'kelly@xyz.org'
```

```
In [220]: first_dict['id']
```

```
Out[220]: [23, 81]
```

```
In [221]: first_dict.update({'id':[32,55]})
```

```
In [222]: first_dict
```

```
Out[222]: {'John': 'john@abc.com', 'Kelly': 'kelly@xyz.org', 'id': [32, 55]}
```

```
In [223]: del first_dict['id']
```

```
In [224]: first_dict
```

```
Out[224]: {'John': 'john@abc.com', 'Kelly': 'kelly@xyz.org'}
```

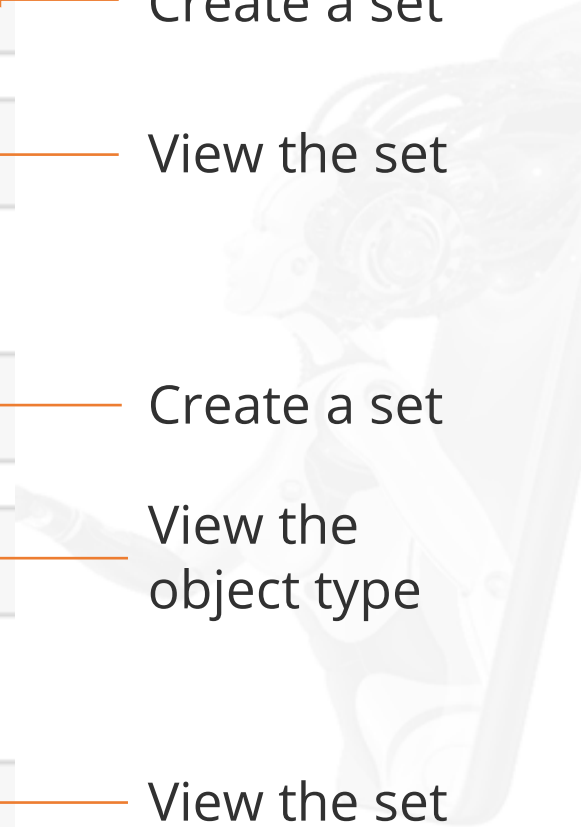
Access with key

Modify dictionary:  
update

Modify dictionary:  
delete

# Data Structure: Set

A set is an unordered collection of unique elements.



```
In [327]: auto_survey = set(['Audi', 'BMW', 'BMW', 'Ferrari', 'GM', 'Mercedes', 'Cheverolet', 'GM'])
```

← Create a set

```
In [328]: auto_survey
```

← View the set

```
Out[328]: {'Audi', 'BMW', 'Cheverolet', 'Ferrari', 'GM', 'Mercedes'}
```

```
In [329]: auto_survey_set = {'Audi', 'BMW', 'BMW', 'Ferrari', 'GM', 'Mercedes', 'Cheverolet', 'GM'}
```

← Create a set

```
In [330]: type(auto_survey_set)
```

← View the object type

```
Out[330]: set
```

```
In [331]: auto_survery_set
```

← View the set

```
Out[331]: {'Audi', 'BMW', 'Cheverolet', 'Ferrari', 'GM', 'Mercedes'}
```

# Data Structure: Set Operations

Let us look at some basic set operations.

```
In [334]: auto_survery_1 = set(['Audi', 'BMW', 'BMW', 'Ferrari', 'GM', 'Mercedes', 'Cheverolet', 'GM', 'Toyota'])  
auto_survery_2 = set(['BMW', 'Ferrari', 'GM', 'Hyundai', 'Kia', 'Cheverolet', 'GM', 'Ford', 'Toyota', 'Zen'])
```

Create sets

```
In [335]: combined_survery_report = auto_survery_1 | auto_survery_2
```

OR – Union  
set operation

```
In [336]: combined_survery_report
```

```
Out[336]: {'Audi',  
          'BMW',  
          'Cheverolet',  
          'Ferrari',  
          'Ford',  
          'GM',  
          'Hyundai',  
          'Kia',  
          'Mercedes',  
          'Toyota',  
          'Zen'}
```

View the output of the OR  
operation

```
In [337]: common_survey_report = auto_survery_1 & auto_survery_2
```

AND – Intersection set operation

```
In [338]: common_survey_report
```

```
Out[338]: {'BMW', 'Cheverolet', 'Ferrari', 'GM', 'Toyota'}
```

View the output of the  
**AND** operation

# Basic Operator: in

The `in` operator is used to generate a Boolean value to indicate whether a given value is present in the container or not.

```
In [225]: student_list = ['Tom', 'Jack', 'Nick', 'Sarah', 'Nicole']
```

 ← Create a list

```
In [226]: 'Nick' in student_list
```

```
Out[226]: True
```

```
In [227]: 'Mark' in student_list
```

```
Out[227]: False
```

Test presence of string  
with `in` operator

```
In [228]: word = 'encyclopedia'
```

 ← Create a string

```
In [229]: 't' in word
```

```
Out[229]: False
```

```
In [230]: 'i' in word
```

```
Out[230]: True
```

Test presence of substrings  
with `in` operator



## Basic Operator: +

The **plus** operator produces a new tuple, list, or string whose value is the concatenation of its arguments.

```
In [239]: test_score_1 = (68, 96, 71)
          test_score_2 = (92, 87, 83)
```

Create tuples

```
In [240]: test_score = test_score_1 + test_score_2
          test_score
```

Add tuples

```
Out[240]: (68, 96, 71, 92, 87, 83)
```

```
In [241]: country_list_1 = ['USA', 'UK', 'China', 'Brazil', 'Mexico']
          country_list_2 = ['Australia', 'Spain', 'Italy']
```

Create lists

```
In [242]: country_list_final = country_list_1 + country_list_2
          country_list_final
```

Add lists

```
Out[242]: ['USA', 'UK', 'China', 'Brazil', 'Mexico', 'Australia', 'Spain', 'Italy']
```

```
In [243]: first_name = 'George'
          last_name = 'Washington'
```

Create strings

```
In [244]: full_name = first_name + ' ' + last_name
          full_name
```

Concatenate strings

```
Out[244]: 'George Washington'
```

## Basic Operator: \*

The **multiplication** operator produces a new tuple, list, or string that **repeats** the original content.

```
In [249]: age = (12,17,9) * 3  
age
```

\* operator with tuple

```
Out[249]: (12, 17, 9, 12, 17, 9, 12, 17, 9)
```

```
In [250]: ID = [101,23,77,45] * 2  
ID
```

\* operator with list

```
Out[250]: [101, 23, 77, 45, 101, 23, 77, 45]
```

```
In [251]: name = 'friend'*3  
name
```

\* operator with string

```
Out[251]: 'friendfriendfriend'
```



The \* operator does not actually multiply the values; it only repeats the values for the specified number of times.

## Functions

# Functions

Functions are the primary and most important method of code organization and reuse in Python.

## Syntax

```
def <name>(arg1, arg2, ..., argN):  
    <statements>  
    return <value>
```

## Properties

- Outcome of the function is communicated by return statement
- Arguments in parenthesis are basically assignments

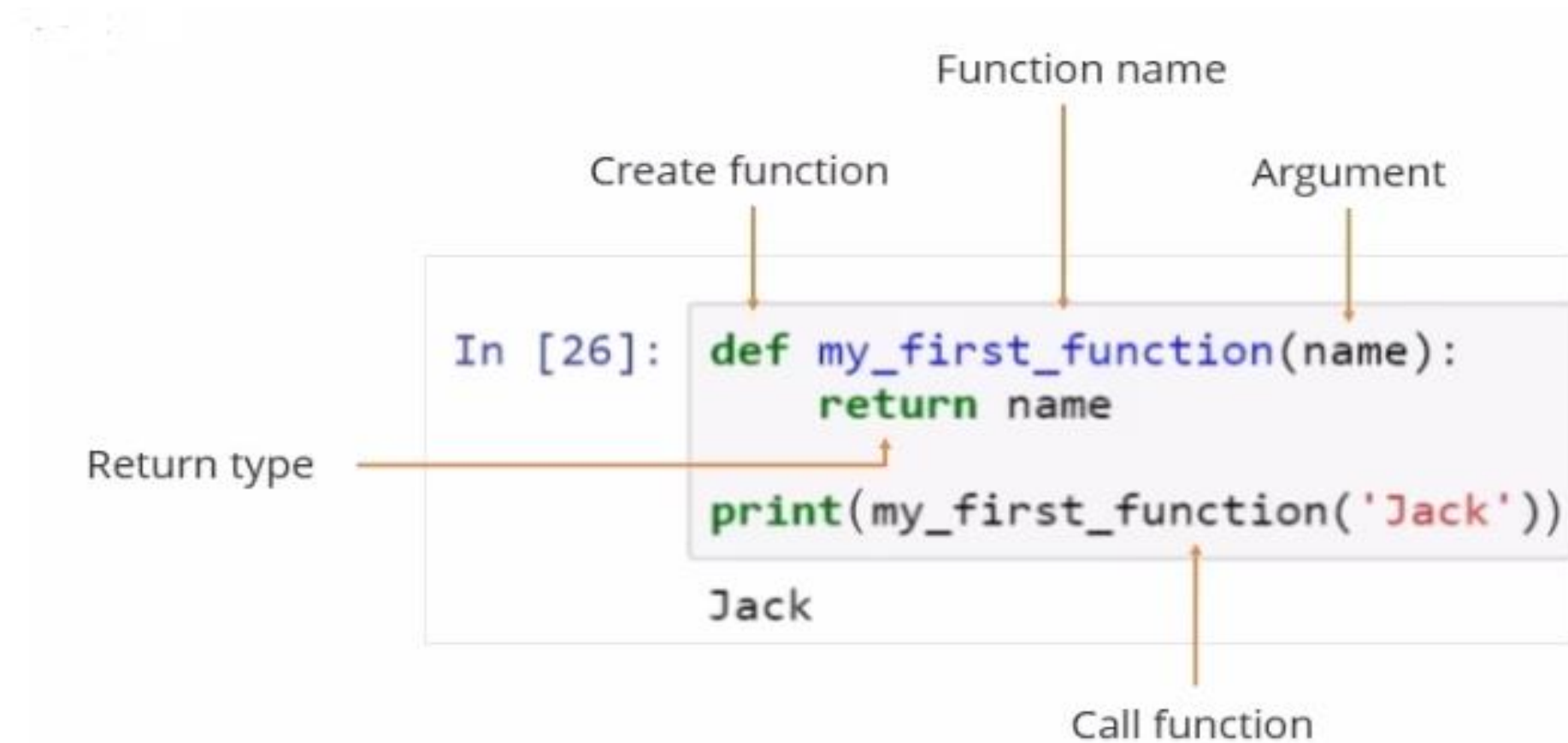


Use **def** to create a function and assign it a name.

# Functions: Considerations

Some important points to consider while defining functions:

- A function should always have a **return** value.
- If **return** is not defined, then it returns **None**.
- Function overloading is not permitted.





# Functions: Returning Values

You can use a function to return a single value or multiple values.

In [256]: `def add_two_numbers(num1, num2):` ← Create function  
 `return num1+num2`

`number1 = 23`  
`number2 = 47.5`  
`result = add_two_numbers(number1, number2)` ← Call function  
`result`

Out[256]: 70.5

In [257]: `def profile():` ← Create function  
 `age = 21`  
 `height = 5.5`  
 `weight = 130`  
 `return age, height, weight` ← Multiple return

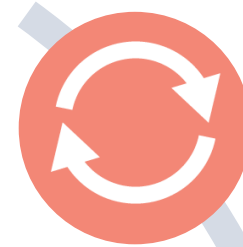
`age, height, weight = profile()` ← Call function

In [258]: `print (age, height, weight)`

21 5.5 130

# Built-in Sequence Functions

The built-in sequence functions of Python are:



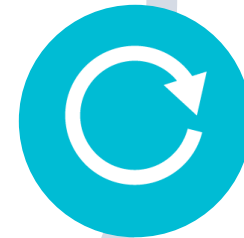
## **enumerate**

Indexes data to keep track of indices and corresponding data mapping



## **sorted**

Returns the new sorted list for the given sequence



## **reversed**

Iterates the data in reverse order



## **Zip**

Creates lists of tuples by pairing up elements of lists, tuples, or other sequence



# Built-in Sequence Functions: enumerate

```
In [20]: store_list = ['McDonald', 'Taco Bell', 'Dunkin', 'Wendys', 'Chipotle' ]
```

← List of food stores

```
In [21]: for position, name in enumerate(store_list):  
         print(position, name)
```

```
0 McDonald  
1 Taco Bell  
2 Dunkin  
3 Wendys  
4 Chipotle
```

← Print data element and index using enumerate method

```
In [22]: store_map = dict((name, position) for position, name in enumerate(store_list))
```

```
In [23]: store_map
```

← Create a data element and index map using dict

```
Out[23]: {'Chipotle': 4, 'Dunkin': 2, 'McDonald': 0, 'Taco Bell': 1, 'Wendys': 3}
```

← View the store map in the form of key-value pair

# Built-in Sequence Functions: sorted

In [27]: `sorted([91,43,65,56,7,33,21])` ← Sort numbers

Out[27]: [7, 21, 33, 43, 56, 65, 91]

In [28]: `sorted('the data science')` ← Sort a string value

Out[28]: [' ',  
' ',  
'a',  
'a',  
'c',  
'c',  
'd',  
'e',  
'e',  
'e',  
'h',  
'i',  
'n',  
's',  
't',  
't']



## Built-in Sequence Functions: reversed and zip

In [50]: `num_list = range(15)` ← Create a list of numbers for range 15

In [51]: `list(reversed(num_list))` ← Use reversed function to reverse the order

Out[51]: `[14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]`

In [52]: `subjects = ['math', 'statistics', 'algebra']` ← Define list of subjects and count  
`subject_count = ['one', 'two', 'three']`

In [53]: `total_subject = zip(subjects, subject_count)` ← Zip function to pair the data elements of lists  
`total_subject`

Out[53]: `[('math', 'one'), ('statistics', 'two'), ('algebra', 'three')]` ← Returns list of tuples

In [54]: `type(total_subject)` ← View type

Out[54]: `list`



# Control Flow Statements: if, elif, else

The **if**, **elif**, and **else** statements are the most commonly used control flow statements.

```
In [341]: age = 21
```

```
In [342]: if age<18:
           print('minor')
           else:
           print('adult')
```

adult

```
In [343]: marks = 81
```

```
In [344]: if marks>90:
           print('grade A')
           elif 80<=marks<=90:
           print('grade B')
           elif 70<=marks<=80:
           print('grade c')
           elif 60<=marks<=70:
           print('grade d')
           else:
           print('grade f')
```

grade B

If condition

Else block

Nested if, elif, and else

# Control Flow Statements : for Loops

A for loop is used to iterate over a collection (like a list or tuple) or an iterator.

```
In [278]: stock_tickers =['AAPL','MSFT','GOOGL',None,'AMZN','CSCO','ORCL']
```

```
In [279]: for tickers in (stock_tickers):  
          if(tickers is None):  
              continue  
          print('tickers')
```

For loop iterator

The **continue** statement

AAPL  
MSFT  
GOOGL  
AMZN  
CSCO  
ORCL

```
In [280]: for tickers in (stock_tickers):  
          if(tickers is None):  
              break  
          print('tickers')
```

The **break** statement

AAPL  
MSFT  
GOOGL

# Control Flow Statements : while Loops

A while loop specifies a condition and a block of code that is to be executed until the condition evaluates to False or the loop is explicitly ended with break.

```
In [283]: temperature = 100  
while temperature > 95:  
    print(temperature)  
    temperature = temperature - 1
```

While condition

```
100  
99  
98  
97  
96
```

# Control Flow Statements : Exception Handling

Handling Python errors or exceptions gracefully is an important part of building robust programs and algorithms.

```
In [307]: def test_float(number):  
         return float(number)
```

Create function

```
In [308]: test_float(7.32453)
```

```
Out[308]: 7.32453
```

```
In [309]: test_float('test float')
```

Pass wrong argument type

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-309-d3d4bead5bfb> in <module>()  
----> 1 test_float('test float')  
  
<ipython-input-307-c9efb2931c9f> in test_float(number)  
      1 def test_float(number):  
----> 2     return float(number)  
  
ValueError: could not convert string to float: test float
```

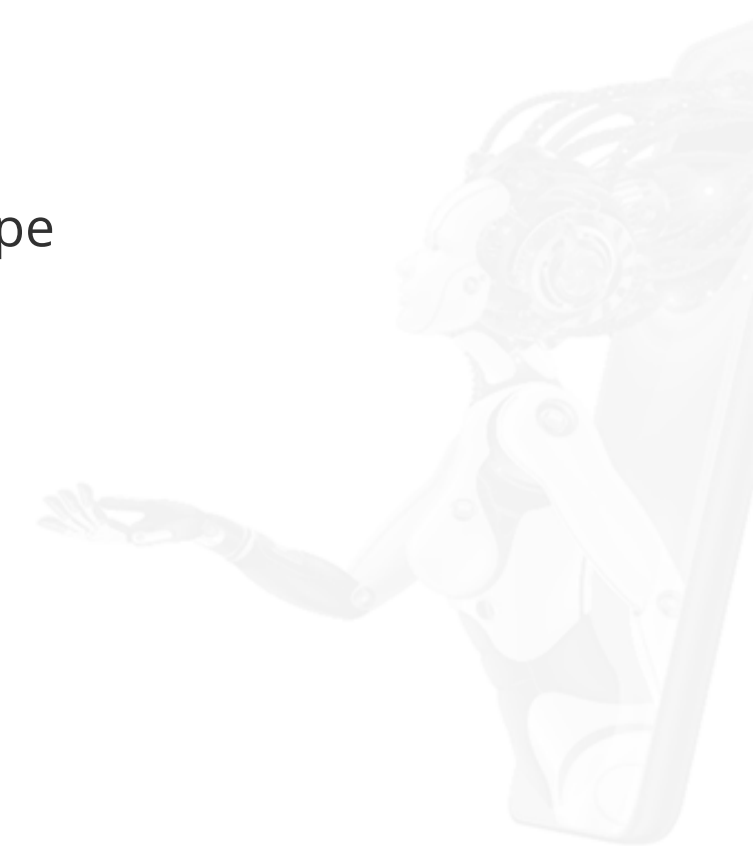
Error

```
In [310]: def test_float(number):  
         try:  
             return float(number)  
         except ValueError:  
             return 'not a number, the input value is', number
```

Exception handling with try-except block

```
In [311]: test_float('test')
```

```
Out[311]: ('not a number, the input value is', 'test')
```



# DATA AND ARTIFICIAL INTELLIGENCE



## Knowledge Check



## Knowledge Check

1

What is the data type of the object  $x = 3 * 7.5$ ?

- a. Int
- b. Float
- c. String
- d. None of the above



## Knowledge Check

1

What is the data type of the object  $x = 3 * 7.5$ ?

- a. Int
- b. Float
- c. String
- d. None of the above



The correct answer is **b**

Since one of the operands is float, the  $x$  variable will also be of the float data type.

## Knowledge Check

2

Which of the data structures can be modified? Select all that apply.

- a. tuple
- b. list
- c. dict
- d. set



## Knowledge Check

2

Which of the data structures can be modified? Select all that apply.

- a. tuple
- b. list
- c. dict
- d. set



The correct answer is **b, c, d**

Only a tuple is immutable and cannot be modified. All the other data structures can be modified.

## Knowledge Check

3

What will be the output of the following code?

```
In [350]: summit_venue = ['NYC', 'LA', 'Miami', 'London', 'Madrid', 'Paris']  
summit_venue[3:-1]
```

- a. ['NYC', 'Madrid']
- b. ['London', 'Madrid']
- c. ['Miami', 'Madrid']
- d. ['Miami', 'Paris']





## Knowledge Check

3

What will be the output of the following code?

```
In [350]: summit_venue = ['NYC', 'LA', 'Miami', 'London', 'Madrid', 'Paris']  
summit_venue[3:-1]
```

- a. ['NYC', 'Madrid']
- b. ['London', 'Madrid']
- c. ['Miami', 'Madrid']
- d. ['Miami', 'Paris']



The correct answer is **b**

Slicing starts at the first index and stops before the second index. Here, the element at index 3 is **London** and the element before index -1 is **Madrid**.

Knowledge  
Check

4

Which of the following data structures is preferred to contain a unique collection of values?

- a. dict
- b. list
- c. set
- d. tuple



## Knowledge Check

4

Which of the following data structures is preferred to contain a unique collection of values?

- a. dict
- b. list
- c. set
- d. tuple



The correct answer is **c**

A set is used when a unique collection of values is desired.

## Key Takeaways

You are now able to:

- 🕒 Explain Anaconda and Jupyter notebook installation
- 🕒 List the important data types supported by Python
- 🕒 Discuss data structures, such as lists, tuples, sets, and dicts
- 🕒 Explain slicing and accessing the four data structures
- 🕒 Discuss basic operators and functions
- 🕒 Outline the important control flow statements



**Thank You**