Python Workshop for Beginners

MC Pang

September 8, 2022

Table of Contents

Introduction

Introduction

- Python is a programming language pioneered by Guido van Rossum and released in 1991.
- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi)
- Python is open-source and has a large collection of community-driven packages.
  - Example packages: matplotlib, pandas, plotly, scikit-learn, keras, numpy, scipy, statsmodels,
- Python runs in a procedural (from top to bottom), object-oriented or functional manner.

Example

```
import pandas as pd
import matplotlib.pyplot as plt
```

For what applications can we use Python?

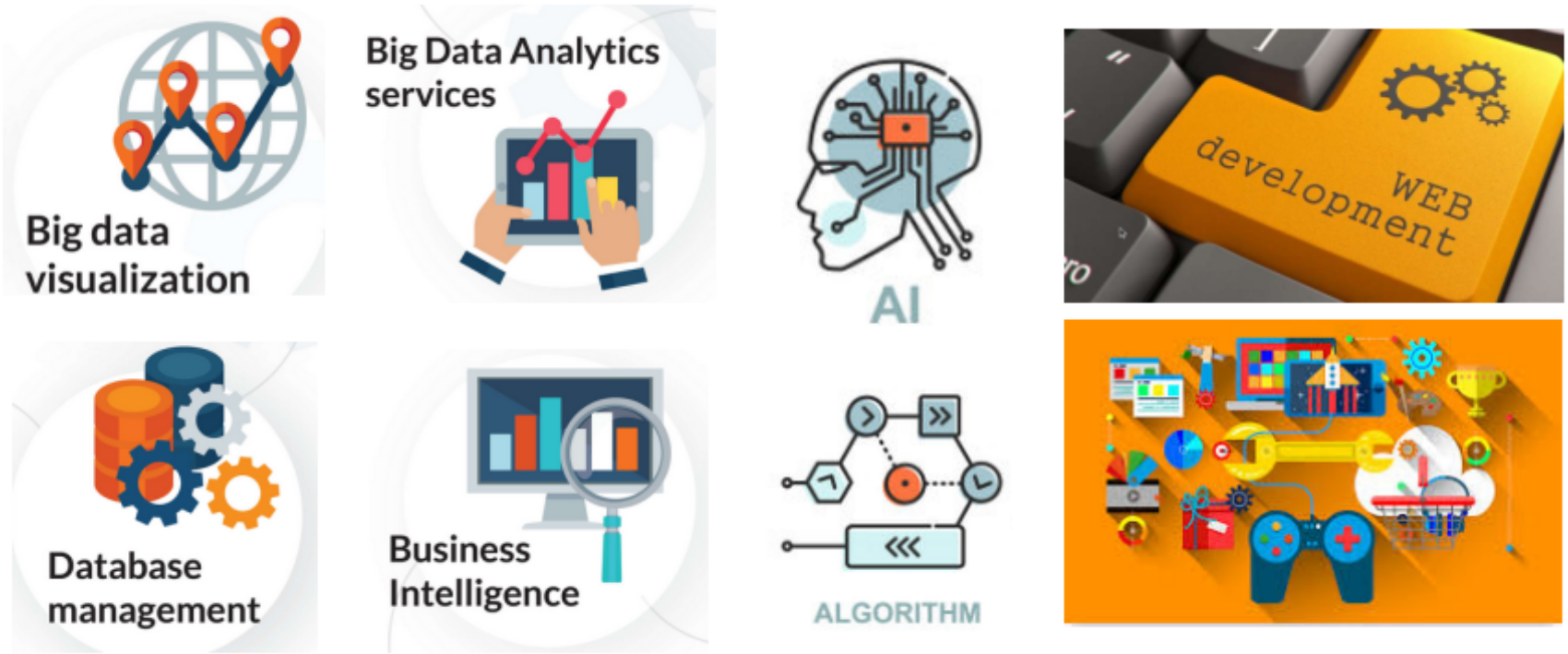


Figure: Python has a large collection of community-driven packages that enables diverse applications.

Installation and setup

Working environment installation and setup

- Download Anaconda
  - from https://www.anaconda.com/products/distribution (Python 3.9, 64-Bit Graphical Installer)
  - Launch Powershell Prompt and type *conda create -n python_workshop python=3.9 anaconda*
  - Activate the virtual environment with *conda activate python_workshop*
  - From the Powershell: type *jupyter-notebook*
- Download git from
  - https://git-scm.com/download/win (Windows)
  - https://git-scm.com/download/mac (Mac)
- Setup a new Github account on https://github.com/

Basics

Print statement

In [1]:
```
# Print a message
print('Hello, world!')

# Check datatype
print(type('Hello, world!'))
```

```
Hello, world!
<class 'str'>
```

In [2]:
```
# Print multiple values of different types
ndays = 365
print('There are', ndays, 'in a year!')

# Check datatypes
print(type(ndays))
```

```
There are 365 in a year!
<class 'int'>
```

In [3]:
```
# Asking the user for an input
name = input('What is your name?\n')
```

```
What is your name?
David
```

In [4]:
```
age = int(input('Enter your age:\n'))
```

```
Enter your age:
18
```

Comments in Python

- We should use comments to explain:
  - The reasons behind our codes
  - How our codes work
- Comments in Python start with the symbol #.
- Recommended best practice (PEP8): Limit the line length of comments and docstrings to 72 characters.

Comments in Python

Single-line comment

In [5]:
```
# Single-line comment
a = 100
b = 50 # This is an in-line comment
```

Both lines above are bad practice for coding because they don't explain the reasoning behind the variable assignment.

What do `a` and `b` mean?

In [6]:
```
# Better code
# This basket has 20 appples and 70 bananas.
apple_count = 20
banana_count = 70

print(apple_count)
print(banana_count)
```

```
20
70
```

Comments in Python

Multiline comments

In [7]:
```
# Multiline comment
# This is a comment
# written in
# more than just one line
```

In [8]:
```
"""
This is a comment
written in
more than just one line
"""
```

Out[8]:
```
'\nThis is a comment\nwritten in\nmore than just one line\n'
```

String manipulations

String basics

A string consists of multiple characters joined together by single, double or triple quotes.

In [16]:
```
# Example strings:

country = 'united kingdom'
city = "london"
book_name = '''lord of the rings'''
postcode = 'se8 4ln'
```

In [17]:
```
type(postcode)
```

Out[17]:
```
str
```

In [18]:
```
# Join different strings
city + ' ' + country
```

Out[18]:
```
'london united kingdom'
```

In [19]:
```
# Mutiply strings by integers
'Please wait...'*3
```

Out[19]:
```
'Please wait...Please wait...Please wait...'
```

String basics

When we enclose numeric characters with quotes, the numeric characters will be interpreted as type string instead of integers or floats.

In [20]:
```
# Check the length of the string
len('Please wait...')
```

Out[20]:
```
14
```

In [21]:
```
# Convert string into integers
# type string
num_in_str = '100'
print(f'{num_in_str} has type {type(num_in_str)}.')

# type integer
num_int = int(num_in_str)
print(f'{num_int} has type {type(num_int)}.')
```

```
100 has type <class 'str'>.
100 has type <class 'int'>.
```

String methods

Python has useful string methods that could be used to automate strings processing (see the code snippets on the github repos for a comprehensive summary).

String methods

- capitalize(): Converts first character to upper case and the rest to lowercase.
- title(): Converts the first character of each word to upper case
- upper(): Convers all characters to uppercase
- lower(): Convers all characters to lowercase
- replace(old-substring, new-substring): Replace or delete old subtext with new subtext.

Examples of string methods

In [26]:
```
country = 'united kingdom'
city = "london"
book_name = '''lord of the rings'''
postcode = 'se8 4ln'


# converts the first character of each word to upper case
print(book_name.title())

# converts first character to upper case and the rest to lowercase.
print(city.capitalize())

# convers all characters to uppercase
print(postcode.upper())

# replace characters in a given string
print(book_name.replace('rings','flies'))

# deletes characters from the string
print(country.replace('united ',''))
```

```
Lord Of The Rings
London
SE8 4LN
lord of the flies
kingdom
```

String formatting

String formatting is inserting a string saved as a variable into another string.
Useful for:

- Creating a template
- Printing a variable to output
- Logging and debugging codes

Python has three ways to format strings:

String formatting methods

- Use %s to convert any variable into a string and insert into another string.
- Use .format()
- Use f-strings (my favourite)

String formatting

Why do we need to use string formatting?

In [27]:
```
print('The new student class 6A is Kevin.')
print('The student is 25 years old.')
print('*'*50)

print('The new student in class 6A is Megan.')
print('The student is 18 years old.')
print('*'*50)

print('The new student in class 6A is Timothy.')
print('The student is 21 years old.')
print('*'*50)
```

```
The new student class 6A is Kevin.
The student is 25 years old.
**************************************************
The new student in class 6A is Megan.
The student is 18 years old.
**************************************************
The new student in class 6A is Timothy.
The student is 21 years old.
**************************************************
```

String formatting

Instead of repeating the same print statement multiple times, we could create an in-place variable for each new student and their age.

In [28]:
```
# Method 1: Use %
# Student number-1
student_name = 'Kevin'
student_age = '25'

# Method 1: use %s to create placeholder at specified location within the string
# Use %variable_name at the end of the string
print('String formatting with method 1:')
print('The new student in class 6A is %s.' %student_name)
print('The student is %s years old.' %student_age)
print('*'*50)

# Method 2: Use {} to create a placeholder at specific location within the string
# At the end of each string, use .format(variable name)
print('String formatting with method 2:')
print('The new student in class 6A is {}.'.format(student_name))
print('The student is {} years old.'.format(student_age))
print('*'*50)

# Method 3: Use f-string to create a placeholder directly
# Start each string with the letter f
print('String formatting with method 3:')
print(f'The new student in class 6A is {student_name}.')
print(f'The student is {student_age} years old.')
print('*'*50)
```

```
String formatting with method 1:
The new student in class 6A is Kevin.
The student is 25 years old.
**************************************************
String formatting with method 2:
The new student in class 6A is Kevin.
The student is 25 years old.
**************************************************
String formatting with method 3:
The new student in class 6A is Kevin.
The student is 25 years old.
**************************************************
```

Review String Manipulation Understanding

Exercise 1

Create a dynamic user profile considering their

- Name,
- Age,
- School or company's name,
- House address,
- Email address.

Numbers and basic numeric operations

Numbers in Python

Two commonly used numeric types are: integers (real numbers without fractional components, including zero) and floats (numbers with decimal points).

In [1]:
```
# To create an integer:
integer_one = 10
print(f'{integer_one} has type {type(integer_one)}')


# or use the int callable:
integer_two = int(-10)
print(f'{integer_two} has type {type(integer_two)}')
```

```
10 has type <class 'int'>
-10 has type <class 'int'>
```

In [2]:
```
# To create a float
float_num = 3.0
print(float_num)

# To convert integer to float
# use the float callable
float_converted = float(integer_one)
print(float_converted)
print(type(float_converted))
```

```
3.0
10.0
<class 'float'>
```

Selected numeric operations

Operation	Meaning
$x + y$	The sum of x and y
$x - y$	The difference of x and y
$x * y$	The product of x and y
x / y	The quotient of x and y
$x // y$	The floored quotiet of x and y
$x \% y$	The remainder of x/y
$\mathrm{abs}(x)$	The absolute value of x
$\mathrm{int}(x)$	x converted to integer
$\mathrm{divmod}(x, y)$	The pair ($x//y$ and $x \% y$)
$\mathrm{float}(x)$	x converted to floating point
$x * *y$	x to the power of y

Table: Commonly used numeric operations.
Link: https://docs.python.org/3/library/stdtypes.html

Examples of numeric operations

In [3]:
```
x = 10
y = 3

print(f'The floored quotient of x and y is {x // y}')
print(f'The remainder of x and y is {x % y}')
print(f'The divmod of x and y is {divmod(x, y)}')
```

```
The floored quotient of x and y is 3
The remainder of x and y is 1
The divmod of x and y is (3, 1)
```

When creating and assigning variables:

- Use a lower case single letter, word or words. Separate words by underscores to improve the readability. Use explicit naming whenever possible!
- Examples: *a*, *even_nums*, *student_count*

Review numeric operations

Exercise-2

Create a simple currency conversion calculator to convert MYR to

- GBP
- EUR
- USD
- Japanese Yen

Data structures in Python

General characteristics of data structures

There are four different types of data structures in Python: **list**, **dictionary**, **set** and **tuple**. Common characteristics of a data structure:

- **Ordered**: The orders of the element are maintained with each insertion.
- **Sequential**: The data structure can be iterated over.
- **Heterogeneous**: The elements within the data structure can have different data types such as a mixture of string with integers and float.
- **indexed**: Every element within the data structure has an associated index.
- **mutable**: Elements are can be added, deleted or updated **after** the creation of the data structure.
- **unique**: The same element cannot be stored more than once within the data structure.

List

A list stores data as comma-separated elements using square brackets - [element 1, element 2, element 3]. It is an ordered, sequential, heterogeneous, indexed, mutable and non-unique data structure.

In [1]:
```
# creates a list with square brackets
# a list of numbers
list_of_nums = [10, 20, 30]
print(list_of_nums)

# type int converted to type list
print(type(list_of_nums))
print('*'*50)

# list of names
names = ['jeff', 'tom', 'grace', 'joyce', 'tan', 'yi']
print(names)

# type string converted to type list
print(type(names))
```

```
[10, 20, 30]
<class 'list'>
**************************************************
['jeff', 'tom', 'grace', 'joyce', 'tan', 'yi']
<class 'list'>
```

In [2]:
```
# creates a list with the built-in list function
# converts a string into a list of characters
list_of_strings = list('pokemon')
list_of_strings
```

Out[2]:
```
['p', 'o', 'k', 'e', 'm', 'o', 'n']
```

List

- A list can have heterogeneous data types. Nevertheless, when a list is constructured, the type of a heterogeneous list becomes a list!
- A list can store duplicated elements.

In [3]:
```
# A list can be empty without element
empty_list = []
print(empty_list)
```

```
[]
```

In [4]:
```
# heterogenenous list
mixed_list = [100, 'names', 300.2052]
mixed_type = [type(100), type('names'), type(300.2052)]

print(mixed_list)
print(mixed_type)

# regardless of the type of each element,
# when a list is constructed, the datatype becomes type li
print(type(mixed_type))
```

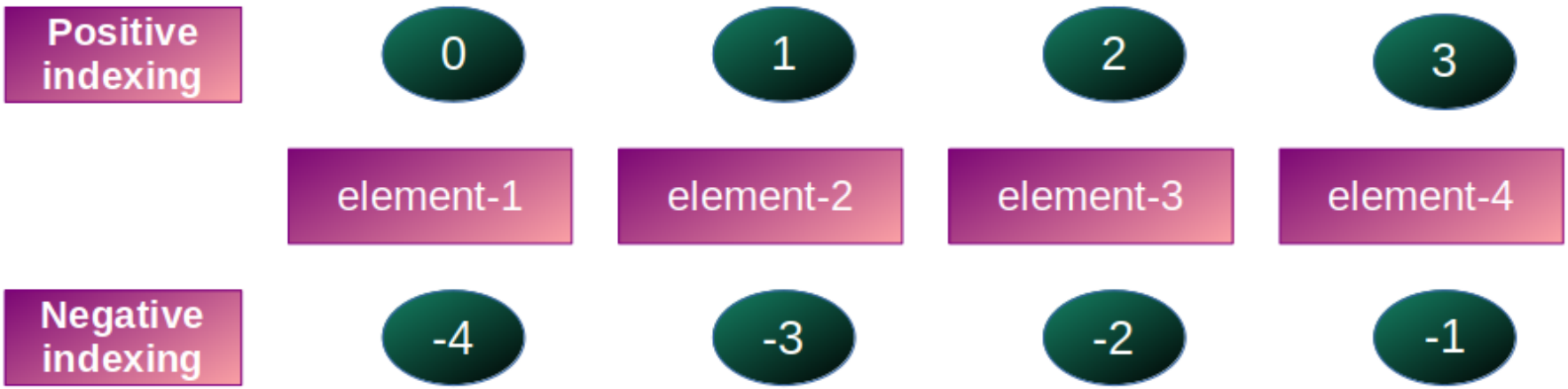
```
[100, 'names', 300.2052]
[<class 'int'>, <class 'str'>, <class 'float'>]
<class 'list'>
```

In [5]:
```
# non-unique
# a list can contain duplicated elements
dup_list = ['jeff', 'jeff', 'jeff', 'jeffrey']
print(dup_list)
```

```
['jeff', 'jeff', 'jeff', 'jeffrey']
```

List indexing

- Every element in a list has an associated index. The index is like an address and is used to access the element.
- Positive indexing starts at 0 and continues to $n-1$ for the last element.
- Negative indexing starts at -1 and continues to $-n$ for the first element.



Add elements to a list

New elements to an existing list using the **append** or the **insert** method.

In [8]:
```
# list of names
names = ['jeff', 'tom', 'grace', 'joyce', 'tan', 'yi']
print(names)

# append(): add an element to the end of list
names.append('vincent')
print(names)
```

```
['jeff', 'tom', 'grace', 'joyce', 'tan', 'yi']
['jeff', 'tom', 'grace', 'joyce', 'tan', 'yi', 'vincent']
```

In [9]:
```
# list of names
names = ['jeff', 'tom', 'grace', 'joyce', 'tan', 'yi']
print(names)

# insert element to a specific position
# insert with positive-indexing
names.insert(3, 'kevin')
print(names)

# insert with negative-indexing
names.insert(-2, 'megan')
print(names)
```

```
['jeff', 'tom', 'grace', 'joyce', 'tan', 'yi']
['jeff', 'tom', 'grace', 'kevin', 'joyce', 'tan', 'yi']
['jeff', 'tom', 'grace', 'kevin', 'joyce', 'megan', 'tan', 'yi']
```

Add elements to a list

A new list can be added to an existing list using the **extend** method or the +-operator.

In [10]:
```
# combines two lists with the list method extend
fruit = ['durian','papaya','banana']
names.extend(fruit)
print(names)
```

```
['jeff', 'tom', 'grace', 'kevin', 'joyce', 'megan', 'tan', 'yi', 'durian', 'papaya', 'banana']
```

In [11]:
```
# combined two list with the + operator
language = ['english', 'mandarin', 'japanese']

merged_list = fruit + language
print(merged_list)
```

```
['durian', 'papaya', 'banana', 'english', 'mandarin', 'japanese']
```

Deletes elements from a list

Elements from an existing list can be removed using the list method:

- **clear()**: Removes every element from the list and creates an empty list.
- **pop()**: Removes element using element's index.
- **del()**: Removes element using element's name.
- **remove()**: Removes element using element's name.

```
In [12]: language = ['english', 'mandarin', 'japanese']
         print(language)

         # clear() remove everything from the list
         language.clear()
         print(language)
```

```
['english', 'mandarin', 'japanese']
[]
```

Deletes elements from a list

pop(): Removes element using element's index. When the element's index is not specified, **pop()** removes the last element by default.

```
In [13]: # pop removes element using index
         fruit = ['durian','papaya','banana', 'orange', 'apples']
         print(f'Before using pop: {fruit}')

         # remove the last element in the list by default
         fruit.pop()
         print(f'After using pop without indexing: {fruit}')

         # remove using index (positive-indexing)
         fruit.pop(0)
         print(f'After using pop with indexing: {fruit}')

Before using pop: ['durian', 'papaya', 'banana', 'orange', 'apples']
After using pop without indexing: ['durian', 'papaya', 'banana', 'orange']
After using pop with indexing: ['papaya', 'banana', 'orange']
```

```
In [15]: # pop cannot work with element's name
         fruit.pop('banana')

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [15], in <cell line: 2>()
      1 # pop cannot work with element's name
----> 2 fruit.pop('banana')

TypeError: 'str' object cannot be interpreted as an integer
```

Deletes elements from a list

del(): Removes element using element's index.

```
In [16]: # remove using index (negative-indexing)
         del fruit[-1]
         print(f'After removing with del method: {fruit}')

         After removing with del method: ['papaya', 'banana']

In [17]: # del cannot work with element's name
         del fruit['papaya']

         ---------------------------------------------------------------------------
         TypeError                                 Traceback (most recent call last)
         Input In [17], in <cell line: 2>()
               1 # del cannot work with element's name
         ----> 2 del fruit['papaya']

         TypeError: list indices must be integers or slices, not str
```

remove(): Removes element using element's name.

```
In [18]: # remove using element's name
         animal = ['cat', 'dog', 'monkey', 'bird']
         print(f'Before removing: {animal}')
         animal.remove('cat')
         print(f'Remove by using element name: {animal}')

         Before removing: ['cat', 'dog', 'monkey', 'bird']
         Remove by using element name: ['dog', 'monkey', 'bird']

In [19]: # try to remove using index
         animal.remove(0)

         ---------------------------------------------------------------------------
         ValueError                                Traceback (most recent call last)
         Input In [19], in <cell line: 2>()
               1 # try to remove using index
         ----> 2 animal.remove(0)

         ValueError: list.remove(x): x not in list
```

List slicing

Creates a subset of list by using the start-index and stop-index. The basic syntax is **name-of-list[start-index: stop-index: index-jump].**
Note: list slicing begins from the starts-index through the stop-index, but DOES NOT include the stop-index.

In [26]:
```
# list of names
names = ['jeff', 'tom', 'grace', 'joyce', 'tan', 'yi']

# positive list slicing
# default index-jump is 1
new_names = names[0:2]
print(new_names)
```

```
['jeff', 'tom']
```

In [27]:
```
# negative list slicing
new_names = names[-3:-1]
print(new_names)
```

```
['joyce', 'tan']
```

In [28]:
```
# change the default index-jump to 2
new_names = names[0:4:2]
print(new_names)
```

```
['jeff', 'grace']
```

In [29]:
```
# copy the whole list
all_names = names[:]
print(all_names)
```

```
['jeff', 'tom', 'grace', 'joyce', 'tan', 'yi']
```

List sorting

A list can be sorted using:

- **sort()**: sort a list in-place;
- **sorted()**: sort the list and returns a new list.

```
In [23]: nums = [100,-50, 0, 20, 2]
         print(nums)

         nums.sort()
         print(f'After sorting: {nums}')
```

```
[100, -50, 0, 20, 2]
After sorting: [-50, 0, 2, 20, 100]
```

```
In [24]: # create a new variable after sorting
         # Incorrect: use sort() to store the stored list to a new list
         wrong_sorted_num = nums.sort()
         print(wrong_sorted_num)

         # correct: use sorted
         correct_sorted_num = sorted(nums)
         print(correct_sorted_num)

         # reverse sorting by using the keyword reverse
         reverse_sorted = sorted(nums, reverse=True)
         print(reverse_sorted)
```

```
None
[-50, 0, 2, 20, 100]
[100, 20, 2, 0, -50]
```

Review list understanding

Exercise-3

- Create a list of top-10 universities in the world according to QS World University Rankings in 2022;
- Add universities rank-11 to rank-15 to the list you have created;
- Remove the last 10 universities from the existing list;
- Use list slicing to select the top-3 universities only.

Link:
https://www.topuniversities.com/university-rankings/world-university-rankings/2022

Tuples characteristics

A tuple stores data as comma-separated elements using parentheses - (element 1, element 2, element 3). It is an ordered, sequential, heterogeneous, indexed, **immutable** and non-unique data structure.

In [1]:
```
# create a tuple using parentheses ()
fruit = ('banana','papaya','durian', 'orange')
print(fruit)

print(type(fruit))
```

```
('banana', 'papaya', 'durian', 'orange')
<class 'tuple'>
```

In [2]:
```
# for a single element,
# a comma is required to make the element a tuple.
is_tuple = (10,)
print(type(is_tuple))

# without the comma, the element with only the
# parentheses does not make a tuple.
not_a_tuple = (10)
print(type(not_a_tuple))
```

```
<class 'tuple'>
<class 'int'>
```

In [3]:
```
# creates a tuple with the built-in tuple function
# converts a string into a tuple of characters
tuple_of_strings = tuple('pokemon')
tuple_of_strings
```

Out[3]:
```
('p', 'o', 'k', 'e', 'm', 'o', 'n')
```

Tuples characteristics

Like the data structure list, a tuple can be heterogeneous and non-unique (*i.e.* contains duplicated elements).

In [4]:
```
# A tuple can be empty without element
empty_tuple = ()
print(empty_tuple)
```

```
()
```

In [5]:
```
# heterogenenous tuple
mixed_tuple = (100, 'names', 300.2052)
mixed_type = (type(100), type('names'), type(300.2052))

print(mixed_tuple)
print(mixed_type)

# same as the creating the list,
# when creating a tuple consists of different datatypes,
# the final datatype becomes a tuple.
print(type(mixed_type))
```

```
(100, 'names', 300.2052)
(<class 'int'>, <class 'str'>, <class 'float'>)
<class 'tuple'>
```

In [6]:
```
# non-unique
# a tuple can alsp contain duplicated elements
dup_tuple = ('jeff', 'jeff', 'jeff', 'jeffrey')
print(dup_tuple)
```

```
('jeff', 'jeff', 'jeff', 'jeffrey')
```

Tuples indexing and slicing

A tuple can also be indexed using positive and negative indexing, and sliced in the similar manner to list.

Tuples indexing

In [7]:
```
names = ('jeff', 'tom', 'grace', 'joyce', 'tan', 'yi')
print(names)

# positive-index
print(names[0])

# negative-index
print(names[-1])
```

```
('jeff', 'tom', 'grace', 'joyce', 'tan', 'yi')
jeff
yi
```

Tuples slicing

In [8]:
```
print(names[0:4:2])
```

```
('jeff', 'grace')
```

Tuples are immutables

What makes a tuple different from a list is that a tuple is **immutable**. We cannot change the tuple **after** creating the tuple (*i.e.* no addition, removal or replacement).

```
In [9]: names.append('vincent')
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [9], in <cell line: 1>()
----> 1 names.append('vincent')

AttributeError: 'tuple' object has no attribute 'append'
```

```
In [10]: names[-1] = 'peter'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [10], in <cell line: 1>()
----> 1 names[-1] = 'peter'

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

Tuples are useful when we want to create fixed or constant elements in our scripts.

Dictionary characteristics

A dictionary stores data in the form of **key-value** pairs using curly brackets - {key-1: value-1, key-2: value-2, key-3: value-3}. It is an ordered, sequential, heterogeneous, indexed, **mutable with unique key** (*i.e.* keys with the same name can only exist once).

In [1]:
```
# Create a dictionary using curly-bracket with key-value pairs
student_email = {'joshua': 'joshua_tan@gmail.com',
                 'megan': 'megan_dixon@hotmail.com',
                 'vincent': 'vincent_1234@yahoo.com'}

print(student_email)
```

```
{'joshua': 'joshua_tan@gmail.com', 'megan': 'megan_dixon@hotmail.com', 'vincent': 'vincent_1234@yahoo.com'}
```

In [2]:
```
# creates a dictionary using the dict method
# using the dict method on a list of tuples for key-value pair
student_email = dict([('joshua','joshua_tan@gmail.com'),
                      ('megan','megan_dixon@hotmail.com'),
                      ('vincent', 'vincent_1234@yahoo.com')])
print(student_email)
```

```
{'joshua': 'joshua_tan@gmail.com', 'megan': 'megan_dixon@hotmail.com', 'vincent': 'vincent_1234@yahoo.com'}
```

In [3]:
```
# Dictionary are heterogeneous,
# the keys and corresponding values can have different datatype
student_id = {'joshua': 1000,
              'megan': 1001,
              'vincent': 1002}
print(student_id)

# unique key: key with the same name but have different value
# can only exist once
new_id = {'megan': 1005}
student_id.update(new_id)
print(student_id)
```

```
{'joshua': 1000, 'megan': 1001, 'vincent': 1002}
{'joshua': 1000, 'megan': 1005, 'vincent': 1002}
```

Dictionary methods

- **dict.keys()**: To extract all keys of a dict
- **dict.values()**: To extract all values of a dict
- **dict.items()**: To extract all keys and their corresponding values of a dict.

In [4]:
```
# extract all keys of a dictionary
print(student_email.keys())
print('*'*50)

# extract all values
print(student_email.values())
print('*'*50)

# access all keys and their corresponding value
print(student_email.items())
print('*'*50)
```

```
dict_keys(['joshua', 'megan', 'vincent'])
**************************************************
dict_values(['joshua_tan@gmail.com', 'megan_dixon@hotmail.com', 'vincent_1234@yahoo.com'])
**************************************************
dict_items([('joshua', 'joshua_tan@gmail.com'), ('megan', 'megan_dixon@hotmail.com'), ('vincent', 'vincent_1234@yaho
o.com')])
**************************************************
```

Dictionary methods

- **dict[key]**: To extract a specific key;
- **dict.get(key)**: To extract a specific key.

```
In [5]: # extract a specific value using the key
        print(student_email['joshua'])

        # extract a specific value of a key using the get method
        print(student_email.get('joshua'))
```

```
joshua_tan@gmail.com
joshua_tan@gmail.com
```

- **dict.update({new-key: new-value})**: To add new entry into an existing dict. The orders of the dict do not change with each new insertion (from Python 3.6 onwards).

```
In [6]: # creates a new dicctionary using curly bracket
        # use the update method to add new entry
        new_entry = {'benson':'benson@gmail.com'}
        student_email.update(new_entry)
        print(student_email)

        # update by creating a list of new tuple
        student_email.update([('winnie', 'winnie_pooh@yahoo.com')])
        print(student_email)
```

```
{'joshua': 'joshua_tan@gmail.com', 'megan': 'megan_dixon@hotmail.com', 'vincent': 'vincent_1234@yahoo.com', 'benson
': 'benson@gmail.com'}
{'joshua': 'joshua_tan@gmail.com', 'megan': 'megan_dixon@hotmail.com', 'vincent': 'vincent_1234@yahoo.com', 'benson
': 'benson@gmail.com', 'winnie': 'winnie_pooh@yahoo.com'}
```

Dictionary methods

- **dict.popitem()**: Remove the last key-value pair from the dict;
- **dict.pop(key)**: To remove a specific key from the dict.
- **del dict[key]**: To remove a specific key from the dict.

In [7]:
```
# Remove the last item
student_email.popitem()
print(student_email)
```

```
{'joshua': 'joshua_tan@gmail.com', 'megan': 'megan_dixon@hotmail.com', 'vincent': 'vincent_1234@yahoo.com', 'benson
': 'benson@gmail.com'}
```

In [8]:
```
# Removes by using dict's key using the pop method
student_email.pop('vincent')
print(student_email)

# Removes by using the del method
del student_email['joshua']
print(student_email)
```

```
{'joshua': 'joshua_tan@gmail.com', 'megan': 'megan_dixon@hotmail.com', 'benson': 'benson@gmail.com'}
{'megan': 'megan_dixon@hotmail.com', 'benson': 'benson@gmail.com'}
```

Dictionary methods

- **dict[key] = new-value**

In [9]:
```
# We want to change megan's email address
student_email = dict([('joshua','joshua_tan@gmail.com'),
                      ('megan','megan_dixon@hotmail.com'),
                      ('vincent', 'vincent_1234@yahoo.com')])
print(student_email)

# access the key and change the corresponding value
student_email['megan'] = 'megan_new_email@twitter.com'
print(student_email)
```

```
{'joshua': 'joshua_tan@gmail.com', 'megan': 'megan_dixon@hotmail.com', 'vincent': 'vincent_1234@yahoo.com'}
{'joshua': 'joshua_tan@gmail.com', 'megan': 'megan_new_email@twitter.com', 'vincent': 'vincent_1234@yahoo.com'}
```

Review dictionary understanding

Exercise-4

- By using the provided TikTok's dataset (courtesy of Kaggle), create a dictionary of five randomly chosen artist's name with their track names.
- Copy the created dictionary and replace the track's name with the corresponding album's name.

Set characteristics

A set stores comma-separated data using curly brackets - {element-1, element-2, element-3}. It is an **unordered** (the order of existing elements in the set may change with each new insertion), mutable, heterogeneous, **unindexed** (no index for each set element) and **unique** (duplicated elements will be ignored).

In [1]:
```
# creates a set with curly bracket
set_of_nums = {10, 20, 30}
print(set_of_nums)

# type int converted to type set
print(type(set_of_nums))
print('*'*50)

# set of names
names = {'jeff', 'tom', 'grace', 'joyce', 'tan', 'yi'}
print(names)

# type string converted to type set
print(type(names))
print('*'*50)

# heterogeneous set is possible
mixed_set = {100, -200.5, 'jeff'}
print(mixed_set)

mixed_type = {type(100), type(-200.5), type('jeff')}
print(mixed_type)
print(type(mixed_set))
```

```
{10, 20, 30}
<class 'set'>
**************************************************
{'tom', 'jeff', 'yi', 'grace', 'joyce', 'tan'}
<class 'set'>
**************************************************
{-200.5, 100, 'jeff'}
{<class 'int'>, <class 'float'>, <class 'str'>}
<class 'set'>
```

Add and delete set elements

New elements can be added using the **add()** and deleted using the **remove()** or **discard()** methods.

In [2]:
```
# Add new participant to the set of names
names.add('vincent')
print(names)

new_names = {'peter', 'winnie', 'william'}
names.update(new_names)
print(names)
```

```
{'vincent', 'tom', 'jeff', 'joyce', 'yi', 'tan', 'grace'}
{'peter', 'winnie', 'yi', 'vincent', 'grace', 'william', 'tom', 'jeff', 'joyce', 'tan'}
```

In [3]:
```
# Try adding a duplicated name
names.add('jeff')
print(names)

dup_names = {'jeff', 'jeff', 'jeff', 'jeffrey'}
print(dup_names)
```

```
{'peter', 'winnie', 'yi', 'vincent', 'grace', 'william', 'tom', 'jeff', 'joyce', 'tan'}
{'jeffrey', 'jeff'}
```

In [4]:
```
# Compare to list
# a list can contain duplicated elements
dup_list = ['jeff', 'jeff', 'jeff', 'jeffrey']
print(dup_list)
```

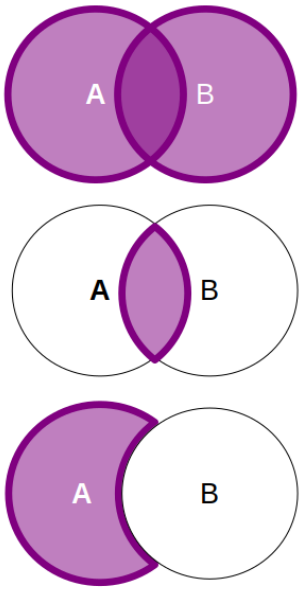
```
['jeff', 'jeff', 'jeff', 'jeffrey']
```

In [5]:
```
names.remove('yi')
print(names)

names.discard('vincent')
print(names)
```

```
{'peter', 'winnie', 'vincent', 'grace', 'william', 'tom', 'jeff', 'joyce', 'tan'}
{'peter', 'winnie', 'grace', 'william', 'tom', 'jeff', 'joyce', 'tan'}
```

Commonly used set methods




```
In [6]: first_basket = {'banana', 'apple', 'durian'}
        second_basket = {'papaya', 'mango', 'watermelon', 'banana', 'apple'}

        # merge two sets into a new set
        print(first_basket.union(second_basket))

        # union doesn't change the original set
        print(first_basket)
```

```
{'papaya', 'banana', 'watermelon', 'apple', 'mango', 'durian'}
{'durian', 'banana', 'apple'}
```

```
In [7]: # extract the common elements from both sets
        print(first_basket.intersection(second_basket))
```

```
{'banana', 'apple'}
```

```
In [8]: # returns element only in the first set
        print(first_basket.difference(second_basket))

        # returns elements only in the second set
        print(second_basket.difference(first_basket))
```

```
{'durian'}
{'papaya', 'watermelon', 'mango'}
```

Summary data structures

Characteristics	List	Tuple	Dict	Set
Ordered	X	X	X*	
Sequential	X	X	X	X
Heterogeneous	X	X	X	X
Indexed	X	X	X	
Mutable	X		X	X
Unique			X**	X

Table: Summary of data structures in Python. Coloured X indicates affirmative characteristics.

* Ordered from Python 3.6.2 onwards.

** Unique keys in dict's key-value pairs.

Lesson references and further reading

- W^3 Schools.
- Python Basics You Need to Know, Always: Data Structures. Shubhangi Hora.
- Python 101. Michael Driscoll.
- PEP 8 – Style Guide for Python Code.

Thank you!

Constructive feedback or questions, please drop me an email.