Python

Fabrice BOISSIER

EPITA - Apprentissage

12 septembre 2023

Version 1

License

- (LaTeX source code partially reused from Gabriel Laskar)
- Copyright © 2022-2024, Fabrice Boissier

Part I

Introduction

1: Introduction

1 Course outline

What are we trying to learn?

Python language

What are the objectives of the course:

- Discovering Python
- Writing small Python's scripts
- Understanding main Python's concepts
- Writing functions and programs in Python
- Being autonomous while writing code

Course organization

- Lecture (30 mins $\sim$ 1h per day)

Course organization

- Lecture (30 mins $\sim$ 1h per day)
- Labs (full day)

Course organization

- Lecture (30 mins $\sim$ 1h per day)
- Labs (full day)
- Submission preparation (15/09/2023 morning)

Course organization

- Lecture (30 mins $\sim$ 1h per day)
- Labs (full day)
- Submission preparation (15/09/2023 morning)
- Exercices submission (XX/09/2023)

tuesday 12th september 2023
wednesday 13th september 2023
friday 15th september 2023 - *[morning only]*

Course organization

- Lecture (30 mins ~ 1h per day)
- Labs (full day)
- Submission preparation (15/09/2023 morning)
- Exercices submission (XX/09/2023)

tuesday 12th september 2023
wednesday 13th september 2023
friday 15th september 2023 - *[morning only]*

Evaluation

Evaluation will only be on submitted exercices

Part II

Python Basics

2: Python Basics

1 Python Language

Discovering Python

- Interpreter

Discovering Python

- Interpreter
  `/usr/bin/python3`

Discovering Python

- Interpreter
  `/usr/bin/python3`
- Imperative programming language

Discovering Python

- Interpreter
  `/usr/bin/python3`
- Imperative programming language
- Object Oriented Programming language

Discovering Python

- Interpreter
  `/usr/bin/python3`
- Imperative programming language
- Object Oriented Programming language
- Multiple versions (currently major version 3)

Discovering Python

- Interpreter
  `/usr/bin/python3`
- Imperative programming language
- Object Oriented Programming language
- Multiple versions (currently major version 3)
  Beware, version 2.7 is still running on some machines

Calling Python

Simply like any program

```
$
```

Calling Python

Simply like any program

```
$ python3
```

Calling Python

Simply like any program

```
$ python3
Python 3.10.4 (main, Apr  2 2022, 09:04:19) [
   GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license
   " for more information.
>>>
```

Calling Python

Quit by typing "**`exit()`**" or by pressing **`Ctrl + D`**

```
$ python3
Python 3.10.4 (main, Apr  2 2022, 09:04:19) [
   GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license
   " for more information.
>>> exit()
```

Calling Python

You can call a script by calling python with an argument

```
$ python3 my_script.py
```

Calling Python

You can call a script by calling python with an argument

```
$ python3 my_script.py
Hello World!
```

Versions

Beware of versions !
We currently use the major version 3 (3.8, 3.10, ...)

```
$ python3
```

Versions

Beware of versions !
We currently use the major version 3 (3.8, 3.10, ...)

```
$ python3
Python 3.10.4 (main, Apr  2 2022, 09:04:19) [
   GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license
   " for more information.
>>> exit()
```

Versions

Beware of versions !
We currently use the major version 3 (3.8, 3.10, ...)

```
$ python3
Python 3.10.4 (main, Apr  2 2022, 09:04:19) [
   GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license
   " for more information.
>>> exit()
```

Do not use the obsolete 2.7 (**`python2`**, **`python27`**, **`python2.7`**)

Versions

```
$ python
Python 3.10.4 (main, Apr  2 2022, 09:04:19)

$ python3
Python 3.10.4 (main, Apr  2 2022, 09:04:19)

$ python3.10
Python 3.10.4 (main, Apr  2 2022, 09:04:19)
```

Versions

```
$ python
Python 3.10.4 (main, Apr  2 2022, 09:04:19)

$ python3
Python 3.10.4 (main, Apr  2 2022, 09:04:19)

$ python3.10
Python 3.10.4 (main, Apr  2 2022, 09:04:19)

$ python2
Python 2.7.18 (default, Jan  2 2021, 09:22:32)
```

Versions

Please, check right now which default version of Python you have when you type **`python`**

Eventually, install package **`python-is-python3`** in order to avoid any problem

Part III

Python Quick Scripting

3: Python Quick Scripting

1 First scripts

2 Quick overview of types

3 Overview of syntax and control flow

Hello World!

```
#! /usr/bin/python

print("Hello World!")
```

Hello World!

```
#! /usr/bin/python

print("Hello World!")
```

L1: The interpreter can be ajdusted

Hello World!

```
#! /usr/bin/python

print("Hello World!")
```

L1: The interpreter can be ajdusted
L3: **print** takes a string as a parameter

Hello World!

```
#! /usr/bin/python

print("Hello World!")
```

L1: The interpreter can be ajdusted
L3: **print** takes a string as a parameter
L3: Strings are enclosed within quotes (') or double quotes (")

Concatenation & Printing integers

```
#! /usr/bin/python

var = 42
print("Hello World! " + str(var))
```

Concatenation & Printing integers

```
#! /usr/bin/python

var = 42
print("Hello World! " + str(var))
```

Semicolons (`;`) at the end of statements are optional (rarely used)

Concatenation & Printing integers

```
#! /usr/bin/python

var = 42
print("Hello World! " + str(var))
```

Semicolons (;) at the end of statements are optional (rarely used)
L3: Declare variables and assign value directly (type not needed)

Concatenation & Printing integers

```
#! /usr/bin/python

var = 42
print("Hello World! " + str(var))
```

Semicolons (;) at the end of statements are optional (rarely used)
L3: Declare variables and assign value directly (type not needed)
L4: String concatenation with **+**

Concatenation & Printing integers

```
#! /usr/bin/python

var = 42
print("Hello World! " + str(var))
```

Semicolons (`;`) at the end of statements are optional (rarely used)
L3: Declare variables and assign value directly (type not needed)
L4: String concatenation with **`+`**
L4: Conversion from integer or float to string with **`str()`** function

Read arguments

```
#! /usr/bin/python

import sys
print("Hello World! " + sys.argv[0])
```

args.py

Read arguments

```python
#! /usr/bin/python

import sys
print("Hello World! " + sys.argv[0])
```

args.py

```
$ python args.py
```

Read arguments

```
#! /usr/bin/python

import sys
print("Hello World! " + sys.argv[0])
```

args.py

```
$ python args.py
Hello World! args.py
```

Read arguments

```
#! /usr/bin/python

import sys
print("Hello World! " + sys.argv[0])
```

Read arguments

```
#! /usr/bin/python

import sys
print("Hello World! " + sys.argv[0])
```

L3: Includes an external module (**import module**)

Read arguments

```
#! /usr/bin/python

import sys
print("Hello World! " + sys.argv[0])
```

L3: Includes an external module (**`import module`**)
L3: Access to arguments is made through **`sys`** module
L4: Access to an attribute is made with a dot (`.`)

Read arguments

```
#! /usr/bin/python

import sys
print("Hello World! " + sys.argv[0])
```

L3: Includes an external module (**`import module`**)
L3: Access to arguments is made through **`sys`** module
L4: Access to an attribute is made with a dot (`.`)
L4: **`argv`** is an array (like in C and others)

Read arguments

```python
#! /usr/bin/python

import sys
print("Hello World! " + sys.argv[0])
```

L3: Includes an external module (**`import module`**)
L3: Access to arguments is made through **`sys`** module
L4: Access to an attribute is made with a dot (`.`)
L4: **`argv`** is an array (like in C and others)
L4: Arrays begin at index **0**

3: Python Quick Scripting

1. First scripts
2. Quick overview of types
3. Overview of syntax and control flow

Show types/Quick debug

```python
#! /usr/bin/python
type(42)
type("Hello World!")
```

types1.py

```
$ python types1.py
```

Show types/Quick debug

```
#! /usr/bin/python
type(42)
type("Hello World!")
```

types1.py

```
$ python types1.py
<class 'int'>
<class 'str'>
```

Show types/Quick debug

```
#! /usr/bin/python
type(42)
type("Hello World!")
```

types1.py

```
$ python types1.py
<class 'int'>
<class 'str'>
```

Function **`type()`** writes the type of the parameter

Show types/Quick debug

```
#! /usr/bin/python
import sys
type(sys)
type(print("lol"))
```

types2.py

```
$ python types2.py
```

Show types/Quick debug

```python
#! /usr/bin/python
import sys
type(sys)
type(print("lol"))
```

types2.py

```
$ python types2.py
<class 'module'>
lol
<class 'NoneType'>
```

3: Python Quick Scripting

1 First scripts

2 Quick overview of types

3 Overview of syntax and control flow

Functions overview

```python
print("Hello World!")
```

functions1.py

Functions overview

```python
def MyFunction():
  print("Hello World!")
  return (0)
```

functions1.py

Functions overview

```python
def MyFunction():
  print("Hello World!")
  return (0)

MyFunction()
```

functions1.py

Functions overview

```python
def MyFunction():
  print("Hello World!")
  return (0)

MyFunction()
```

functions1.py

```
$ python functions1.py
```

Functions overview

```python
def MyFunction():
  print("Hello World!")
  return (0)

MyFunction()
```

functions1.py

```
$ python functions1.py
Hello World!
```

Functions overview

```
def MyFunction():
  print("Hello World!")
  return (0)

MyFunction()
```

Functions overview

```
def MyFunction():
  print("Hello World!")
  return (0)

MyFunction()
```

L1: Functions begin by a **`def`** and are followed by their parameters

Functions overview

```
def MyFunction():
  print("Hello World!")
  return (0)

MyFunction()
```

L1: Functions begin by a **`def`** and are followed by their parameters
L1: Definition of functions are terminated by a semicolon (`:`)

Functions overview

```python
def MyFunction():
  print("Hello World!")
  return (0)

MyFunction()
```

L1: Functions begin by a **`def`** and are followed by their parameters
L1: Definition of functions are terminated by a semicolon (**:**)
L2: Indentation defines in which scope the line is

Functions overview

```
def MyFunction():
  print("Hello World!")
  return (0)

MyFunction()
```

L1: Functions begin by a **`def`** and are followed by their parameters
L1: Definition of functions are terminated by a semicolon (**`:`**)
L2: Indentation defines in which scope the line is
L2: **Indentation is VERY IMPORTANT!**

Functions overview

```
def MyFunction():
  print("Hello World!")
  return (0)

MyFunction()
```

L1: Functions begin by a **def** and are followed by their parameters
L1: Definition of functions are terminated by a semicolon (**:**)
L2: Indentation defines in which scope the line is
L2: **Indentation is VERY IMPORTANT!**
L3 & L5: Returning and calling are similar to C

Conditions overview

`if` … `elif` … `else` …

```
def MyOtherFunction():
  var = 42
  if (var < 42):
    print("Oh no...")
  elif (var > 42):
    print("WT...")
  else:
    print("OK")
```

Conditions overview

if ... **elif** ... **else** ...

```
def MyOtherFunction():
  var = 42
  if (var < 42):
    print("Oh no...")
  elif (var > 42):
    print("WT...")
  else:
    print("OK")
```

L3 & L5 & L6: Conditions are followed by a semicolon (:)

Conditions overview

match ... **case** ... *(only since Python 3.10)*

```
var = "Hello World!"
match var:
  case ['Hello']:
    print("Beginning")
  case ['World!' | 'Hello World!']:
    print("End")
  case _:
    print("In any other cases")
```

Conditions overview

match ... **case** ... *(only since Python 3.10)*

```
var = "Hello World!"
match var:
  case ['Hello']:
    print("Beginning")
  case ['World!' | 'Hello World!']:
    print("End")
  case _:
    print("In any other cases")
```

Similar to *switch-case* in C, without needing *return* or *break*

Conditions overview

match ... **case** ... *(only since Python 3.10)*

```
var = "Hello World!"
match var:
  case ['Hello']:
    print("Beginning")
  case ['World!' | 'Hello World!']:
    print("End")
  case _:
    print("In any other cases")
```

Similar to *switch-case* in C, without needing *return* or *break*
More complex than that (check the documentation after the course)

Loops overview

while

```
var = 0
while var < 42:
  print("Hi Nations!")
  var += 1
```

Loops overview

while

```
var = 0
while var < 42:
  print("Hi Nations!")
  var += 1
```

L2: Regular **while** loop
L4: **+=** operator acts like **var = var + 1**

Loops overview

while

```
var = 0
while var < 42:
  print("Hi Nations!")
  var += 1
```

L2: Regular **while** loop
L4: **+=** operator acts like **var = var + 1**
Don't forget indentation

Loops overview

for (1)

```
for var in range(0, 10):
  print("Hi Nations!")

my_text = "Yo Countries!"
for char in my_text:
  print(char)
```

Loops overview

for (1)

```
for var in range(0, 10):
  print("Hi Nations!")

my_text = "Yo Countries!"
for char in my_text:
  print(char)
```

L1: **range** calculates values from **0** to **10**

Loops overview

for (1)

```
for var in range(0, 10):
  print("Hi Nations!")

my_text = "Yo Countries!"
for char in my_text:
  print(char)
```

L1: **range** calculates values from **0** to **10**
L1: **in** iterates through each value of a list
L1: Each value will be put into the variable before **in**

Loops overview

for (1)

```
for var in range(0, 10):
  print("Hi Nations!")

my_text = "Yo Countries!"
for char in my_text:
  print(char)
```

L1: **range** calculates values from **0** to **10**
L1: **in** iterates through each value of a list
L1: Each value will be put into the variable before **in**
L4 & L5: Strings are considered as characters lists

Loops overview

for (2)

```
my_list = [ 5, 2, 3, 1, 4 ]
for var in range(len(my_list)):
  if (var < 2):
    break
  else:
    print("Hi Nations!")
```

Loops overview

for (2)

```
my_list = [ 5, 2, 3, 1, 4 ]
for var in range(len(my_list)):
  if (var < 2):
    break
  else:
    print("Hi Nations!")
```

L1: Declaration of a list

Loops overview

for (2)

```
my_list = [ 5, 2, 3, 1, 4 ]
for var in range(len(my_list)):
  if (var < 2):
    break
  else:
    print("Hi Nations!")
```

L1: Declaration of a list
L1: Never put a dash (−) in variables name (use an underscore _)

Loops overview

for (2)

```
my_list = [ 5, 2, 3, 1, 4 ]
for var in range(len(my_list)):
  if (var < 2):
    break
  else:
    print("Hi Nations!")
```

L1: Declaration of a list
L1: Never put a dash (**-**) in variables name (use an underscore _)
L2: **len** gets the length of a list

Loops overview

for (2)

```
my_list = [ 5, 2, 3, 1, 4 ]
for var in range(len(my_list)):
  if (var < 2):
    break
  else:
    print("Hi Nations!")
```

L1: Declaration of a list
L1: Never put a dash (**-**) in variables name (use an underscore _)
L2: **len** gets the length of a list
L4: **break** ends the loop

Exceptions overview

try … **except** …

```
var = 3
try:
  var = 42 / var
except Exception as exc:
  print("Error: " + str(exc))
```

Exceptions overview

try ... **except** ...

```
var = 3
try:
  var = 42 / var
except Exception as exc:
  print("Error: " + str(exc))
```

L2: Begin the block of code to check with a **try**

Exceptions overview

try ... **except** ...

```
var = 3
try:
  var = 42 / var
except Exception as exc:
  print("Error: " + str(exc))
```

L2: Begin the block of code to check with a **try**

L4: Catch exceptions with a **except**

Exceptions overview

try … **except** …

```
var = 3
try:
  var = 42 / var
except Exception as exc:
  print("Error: " + str(exc))
```

L2: Begin the block of code to check with a **try**
L4: Catch exceptions with a **except**
L4: Catch any exception with **Exception** and put it in **exc**

Exceptions overview

try ... **except** ...

```
var = 3
try:
  var = 42 / var
except Exception as exc:
  print("Error: " + str(exc))
```

L2: Begin the block of code to check with a **try**
L4: Catch exceptions with a **except**
L4: Catch any exception with **Exception** and put it in **exc**
L4 & L5: Write the actions to take if an exception occurs

Exceptions overview

try … **except** …

```
var = 3
try:
  var = 42 / var
except Exception as exc:
  print("Error: " + str(exc))
```

L2: Begin the block of code to check with a **try**

L4: Catch exceptions with a **except**

L4: Catch any exception with **Exception** and put it in **exc**

L4 & L5: Write the actions to take if an exception occurs

L5: Don't forget to convert to a string with **str()**

Exceptions overview

try ... **except** ... **else** ... **finally**

```
try:
  var = 42 / 3
except ZeroDivisionError:
  print("There was a division by zero")
else:
  print("It worked, result: ", var)
finally:
  print("--After everything--")
```

Exceptions overview

try ... **except** ... **else** ... **finally**

```
try:
  var = 42 / 3
except ZeroDivisionError:
  print("There was a division by zero")
else:
  print("It worked, result: ", var)
finally:
  print("--After everything--")
```

L2: Check for a specific exception (division by zero)

Exceptions overview

`try` … **`except`** … **`else`** … **`finally`**

```python
try:
  var = 42 / 3
except ZeroDivisionError:
  print("There was a division by zero")
else:
  print("It worked, result: ", var)
finally:
  print("--After everything--")
```

L2: Check for a specific exception (division by zero)

L4: If no exception was raised, it executes the **`else`**

Exceptions overview

`try` ... **`except`** ... **`else`** ... **`finally`**

```
try:
  var = 42 / 3
except ZeroDivisionError:
  print("There was a division by zero")
else:
  print("It worked, result: ", var)
finally:
  print("--After everything--")
```

L2: Check for a specific exception (division by zero)
L4: If no exception was raised, it executes the **`else`**
L6: In any case (exception or not), it executes the **`finally`** clause

raise

```
def FragileFunction():
  var = 42
  if (var == 42):
    raise ValueError("Argf")
  else:
    return (0)
```

raise

```
def FragileFunction():
  var = 42
  if (var == 42):
    raise ValueError("Argf")
  else:
    return (0)

def MainFunction():
  print("--Before--")
  try:
    FragileFunction()
  except ValueError as exp:
    print("Exception caught: ", exp)
  print("--After--")
```

Exceptions overview

`raise` and custom exceptions

Exceptions overview

`raise` and custom exceptions

- **`raise`** instruction triggers an exception

Exceptions overview

`raise` and custom exceptions

- **`raise`** instruction triggers an exception
- Use **`ValueError`** in order to customize the message...

Exceptions overview

`raise` and custom exceptions

- **`raise`** instruction triggers an exception
- Use **`ValueError`** in order to customize the message...
- ...or create a class with the name of your exception and **`Exception`** as a parameter

Exceptions overview

`raise` and custom exceptions

- **`raise`** instruction triggers an exception
- Use **`ValueError`** in order to customize the message...
- ...or create a class with the name of your exception and **`Exception`** as a parameter
- When raising a custom exception, the message is optional

Exceptions overview

`raise` and custom exceptions

- **`raise`** instruction triggers an exception
- Use **`ValueError`** in order to customize the message...
- ...or create a class with the name of your exception and **`Exception`** as a parameter
- When raising a custom exception, the message is optional

Exceptions overview

`raise` and custom exceptions

- **`raise`** instruction triggers an exception
- Use **`ValueError`** in order to customize the message...
- ...or create a class with the name of your exception and **`Exception`** as a parameter
- When raising a custom exception, the message is optional

```
class MyCustomError(Exception):
  pass
```

Exceptions overview

`raise` and custom exceptions

- **`raise`** instruction triggers an exception
- Use **`ValueError`** in order to customize the message...
- ...or create a class with the name of your exception and **`Exception`** as a parameter
- When raising a custom exception, the message is optional

```
class MyCustomError(Exception):
  pass

def FragileFunction():
  raise MyCustomError
```

`raise` and custom exceptions

raise and custom exceptions

```
class MyCustomError(Exception):
  pass
```

raise and custom exceptions

```
class MyCustomError(Exception):
  pass

def FragileFunction():
  raise MyCustomError("Argf")
```

raise and custom exceptions

```
class MyCustomError(Exception):
  pass

def FragileFunction():
  raise MyCustomError("Argf")

def MainFunction():
  print("--Before--")
  try:
    FragileFunction()
  except MyCustomError as exp:
    print("Exception caught: ", exp)
  print("--After--")
```

Part IV

Functions, Imports & Modules

4: Functions, Imports & Modules

1. Functions
   - Parameters
   - Scope of variables
   - Return values
2. Modules & Imports

Functions

Usual function definition and call:

Functions

Usual function definition and call:

```
def MyFunction(name):
  print("Hello " + str(name) + "!")
  return (0)
```

Functions

Usual function definition and call:

```
def MyFunction(name):
  print("Hello " + str(name) + "!")
  return (0)

MyFunction("Fabrice")
```

Functions

Usual function definition and call:

```
def MyFunction(name):
  print("Hello " + str(name) + "!")
  return (0)

MyFunction("Fabrice")
MyFunction(42)
```

Functions

Usual function definition and call:

```
def MyFunction(name):
  print("Hello " + str(name) + "!")
  return (0)

MyFunction("Fabrice")
MyFunction(42)
```

MyFunction is the function name

Functions

Usual function definition and call:

```
def MyFunction(name):
  print("Hello " + str(name) + "!")
  return (0)

MyFunction("Fabrice")
MyFunction(42)
```

MyFunction is the function name
name is a parameter (without a type)

Functions

Usual function definition and call:

```
def MyFunction(name):
  print("Hello " + str(name) + "!")
  return (0)

MyFunction("Fabrice")
MyFunction(42)
```

MyFunction is the function name
name is a parameter (without a type)
"Fabrice" and *42* are arguments

4: Functions, Imports & Modules

1 Functions
- Parameters
- Scope of variables
- Return values

2 Modules & Imports

Parameters

- Positional arguments

Parameters

- Positional arguments
  - `def MyFun(param1, param2, param3)`

Parameters

- Positional arguments
  - `def MyFun(param1, param2, param3)`
  - `MyFun("abc", 42, 1337)`

Parameters

- Positional arguments
  - `def MyFun(param1, param2, param3)`
  - `MyFun("abc", 42, 1337)`
  - `MyFun(42, "abc", 1337)`

Parameters

- Positional arguments
  - `def MyFun(param1, param2, param3)`
  - `MyFun("abc", 42, 1337)`
  - MyFun(~~42~~, ~~"abc"~~, 1337)

Parameters

- Positional arguments
  - **`def MyFun(param1, param2, param3)`**
  - **`MyFun("abc", 42, 1337)`**
  - **MyFun(~~42~~, ~~"abc"~~, 1337)**
- Keywords arguments

Parameters

- Positional arguments
  - `def MyFun(param1, param2, param3)`
  - `MyFun("abc", 42, 1337)`
  - `MyFun(`~~`42`~~`, `~~`"abc"`~~`, 1337)`
- Keywords arguments (with default values)
  - `def MyFun(p1="A", p2=1, p3=9)`

Parameters

- Positional arguments
  - **`def MyFun(param1, param2, param3)`**
  - **`MyFun("abc", 42, 1337)`**
  - **`MyFun(`~~`42`~~`, `~~`"abc"`~~`, 1337)`**
- Keywords arguments (with default values)
  - **`def MyFun(p1="A", p2=1, p3=9)`**
  - **`MyFun("abc", 42, 1337)`**

Parameters

- Positional arguments
  - **`def MyFun(param1, param2, param3)`**
  - **`MyFun("abc", 42, 1337)`**
  - **MyFun(~~42~~, ~~"abc"~~, 1337)**
- Keywords arguments (with default values)
  - **`def MyFun(p1="A", p2=1, p3=9)`**
  - **`MyFun("abc", 42, 1337)`**
  - **`MyFun(p1="abc", p2=42, p3=1337)`**

Parameters

- Positional arguments
  - `def MyFun(param1, param2, param3)`
  - `MyFun("abc", 42, 1337)`
  - MyFun(~~42~~, ~~"abc"~~, 1337)
- Keywords arguments (with default values)
  - `def MyFun(p1="A", p2=1, p3=9)`
  - `MyFun("abc", 42, 1337)`
  - `MyFun(p1="abc", p2=42, p3=1337)`
  - `MyFun(p3=1337, p2=42, p1="abc")`

Parameters

- Positional arguments
  - `def MyFun(param1, param2, param3)`
  - `MyFun("abc", 42, 1337)`
  - MyFun(~~42~~, ~~"abc"~~, 1337)
- Keywords arguments (with default values)
  - `def MyFun(p1="A", p2=1, p3=9)`
  - `MyFun("abc", 42, 1337)`
  - `MyFun(p1="abc", p2=42, p3=1337)`
  - `MyFun(p3=1337, p2=42, p1="abc")`
  - `MyFun(p2=42)`

Positional arguments

Positional arguments

```
from datetime import date
def GreetingsPos(Name, Year):
  print("Hi " + Name + "!")
  if (Year <= 0):
    print("(unknown age)")
  else:
    age = date.today().year - Year
    print("(" + str(age) + " years)")
```

Positional arguments

```python
from datetime import date
def GreetingsPos(Name, Year):
  print("Hi " + Name + "!")
  if (Year <= 0):
    print("(unknown age)")
  else:
    age = date.today().year - Year
    print("(" + str(age) + " years)")

GreetingsPos("Fabrice", 1988)
```

Positional arguments

```python
from datetime import date
def GreetingsPos(Name, Year):
  print("Hi " + Name + "!")
  if (Year <= 0):
    print("(unknown age)")
  else:
    age = date.today().year - Year
    print("(" + str(age) + " years)")

GreetingsPos("Fabrice", 1988) # OK
GreetingsPos("Fabrice")
```

Positional arguments

```
from datetime import date
def GreetingsPos(Name, Year):
  print("Hi " + Name + "!")
  if (Year <= 0):
    print("(unknown age)")
  else:
    age = date.today().year - Year
    print("(" + str(age) + " years)")

GreetingsPos("Fabrice", 1988) # OK
GreetingsPos("Fabrice")       # error
```

Positional arguments

```
from datetime import date
def GreetingsPos(Name, Year):
  print("Hi " + Name + "!")
  if (Year <= 0):
    print("(unknown age)")
  else:
    age = date.today().year - Year
    print("(" + str(age) + " years)")

GreetingsPos("Fabrice", 1988) # OK
GreetingsPos("Fabrice")       # error
```

All of the parameters are required if positional arguments

Keywords arguments

Keywords arguments

```
def GreetingsKey(FName="Unknown", BYear=0):
  GreetingsPos(FName, BYear)
```

Keywords arguments

```
def GreetingsKey(FName="Unknown", BYear=0):
  GreetingsPos(FName, BYear)

GreetingsKey("Fabrice", 1988)
```

Keywords arguments

```
def GreetingsKey(FName="Unknown", BYear=0):
  GreetingsPos(FName, BYear)

GreetingsKey("Fabrice", 1988)  # OK
GreetingsKey("Fabrice")
```

Keywords arguments

```python
def GreetingsKey(FName="Unknown", BYear=0):
  GreetingsPos(FName, BYear)

GreetingsKey("Fabrice", 1988)  # OK
GreetingsKey("Fabrice")        # OK
GreetingsKey(BYear=1988, FName="Fabrice")
```

Keywords arguments

```python
def GreetingsKey(FName="Unknown", BYear=0):
  GreetingsPos(FName, BYear)

GreetingsKey("Fabrice", 1988)  # OK
GreetingsKey("Fabrice")        # OK
GreetingsKey(BYear=1988, FName="Fabrice") # OK
GreetingsKey("Fabrice", BYear=1988)
```

Keywords arguments

```python
def GreetingsKey(FName="Unknown", BYear=0):
  GreetingsPos(FName, BYear)

GreetingsKey("Fabrice", 1988)  # OK
GreetingsKey("Fabrice")        # OK
GreetingsKey(BYear=1988, FName="Fabrice") # OK
GreetingsKey("Fabrice", BYear=1988) # OK
```

Keywords arguments

```
def GreetingsKey(FName="Unknown", BYear=0):
  GreetingsPos(FName, BYear)

GreetingsKey("Fabrice", 1988)   # OK
GreetingsKey("Fabrice")         # OK
GreetingsKey(BYear=1988, FName="Fabrice") # OK
GreetingsKey("Fabrice", BYear=1988) # OK
```

Keywords arguments are more flexibles

Keywords arguments

Keywords arguments

```
def GreetingsKey2(FName="", BYear=""):
  GreetingsPos(FName, BYear)
```

Keywords arguments

```
def GreetingsKey2(FName="", BYear=""):
  GreetingsPos(FName, BYear)

GreetingsKey2("Fabrice", 1988)
```

Keywords arguments

```
def GreetingsKey2(FName="", BYear=""):
  GreetingsPos(FName, BYear)

GreetingsKey2("Fabrice", 1988)  # OK
GreetingsKey2(BYear=1988, FName="Fabrice")
```

Keywords arguments

```
def GreetingsKey2(FName="", BYear=""):
  GreetingsPos(FName, BYear)

GreetingsKey2("Fabrice", 1988)  # OK
GreetingsKey2(BYear=1988, FName="Fabrice") # OK
GreetingsKey2(BYear=1988)
```

Keywords arguments

```
def GreetingsKey2(FName="", BYear=""):
  GreetingsPos(FName, BYear)

GreetingsKey2("Fabrice", 1988)  # OK
GreetingsKey2(BYear=1988, FName="Fabrice") # OK
GreetingsKey2(BYear=1988)  # OK
GreetingsKey2()
```

Keywords arguments

```
def GreetingsKey2(FName="", BYear=""):
  GreetingsPos(FName, BYear)

GreetingsKey2("Fabrice", 1988)  # OK
GreetingsKey2(BYear=1988, FName="Fabrice") # OK
GreetingsKey2(BYear=1988)  # OK
GreetingsKey2() # error
GreetingsKey2("Fabrice")
```

Keywords arguments

```
def GreetingsKey2(FName="", BYear=""):
  GreetingsPos(FName, BYear)

GreetingsKey2("Fabrice", 1988)  # OK
GreetingsKey2(BYear=1988, FName="Fabrice") # OK
GreetingsKey2(BYear=1988)  # OK
GreetingsKey2() # error
GreetingsKey2("Fabrice") # error
```

Keywords arguments

```
def GreetingsKey2(FName="", BYear=""):
  GreetingsPos(FName, BYear)

GreetingsKey2("Fabrice", 1988)  # OK
GreetingsKey2(BYear=1988, FName="Fabrice") # OK
GreetingsKey2(BYear=1988)  # OK
GreetingsKey2() # error
GreetingsKey2("Fabrice") # error
```

Keywords arguments forces a default value

4: Functions, Imports & Modules

1. Functions
   - Parameters
   - Scope of variables
   - Return values
2. Modules & Imports

Scope of variables

Variables are searched within scopes in a specific order:

- Local (scope of the current function)

Scope of variables

Variables are searched within scopes in a specific order:

- Local (scope of the current function)
- Global (global variables of the program)

Scope of variables

Variables are searched within scopes in a specific order:

- Local (scope of the current function)
- Global (global variables of the program)
- Internal (variables of the interpreter)

Scope of variables

Variables are searched within scopes in a specific order:

- Local (scope of the current function)
- Global (global variables of the program)
- Internal (variables of the interpreter)

LGI rule

Scope of variables

Variables are searched within scopes in a specific order:

- Local (scope of the current function)
- Global (global variables of the program)
- Internal (variables of the interpreter)

LGI rule

Constants are global variables (uppercase name)

4: Functions, Imports & Modules

1 Functions
- Parameters
- Scope of variables
- Return values

2 Modules & Imports

Multiple return of values

Multiple return of values

```
def ConvertTemperature(kelvin):
  celsius = kelvin - 273
  fahrenheit = ((celsius * 9) / 5) + 32
  reaumur = (kelvin * 4) / 5
  return celsius, fahrenheit, reaumur
```

Multiple return of values

```
def ConvertTemperature(kelvin):
  celsius = kelvin - 273
  fahrenheit = ((celsius * 9) / 5) + 32
  reaumur = (kelvin * 4) / 5
  return celsius, fahrenheit, reaumur

temps = ConvertTemperature(42)
```

Multiple return of values

```
def ConvertTemperature(kelvin):
  celsius = kelvin - 273
  fahrenheit = ((celsius * 9) / 5) + 32
  reaumur = (kelvin * 4) / 5
  return celsius, fahrenheit, reaumur

temps = ConvertTemperature(42)
print("°K : " + str(42))
print("°C : " + str(temps[0]))
print("°F : " + str(temps[1]))
print("°Re: " + str(temps[2]))
```

Multiple return of values

```
def ConvertTemperature(kelvin):
  celsius = kelvin - 273
  fahrenheit = ((celsius * 9) / 5) + 32
  reaumur = (kelvin * 4) / 5
  return celsius, fahrenheit, reaumur

temps = ConvertTemperature(42)
print("°K : " + str(42))        # 42
print("°C : " + str(temps[0])) # -231
print("°F : " + str(temps[1])) # -383
print("°Re: " + str(temps[2])) # 33
```

Multiple return of values

```
def ConvertTemperature(kelvin):
  celsius = kelvin - 273
  fahrenheit = ((celsius * 9) / 5) + 32
  reaumur = (kelvin * 4) / 5
  return celsius, fahrenheit, reaumur

temps = ConvertTemperature(42)
print("°K : " + str(42))       # 42
print("°C : " + str(temps[0])) # -231
print("°F : " + str(temps[1])) # -383
print("°Re: " + str(temps[2])) # 33
```

Multiple return values uses tuples

Multiple return of values

Multiple affectations are also possible

Multiple return of values

Multiple affectations are also possible

```
def ConvertTemperature(kelvin):
  celsius = kelvin - 273
  fahrenheit = ((celsius * 9) / 5) + 32
  reaumur = (kelvin * 4) / 5
  return celsius, fahrenheit, reaumur
```

Multiple return of values

Multiple affectations are also possible

```
def ConvertTemperature(kelvin):
  celsius = kelvin - 273
  fahrenheit = ((celsius * 9) / 5) + 32
  reaumur = (kelvin * 4) / 5
  return celsius, fahrenheit, reaumur

tC, tF, tR = ConvertTemperature(42)
```

Multiple return of values

Multiple affectations are also possible

```
def ConvertTemperature(kelvin):
  celsius = kelvin - 273
  fahrenheit = ((celsius * 9) / 5) + 32
  reaumur = (kelvin * 4) / 5
  return celsius, fahrenheit, reaumur

tC, tF, tR = ConvertTemperature(42)
print("°K : " + str(42))
print("°C : " + str(tC))
print("°F : " + str(tF))
print("°Re: " + str(tR))
```

Multiple return of values

Multiple affectations are also possible

```
def ConvertTemperature(kelvin):
  celsius = kelvin - 273
  fahrenheit = ((celsius * 9) / 5) + 32
  reaumur = (kelvin * 4) / 5
  return celsius, fahrenheit, reaumur

tC, tF, tR = ConvertTemperature(42)
print("°K : " + str(42)) # 42
print("°C : " + str(tC)) # -231
print("°F : " + str(tF)) # -383
print("°Re: " + str(tR)) # 33
```

4: Functions, Imports & Modules

1. Functions
2. Modules & Imports
   - Modules
   - Imports
   - Packages

4: Functions, Imports & Modules

1 Functions

2 Modules & Imports
- Modules
- Imports
- Packages

Modules

A module contains functions and variables (and classes)

Module name is the filename withtout **`.py`**

Modules

A module contains functions and variables (and classes)

Module name is the filename withtout **`.py`**

Some conventions for a nice documentation:

Modules

A module contains functions and variables (and classes)

Module name is the filename withtout **`.py`**

Some conventions for a nice documentation:

- Begin your module with a *docstring*
  - triple quotes `""" docstring """`
  - write the description of your module
  - docstring can be on multiple lines

Modules

A module contains functions and variables (and classes)

Module name is the filename withtout **`.py`**

Some conventions for a nice documentation:

- Begin your module with a *docstring*
  - triple quotes `""" docstring """`
  - write the description of your module
  - docstring can be on multiple lines
- Within each function, write a docstring about it

Modules

A module contains functions and variables (and classes)

Module name is the filename withtout **`.py`**

Some conventions for a nice documentation:

- Begin your module with a *docstring*
  - triple quotes `""" docstring """`
  - write the description of your module
  - docstring can be on multiple lines
- Within each function, write a docstring about it
- Eventually, add useful constants

Modules

A module contains functions and variables (and classes)

Module name is the filename withtout **`.py`**

Some conventions for a nice documentation:

- Begin your module with a *docstring*
  - triple quotes `""" docstring """`
  - write the description of your module
  - docstring can be on multiple lines
- Within each function, write a docstring about it
- Eventually, add useful constants

Modules

A module contains functions and variables (and classes)

Module name is the filename withtout **`.py`**

Some conventions for a nice documentation:

- Begin your module with a *docstring*
  - triple quotes `""" docstring """`
  - write the description of your module
  - docstring can be on multiple lines
- Within each function, write a docstring about it
- Eventually, add useful constants

Check result with **`help(module)`** (after importing it)

Modules

`MyModule.py` (module name: **`MyModule`**)

Modules

`MyModule.py` (module name: **`MyModule`**)

```python
""" Module for explaining modules """
```

Modules

MyModule.py (module name: **MyModule**)

```
""" Module for explaining modules """
MYCONST=42
```

Modules

`MyModule.py` (module name: **`MyModule`**)

```
""" Module for explaining modules """
MYCONST=42

def MyFunc(test):
  print("Hello World!")


def OtherFunc(var):
  print("Test.")
```

Modules

MyModule.py (module name: **MyModule**)

```
""" Module for explaining modules """
MYCONST=42

def MyFunc(test):
  """ Function for explaining modules """
  print("Hello World!")

def OtherFunc(var):
  """ Another function """
  print("Test.")
```

4: Functions, Imports & Modules

1 Functions

2 Modules & Imports
- Modules
- Imports
- Packages

Imports

Multiple ways for importing modules:

- Method 1: **`import MyModule`**

Imports

Multiple ways for importing modules:

- Method 1: **`import MyModule`**
- Method 2: **`import MyModule as MyM`**

Imports

Multiple ways for importing modules:

- Method 1: **`import MyModule`**
- Method 2: **`import MyModule as MyM`**
- Method 3: **`from MyModule import MyFunc`**

Imports

Multiple ways for importing modules:

- Method 1: **`import MyModule`**
- Method 2: **`import MyModule as MyM`**
- Method 3: **`from MyModule import MyFunc`**

Variables and constants can be imported too

Imports

Multiple ways for importing modules:

- Method 1: **`import MyModule`**
- Method 2: **`import MyModule as MyM`**
- Method 3: **`from MyModule import MyFunc`**

Variables and constants can be imported too

Beware: with method 1, *everything* is imported

Imports

Method 1: **`MyModule.py`**

Imports

Method 1: **`MyModule.py`**

```
import MyModule
# function: def MyFunc(test)
```

Imports

Method 1: **MyModule.py**

```
import MyModule
# function: def MyFunc(test)

MyModule.MyFunc(42)
```

Imports

Method 2: **`MyModule.py`**

Imports

Method 2: **MyModule.py**

```
import MyModule as MyM
# function: def MyFunc(test)
```

Imports

Method 2: **MyModule.py**

```
import MyModule as MyM
# function: def MyFunc(test)

MyM.MyFunc(42)
```

Imports

Method 3: **`MyModule.py`**

Imports

Method 3: **MyModule.py**

```
from MyModule import MyFunc
# function: def MyFunc(test)
```

Imports

Method 3: **MyModule.py**

```
from MyModule import MyFunc
# function: def MyFunc(test)

MyFunc(42)
```

4: Functions, Imports & Modules

1. Functions

2. Modules & Imports
   - Modules
   - Imports
   - Packages

Packages

Packages contain multiple modules and dependencies

Packages

Packages contain multiple modules and dependencies

(see documentation and tutorials about how to build one)

Part V

OOP and Classes

5: OOP and Classes

1 Object Oriented Programming

2 Classes in Python

OOP

Object Oriented Programming (OOP) vocabulary

OOP

Object Oriented Programming (OOP) vocabulary

- Class: the type and structure

OOP

Object Oriented Programming (OOP) vocabulary

- Class: the type and structure
- Attribute: a variable that is a member of the class

OOP

Object Oriented Programming (OOP) vocabulary

- Class: the type and structure
- Attribute: a variable that is a member of the class
- Method: a function/procedure that is a member of the class

OOP

Object Oriented Programming (OOP) vocabulary

- Class: the type and structure
- Attribute: a variable that is a member of the class
- Method: a function/procedure that is a member of the class
- Object: an instance of a class

OOP

Object Oriented Programming (OOP) vocabulary

- Class: the type and structure
- Attribute: a variable that is a member of the class
- Method: a function/procedure that is a member of the class
- Object: an instance of a class
- Constructor: the first method called when creating an object

OOP

Object Oriented Programming (OOP) vocabulary

- Class: the type and structure
- Attribute: a variable that is a member of the class
- Method: a function/procedure that is a member of the class
- Object: an instance of a class
- Constructor: the first method called when creating an object
- Destructor: the method called when deleting an object

OOP

Object Oriented Programming (OOP) vocabulary

- Class: the type and structure
- Attribute: a variable that is a member of the class
- Method: a function/procedure that is a member of the class
- Object: an instance of a class
- Constructor: the first method called when creating an object
- Destructor: the method called when deleting an object
- Inheritance: structure of a class reused as a basis for a new one

OOP

Object Oriented Programming (OOP) vocabulary

- Class: the type and structure
- Attribute: a variable that is a member of the class
- Method: a function/procedure that is a member of the class
- Object: an instance of a class
- Constructor: the first method called when creating an object
- Destructor: the method called when deleting an object
- Inheritance: structure of a class reused as a basis for a new one
- Private: when a member is accessible only by the object itself

OOP

Object Oriented Programming (OOP) vocabulary

- Class: the type and structure
- Attribute: a variable that is a member of the class
- Method: a function/procedure that is a member of the class
- Object: an instance of a class
- Constructor: the first method called when creating an object
- Destructor: the method called when deleting an object
- Inheritance: structure of a class reused as a basis for a new one
- Private: when a member is accessible only by the object itself
- Public: when a member is accessible by any object

OOP

Object Oriented Programming (OOP) concepts examples

- Class: the type and structure
- Attribute: a variable that is a member of the class
- Method: a function/procedure that is a member of the class
- Object: an instance of a class

OOP

Object Oriented Programming (OOP) concepts examples

- Class: the type and structure
- Attribute: a variable that is a member of the class
- Method: a function/procedure that is a member of the class
- Object: an instance of a class
- Classes: vehicle, car, airplane, boat, ...

OOP

Object Oriented Programming (OOP) concepts examples

- Class: the type and structure
- Attribute: a variable that is a member of the class
- Method: a function/procedure that is a member of the class
- Object: an instance of a class
- Classes: vehicle, car, airplane, boat, ...
- Attributes: speed, passengers, engine, ...

OOP

Object Oriented Programming (OOP) concepts examples

- Class: the type and structure
- Attribute: a variable that is a member of the class
- Method: a function/procedure that is a member of the class
- Object: an instance of a class

- Classes: vehicle, car, airplane, boat, ...
- Attributes: speed, passengers, engine, ...
- Methods: accelerate, decelerate, embark, ...

OOP

Object Oriented Programming (OOP) concepts examples

- Class: the type and structure
- Attribute: a variable that is a member of the class
- Method: a function/procedure that is a member of the class
- Object: an instance of a class

- Classes: vehicle, car, airplane, boat, ...
- Attributes: speed, passengers, engine, ...
- Methods: accelerate, decelerate, embark, ...
- Objects/Instances:
  - My Peugeot 206 (*BN-340-FT*)
  - Your Toyota Corolla (*UE-042-FI*)

OOP

Inheritance explanation

OOP

Inheritance explanation

- Parent Class: the class more general or abstract

OOP

Inheritance explanation

- Parent Class: the class more general or abstract
- Child Class / Sub Class: a specialized class derived from a parent

OOP

Inheritance explanation

- Parent Class: the class more general or abstract
- Child Class / Sub Class: a specialized class derived from a parent
- *the child class inherits attributes and methods from its parent*

OOP

Inheritance explanation

- Parent Class: the class more general or abstract
- Child Class / Sub Class: a specialized class derived from a parent
- *the child class inherits attributes and methods from its parent*

Attribute/Method access modifiers

OOP

Inheritance explanation

- Parent Class: the class more general or abstract
- Child Class / Sub Class: a specialized class derived from a parent
- *the child class inherits attributes and methods from its parent*

Attribute/Method access modifiers

- Private: only the class itself can access it

OOP

Inheritance explanation

- Parent Class: the class more general or abstract
- Child Class / Sub Class: a specialized class derived from a parent
- *the child class inherits attributes and methods from its parent*

Attribute/Method access modifiers

- Private: only the class itself can access it
- Protected: the class itself and its childs can access it

OOP

Inheritance explanation

- Parent Class: the class more general or abstract
- Child Class / Sub Class: a specialized class derived from a parent
- *the child class inherits attributes and methods from its parent*

Attribute/Method access modifiers

- Private: only the class itself can access it
- Protected: the class itself and its childs can access it
- Public: any class can access it

OOP

Inheritance example

- Parent Class: the class more general or abstract
- Child Class / Sub Class: a specialized class derived from a parent
- *the child class inherits attributes and methods from its parent*

OOP

Inheritance example

- Parent Class: the class more general or abstract
- Child Class / Sub Class: a specialized class derived from a parent
- *the child class inherits attributes and methods from its parent*

- Class: vehicle

OOP

Inheritance example

- Parent Class: the class more general or abstract
- Child Class / Sub Class: a specialized class derived from a parent
- *the child class inherits attributes and methods from its parent*

- Class: vehicle
- Child classes: car, airplane, boat, ...

OOP

Inheritance example

- Parent Class: the class more general or abstract
- Child Class / Sub Class: a specialized class derived from a parent
- *the child class inherits attributes and methods from its parent*

- Class: vehicle
- Child classes: car, airplane, boat, ...
- Inherited Attributes: speed, passengers, ...

5: OOP and Classes

1 Object Oriented Programming

2 Classes in Python

Classes in Python

Specificities of classes in Python:

Classes in Python

Specificities of classes in Python:

- Only one constructor: `__init__`

Classes in Python

Specificities of classes in Python:

- Only one constructor: `__init__`
- No destructor (python manages the memory by references)

Classes in Python

Specificities of classes in Python:

- Only one constructor: `__init__`
- No destructor (python manages the memory by references)
- Attributes are at least *read-only*, and eventually *writable*

Classes in Python

Specificities of classes in Python:

- Only one constructor: `__init__`
- No destructor (python manages the memory by references)
- Attributes are at least *read-only*, and eventually *writable*
- Writable attributes can be deleted from the object

Classes in Python

Specificities of classes in Python:

- Only one constructor: `__init__`
- No destructor (python manages the memory by references)
- Attributes are at least *read-only*, and eventually *writable*
- Writable attributes can be deleted from the object
- Member beginning by an underscore (_) aren't strictly private, but should be considered internal to the class

Classes in Python

Specificities of classes in Python:

- Only one constructor: `__init__`
- No destructor (python manages the memory by references)
- Attributes are at least *read-only*, and eventually *writable*
- Writable attributes can be deleted from the object
- Member beginning by an underscore (_) aren't strictly private, but should be considered internal to the class
- **`self`** keyword is required as the first parameter of each method

Classes in Python

Classes in Python

```
class Vehicle:
    """ General vehicles """
```

Classes in Python

```python
class Vehicle:
  """ General vehicles """
  __speed = 0
  Passengers = 0
```

Classes in Python

```
class Vehicle:
  """ General vehicles """
  __speed = 0
  Passengers = 0
  # Constructor
  def __init__(self):
    self.Passengers = 1
```

Classes in Python

```
class Vehicle:
  """ General vehicles """
  __speed = 0
  Passengers = 0
  # Constructor
  def __init__(self):
    self.Passengers = 1
  # Method
  def getSpeed(self):
    return (self.__speed)
```

Classes in Python

```
class Vehicle:
  """ General vehicles """
  __speed = 0
  Passengers = 0
  # Constructor
  def __init__(self):
    self.Passengers = 1
  # Methods
  def getSpeed(self):
    return (self.__speed)
  def Accelerate(self, x):
    self.__speed += x
```

Classes in Python

```
class Vehicle:
  """ General vehicles """
  __speed = 0
  Passengers = 0
  # Constructor
  def __init__(self):          # Constructor
    self.Passengers = 1
  # Methods
  def getSpeed(self):          # Accessor
    return (self.__speed)
  def Accelerate(self, x):  # Mutator
    self.__speed += x
```

Classes in Python

Classes in Python

```
class Car(Vehicle):
    """ Cars inherit from Vehicle """
    __CV = 0
```

Classes in Python

```
class Car(Vehicle):
  """ Cars inherit from Vehicle """
  __CV = 0
  # Constructor
  def __init__(self, CO2, P):
    self.Passengers = 1
    self.__CV = (CO2 / 45) + (P / 40)
```

Classes in Python

```
class Car(Vehicle):
  """ Cars inherit from Vehicle """
  __CV = 0
  # Constructor
  def __init__(self, CO2, P):
    self.Passengers = 1
    self.__CV = (CO2 / 45) + (P / 40)
  # Method
  def getCV(self):
    return (self.__CV)
```

Classes in Python

Classes in Python

```
Peugeot206Plus = Car(110, 44)  # 110g/km  44kW
AirbusA340 = Vehicle()
```

Classes in Python

```
Peugeot206Plus = Car(110, 44)  # 110g/km  44kW
AirbusA340 = Vehicle()

Peugeot206Plus.Accelerate(80)
AirbusA340.Accelerate(260)
```

Classes in Python

```
Peugeot206Plus = Car(110, 44)  # 110g/km  44kW
AirbusA340 = Vehicle()

Peugeot206Plus.Accelerate(80)
AirbusA340.Accelerate(260)

Peugeot206Plus.getSpeed()  # 80
AirbusA340.getSpeed()      # 260

Peugeot206Plus.getCV()
AirbusA340.getCV()
```

Classes in Python

```
Peugeot206Plus = Car(110, 44)  # 110g/km  44kW
AirbusA340 = Vehicle()

Peugeot206Plus.Accelerate(80)
AirbusA340.Accelerate(260)

Peugeot206Plus.getSpeed()  # 80
AirbusA340.getSpeed()      # 260

Peugeot206Plus.getCV()
AirbusA340.getCV()  # Error
```

Method Overriding in Python

Method Overriding: when a method is redefined in the child class

Method Overriding in Python

Method Overriding: when a method is redefined in the child class

- Child method is called if redefined

Method Overriding in Python

Method Overriding: when a method is redefined in the child class

- Child method is called if redefined
- Use **`super()`** on current class for calling its parent method: **`super(Class, self).Method()`**

Method Overriding in Python

Method Overriding in Python

```
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")
```

Method Overriding in Python

```
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    print("--Cube--")
```

Method Overriding in Python

```
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    print("--Cube--")
```

Method Overriding in Python

```python
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    print("--Cube--")
```

```python
S = Shape()
C = Cube()
```

Method Overriding in Python

```
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    print("--Cube--")
```

```
S = Shape()
C = Cube()


S.Hello()
C.Hello()
```

Method Overriding in Python

```python
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    print("--Cube--")
```

```python
S = Shape()
C = Cube()


S.Hello() # Shape: Hello!
C.Hello() # Cube: Hello!
```

Method Overriding in Python

```python
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    print("--Cube--")
```

```python
S = Shape()
C = Cube()


S.Hello() # Shape: Hello!
C.Hello() # Cube: Hello!

S.SayShape()
C.SayShape()
```

Method Overriding in Python

```python
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    print("--Cube--")
```

```python
S = Shape()
C = Cube()


S.Hello() # Shape: Hello!
C.Hello() # Cube: Hello!

S.SayShape() # --Shape--
C.SayShape() # --Shape--
```

Method Overriding in Python

```
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    print("--Cube--")
```

```
S = Shape()
C = Cube()


S.Hello() # Shape: Hello!
C.Hello() # Cube: Hello!

S.SayShape() # --Shape--
C.SayShape() # --Shape--

S.SayCube()
C.SayCube()
```

Method Overriding in Python

```python
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    print("--Cube--")
```

```python
S = Shape()
C = Cube()


S.Hello() # Shape: Hello!
C.Hello() # Cube: Hello!

S.SayShape() # --Shape--
C.SayShape() # --Shape--

S.SayCube() # ERROR
C.SayCube() # --Cube--
```

```python
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")
```

```
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    super(Cube, self).
   SayShape()
    print("--Cube--")
```

```
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    super(Cube, self).
   SayShape()
    print("--Cube--")
```

```python
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    super(Cube, self).
   SayShape()
    print("--Cube--")
```

```python
S = Shape()
C = Cube()
```

```python
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    super(Cube, self).
   SayShape()
    print("--Cube--")
```

```python
S = Shape()
C = Cube()


S.Hello()
C.Hello()
```

```python
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    super(Cube, self).
   SayShape()
    print("--Cube--")
```

```python
S = Shape()
C = Cube()


S.Hello() # Shape: Hello!
C.Hello() # Cube: Hello!
```

```python
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    super(Cube, self).
   SayShape()
    print("--Cube--")
```

```python
S = Shape()
C = Cube()


S.Hello() # Shape: Hello!
C.Hello() # Cube: Hello!

S.SayShape()
C.SayShape()
```

```
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    super(Cube, self).
   SayShape()
    print("--Cube--")
```

```
S = Shape()
C = Cube()


S.Hello() # Shape: Hello!
C.Hello() # Cube: Hello!

S.SayShape() # --Shape--
C.SayShape() # --Shape--
```

```python
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    super(Cube, self).
 SayShape()
    print("--Cube--")
```

```python
S = Shape()
C = Cube()


S.Hello() # Shape: Hello!
C.Hello() # Cube: Hello!

S.SayShape() # --Shape--
C.SayShape() # --Shape--

S.SayCube()
C.SayCube()
```

```python
class Shape:
  def Hello(self):
    print("Shape: Hello!")

  def SayShape(self):
    print("--Shape--")

class Cube(Shape):
  def Hello(self):
    print("Cube: Hello!")

  def SayCube(self):
    super(Cube, self).
 SayShape()
    print("--Cube--")
```

```python
S = Shape()
C = Cube()


S.Hello() # Shape: Hello!
C.Hello() # Cube: Hello!

S.SayShape() # --Shape--
C.SayShape() # --Shape--

S.SayCube() # ERROR
C.SayCube() # --Shape--
            # --Cube--
```

Summary of Classes in Python

Summary of vocabulary:

Summary of Classes in Python

Summary of vocabulary:

- **`Vehicle`**: class

Summary of Classes in Python

Summary of vocabulary:

- **`Vehicle`**: class
- **`Car`**: class (inherits from **`Vehicle`**)

Summary of Classes in Python

Summary of vocabulary:

- **`Vehicle`**: class
- **`Car`**: class (inherits from **`Vehicle`**)
- **`Peugeot206Plus`**: instance of **`Car`**

Summary of Classes in Python

Summary of vocabulary:

- **`Vehicle`**: class
- **`Car`**: class (inherits from **`Vehicle`**)
- **`Peugeot206Plus`**: instance of **`Car`**
- **`AirbusA340`**: instance of **`Vehicle`**

Summary of Classes in Python

Summary of vocabulary:

- **`Vehicle`**: class
- **`Car`**: class (inherits from **`Vehicle`**)
- **`Peugeot206Plus`**: instance of **`Car`**
- **`AirbusA340`**: instance of **`Vehicle`**
- **`Peugeot206Plus`** can calculate and give **`CV`**

Summary of Classes in Python

Summary of vocabulary:

- **`Vehicle`**: class
- **`Car`**: class (inherits from **`Vehicle`**)
- **`Peugeot206Plus`**: instance of **`Car`**
- **`AirbusA340`**: instance of **`Vehicle`**
- **`Peugeot206Plus`** can calculate and give **`CV`**
- **`AirbusA340`**: doesn't have **`CV`** method nor attribute

Summary of Classes in Python

Summary of vocabulary:

- **`Vehicle`**: class
- **`Car`**: class (inherits from **`Vehicle`**)
- **`Peugeot206Plus`**: instance of **`Car`**
- **`AirbusA340`**: instance of **`Vehicle`**
- **`Peugeot206Plus`** can calculate and give **`CV`**
- **`AirbusA340`**: doesn't have **`CV`** method nor attribute

`https://docs.python.org/3/tutorial/classes.html`

Part VI

Files and Directories

6: Files and Directories

1. Files Processing
   - Open, Read, Write, Close, Seek, Tell
   - Read & Write examples
2. File existence and management

6: Files and Directories

1. Files Processing
   - Open, Read, Write, Close, Seek, Tell
   - Read & Write examples
2. File existence and management

Files Operations

Files are managed as usual:

Files Operations

Files are managed as usual:

- **`open`**
- **`read`**
- **`write`**
- **`close`**
- **`seek`**
- **`tell`**

Files Operations

Files are managed as usual:

- **`open(filename, mode, `*`[encoding]`*`)`**
- **`read`**
- **`write`**
- **`close`**
- **`seek`**
- **`tell`**

Files Operations

Files are managed as usual:

- **`open(filename, mode, [encoding])`**
- **`read(NbCharacters)`**
- **`write`**
- **`close`**
- **`seek`**
- **`tell`**

Files Operations

Files are managed as usual:

- **`open(filename, mode, [encoding])`**
- **`read(NbCharacters)`**
- **`write(string)`**
- **`close`**
- **`seek`**
- **`tell`**

Files Operations

Files are managed as usual:

- **`open(filename, mode, `*`[encoding]`*`)`**
- **`read(NbCharacters)`**
- **`write(string)`**
- **`close()`**
- **`seek`**
- **`tell`**

Files Operations

Files are managed as usual:

- **`open(filename, mode,`** ***`[encoding]`*** **`)`**
- **`read(NbCharacters)`**
- **`write(string)`**
- **`close()`**
- **`seek(offset,`** ***`[whence]`*** **`)`**
- **`tell`**

Files Operations

Files are managed as usual:

- **`open(filename, mode, [encoding])`**
- **`read(NbCharacters)`**
- **`write(string)`**
- **`close()`**
- **`seek(offset, [whence])`**
- **`tell()`**

Files Operations

Files are managed as usual:

- **`open(filename, mode, `*`[encoding]`*`)`**
- **`read(NbCharacters)`**
- **`write(string)`**
- **`close()`**
- **`seek(offset, `*`[whence]`*`)`**
- **`tell()`**

Always close files in order to be sure to write changes on the physical support

Files Operations

Files are managed as usual:

- **`open(filename, mode, [encoding])`**
- **`read(NbCharacters)`**
- **`write(string)`**
- **`close()`**
- **`seek(offset, [whence])`**
- **`tell()`**

Always close files in order to be sure to write changes on the physical support

Files are managed by an internal python's **class**

Open

Open mode:

Open

Open mode:

- **`"r"`** - open for reading

Open

Open mode:

- **"r"** - open for reading
- **"w"** - truncate the file for overwriting

Open

Open mode:

- **`"r"`** - open for reading
- **`"w"`** - truncate the file for overwriting
- **`"a"`** - append/begin from the end for writing

Open

Open mode:

- **"r"** - open for reading
- **"w"** - truncate the file for overwriting
- **"a"** - append/begin from the end for writing
- **"x"** - create a new file, fail if it already exists

Open

Open mode:

- **`"r"`** - open for reading
- **`"w"`** - truncate the file for overwriting
- **`"a"`** - append/begin from the end for writing
- **`"x"`** - create a new file, fail if it already exists
- **`"r+"`** - open for reading and writing

Open

Open mode:

- **"r"** - open for reading
- **"w"** - truncate the file for overwriting
- **"a"** - append/begin from the end for writing
- **"x"** - create a new file, fail if it already exists
- **"r+"** - open for reading and writing
- **"b"** - binary mode (read/write with bytes objects)

Open

Open mode:

- **"r"** - open for reading
- **"w"** - truncate the file for overwriting
- **"a"** - append/begin from the end for writing
- **"x"** - create a new file, fail if it already exists
- **"r+"** - open for reading and writing
- **"b"** - binary mode (read/write with bytes objects)

Open encoding:

Open

Open mode:

- **"r"** - open for reading
- **"w"** - truncate the file for overwriting
- **"a"** - append/begin from the end for writing
- **"x"** - create a new file, fail if it already exists
- **"r+"** - open for reading and writing
- **"b"** - binary mode (read/write with bytes objects)

Open encoding:

- **encoding="utf-8"** - read/write in `utf-8`

Open

Open mode:

- **"r"** - open for reading
- **"w"** - truncate the file for overwriting
- **"a"** - append/begin from the end for writing
- **"x"** - create a new file, fail if it already exists
- **"r+"** - open for reading and writing
- **"b"** - binary mode (read/write with bytes objects)

Open encoding:

- **encoding="utf-8"** - read/write in `utf-8`
- Can be omitted for platform encoding *[do this for now]*

Open

Open mode:

- **`"r"`** - open for reading
- **`"w"`** - truncate the file for overwriting
- **`"a"`** - append/begin from the end for writing
- **`"x"`** - create a new file, fail if it already exists
- **`"r+"`** - open for reading and writing
- **`"b"`** - binary mode (read/write with bytes objects)

Open encoding:

- **`encoding="utf-8"`** - read/write in `utf-8`
- Can be omitted for platform encoding *[do this for now]*
- Cannot be used when in binary mode

Text & Binary mode

Text mode:

Text & Binary mode

Text mode: **`open(filename, "r+")`**

Text & Binary mode

Text mode: **`open(filename, "r+")`**

- For reading/writting text files (CSV, configuration, log, ...)

Text & Binary mode

Text mode: **`open(filename, "r+")`**

- For reading/writting text files (CSV, configuration, log, ...)
- When writing in a file: **`\n`** is transformed into platform specific line ending (**`\r\n`**, **`\n\r`**, ...)

Text & Binary mode

Text mode: **`open(filename, "r+")`**

- For reading/writting text files (CSV, configuration, log, ...)
- When writing in a file: **`\n`** is transformed into platform specific line ending (**`\r\n`**, **`\n\r`**, ...)
- When reading from a file: platform specific line endings are transformed into simple **`\n`**

Text & Binary mode

Text mode: **`open(filename, "r+")`**

- For reading/writting text files (CSV, configuration, log, ...)
- When writing in a file: **`\n`** is transformed into platform specific line ending (**`\r\n`**, **`\n\r`**, ...)
- When reading from a file: platform specific line endings are transformed into simple **`\n`**

Binary mode:

Text & Binary mode

Text mode: **`open(filename, "r+")`**

- For reading/writting text files (CSV, configuration, log, ...)
- When writing in a file: **`\n`** is transformed into platform specific line ending (**`\r\n`**, **`\n\r`**, ...)
- When reading from a file: platform specific line endings are transformed into simple **`\n`**

Binary mode: **`open(filename, "wb")`**

Text & Binary mode

Text mode: **`open(filename, "r+")`**

- For reading/writting text files (CSV, configuration, log, ...)
- When writing in a file: **`\n`** is transformed into platform specific line ending (**`\r\n`**, **`\n\r`**, ...)
- When reading from a file: platform specific line endings are transformed into simple **`\n`**

Binary mode: **`open(filename, "wb")`**

- For reading/writting binary data (music, video, web page, ...)

Text & Binary mode

Text mode: **`open(filename, "r+")`**

- For reading/writting text files (CSV, configuration, log, ...)
- When writing in a file: **`\n`** is transformed into platform specific line ending (**`\r\n`**, **`\n\r`**, ...)
- When reading from a file: platform specific line endings are transformed into simple **`\n`**

Binary mode: **`open(filename, "wb")`**

- For reading/writting binary data (music, video, web page, ...)
- The line ending characters are never transformed when reading/writting

Text & Binary mode

Text mode: **`open(filename, "r+")`**

- For reading/writting text files (CSV, configuration, log, ...)
- When writing in a file: **`\n`** is transformed into platform specific line ending (**`\r\n`**, **`\n\r`**, ...)
- When reading from a file: platform specific line endings are transformed into simple **`\n`**

Binary mode: **`open(filename, "wb")`**

- For reading/writting binary data (music, video, web page, ...)
- The line ending characters are never transformed when reading/writting

For now: do not use binary mode, except if you work on raw data

Read & Write

Read:

- **`read()`** - get the whole text

Read & Write

Read:

- **`read()`** - get the whole text
- **`read(Nb)`** - get the next *Nb* characters

Read & Write

Read:

- **`read()`** - get the whole text
- **`read(Nb)`** - get the next *Nb* characters
- **`readline()`** - get the next line (whole text until next **`\n`**)

Read & Write

Read:

- **`read()`** - get the whole text
- **`read(Nb)`** - get the next *Nb* characters
- **`readline()`** - get the next line (whole text until next **`\n`**)

Write:

Read & Write

Read:

- **`read()`** - get the whole text
- **`read(Nb)`** - get the next *Nb* characters
- **`readline()`** - get the next line (whole text until next **`\n`**)

Write:

- **`open(filename, "wb")`**

Read & Write

Read:

- **`read()`** - get the whole text
- **`read(Nb)`** - get the next *Nb* characters
- **`readline()`** - get the next line (whole text until next **`\n`**)

Write:

- **`open(filename, "wb")`**
- **`write(str)`** - Write the string and return how many characters were written

Tell & Seek

Tell:

- **`tell()`** - get current position in the file (in bytes)

Tell & Seek

Tell:

- **`tell()`** - get current position in the file (in bytes)
- Useful in binary mode

Tell & Seek

Tell:

- **`tell()`** - get current position in the file (in bytes)
- Useful in binary mode
- More difficult in text mode (because of the *encoding)*

Tell & Seek

Tell:

- **`tell()`** - get current position in the file (in bytes)
- Useful in binary mode
- More difficult in text mode (because of the *encoding)*

Seek:

Tell & Seek

Tell:

- **`tell()`** - get current position in the file (in bytes)
- Useful in binary mode
- More difficult in text mode (because of the *encoding)*

Seek:

- **`seek(offset)`** - put read/write cursor at the offset
- **`seek(offset, whence)`** - move the cursor relatively

Tell & Seek

Tell:

- **`tell()`** - get current position in the file (in bytes)
- Useful in binary mode
- More difficult in text mode (because of the *encoding)*

Seek:

- **`seek(offset)`** - put read/write cursor at the offset
- **`seek(offset, whence)`** - move the cursor relatively
- **`whence=`**
  - **0** : from the beginning of the file *[default]*
  - **1** : from the current cursor position
  - **2** : from the end of the file

6: Files and Directories

1. Files Processing
   - Open, Read, Write, Close, Seek, Tell
   - Read & Write examples
2. File existence and management

Read (1) example

Open existing file for reading

Read (1) example

Open existing file for reading

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

Read (1) example

Open existing file for reading

```
f = open("file.txt", "r")
chars = f.read(10)
print(chars)
```

Read (1) example

Open existing file for reading

```
f = open("file.txt", "r")
chars = f.read(10)
print(chars)
line = f.readline()
print(line)
```

Read (1) example

Open existing file for reading

```
f = open("file.txt", "r")
chars = f.read(10)
print(chars)
line = f.readline()
print(line)
f.close()
```

Read (2) example

Read line by line a file with python specificities

Read (2) example

Read line by line a file with python specificities

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

Read (2) example

Read line by line a file with python specificities

```
f = open("file.txt", "r")
for x in f:
  print(x)
```

Read (2) example

Read line by line a file with python specificities

```
f = open("file.txt", "r")
for x in f:
  print(x)
f.close()
```

Write example

Write at the end of a file (add content)

Write example

Write at the end of a file (add content)

```
f = open("file.txt", "a")
```

Write example

Write at the end of a file (add content)

```
f = open("file.txt", "a")
nb = f.write("-add content-")
```

Write example

Write at the end of a file (add content)

```
f = open("file.txt", "a")
nb = f.write("-add content-")
f.close()
```

Write example

Write at the end of a file (add content)

```
f = open("file.txt", "a")
nb = f.write("-add content-")
f.close()
print(nb)  # 13 chars were added
```

Documentation

Check documentation for more informations

`https://docs.python.org/3/tutorial/inputoutput.html`

6: Files and Directories

1. Files Processing
2. File existence and management

Files and Directories

Management of files is done through the **`os`** module

Files and Directories

Management of files is done through the **`os`** module

- **`os.chdir(path)`** - change the current directory

Files and Directories

Management of files is done through the **`os`** module

- **`os.chdir(path)`** - change the current directory
- **`os.mkdir(dirname)`** - create a directory

Files and Directories

Management of files is done through the **`os`** module

- **`os.chdir(path)`** - change the current directory
- **`os.mkdir(dirname)`** - create a directory
- **`os.makedirs(path)`** - create the full path (recursively)

Files and Directories

Management of files is done through the **`os`** module

- **`os.chdir(path)`** - change the current directory
- **`os.mkdir(dirname)`** - create a directory
- **`os.makedirs(path)`** - create the full path (recursively)
- **`os.remove(filename)`** - remove the file

Files and Directories

Management of files is done through the **`os`** module

- **`os.chdir(path)`** - change the current directory
- **`os.mkdir(dirname)`** - create a directory
- **`os.makedirs(path)`** - create the full path (recursively)
- **`os.remove(filename)`** - remove the file
- **`os.rmdir(dirname)`** - remove the directory

Files and Directories

Management of files is done through the **`os`** module

- **`os.chdir(path)`** - change the current directory
- **`os.mkdir(dirname)`** - create a directory
- **`os.makedirs(path)`** - create the full path (recursively)
- **`os.remove(filename)`** - remove the file
- **`os.rmdir(dirname)`** - remove the directory
- **`os.listdir(path)`** - list files in a directory

Files and Directories

Management of files is done through the **os** module

- **os.chdir(path)** - change the current directory
- **os.mkdir(dirname)** - create a directory
- **os.makedirs(path)** - create the full path (recursively)
- **os.remove(filename)** - remove the file
- **os.rmdir(dirname)** - remove the directory
- **os.listdir(path)** - list files in a directory
- **os.scandir(path)** - list files in a directory (generator)

File type

Type of file and names are done with **`path`** submodule

File type

Type of file and names are done with **`path`** submodule

- **`os.path.exist(path)`** - test if *path* is an existing file

File type

Type of file and names are done with **`path`** submodule

- **`os.path.exist(path)`** - test if *path* is an existing file
- **`os.path.isfile(path)`** - test if *path* is a file

File type

Type of file and names are done with **`path`** submodule

- **`os.path.exist(path)`** - test if *path* is an existing file
- **`os.path.isfile(path)`** - test if *path* is a file
- **`os.path.isdir(path)`** - test if *path* is a directory

File type

Type of file and names are done with **`path`** submodule

- **`os.path.exist(path)`** - test if *path* is an existing file
- **`os.path.isfile(path)`** - test if *path* is a file
- **`os.path.isdir(path)`** - test if *path* is a directory
- **`os.path.abspath(path)`** - get absolute pathname

File type

Type of file and names are done with **`path`** submodule

- **`os.path.exist(path)`** - test if *path* is an existing file
- **`os.path.isfile(path)`** - test if *path* is a file
- **`os.path.isdir(path)`** - test if *path* is a directory
- **`os.path.abspath(path)`** - get absolute pathname
- **`os.path.dirname(path)`** - get dirname of pathname

File type

Type of file and names are done with **`path`** submodule

- **`os.path.exist(path)`** - test if *path* is an existing file
- **`os.path.isfile(path)`** - test if *path* is a file
- **`os.path.isdir(path)`** - test if *path* is a directory
- **`os.path.abspath(path)`** - get absolute pathname
- **`os.path.dirname(path)`** - get dirname of pathname
- **`os.path.basename(path)`** - get basename of pathname

File type

Type of file and names are done with **`path`** submodule

- **`os.path.exist(path)`** - test if *path* is an existing file
- **`os.path.isfile(path)`** - test if *path* is a file
- **`os.path.isdir(path)`** - test if *path* is a directory
- **`os.path.abspath(path)`** - get absolute pathname
- **`os.path.dirname(path)`** - get dirname of pathname
- **`os.path.basename(path)`** - get basename of pathname
- **`os.path.join(parent, child)`** - build a pathname from two paths

File type

Type of file and names are done with **`path`** submodule

- **`os.path.exist(path)`** - test if *path* is an existing file
- **`os.path.isfile(path)`** - test if *path* is a file
- **`os.path.isdir(path)`** - test if *path* is a directory
- **`os.path.abspath(path)`** - get absolute pathname
- **`os.path.dirname(path)`** - get dirname of pathname
- **`os.path.basename(path)`** - get basename of pathname
- **`os.path.join(parent, child)`** - build a pathname from two paths

`https://docs.python.org/3/library/os.path.html`

File browsing example

Browse a directory:

File browsing example

Browse a directory:

```
import os
```

File browsing example

Browse a directory:

```
import os

entries = os.listdir(".")
```

File browsing example

Browse a directory:

```
import os

# Print files & dirs
entries = os.listdir(".")
for entry in entries:
  print(str(entry))
```

File browsing example

Browse a directory:

```
import os

# Print files & dirs
entries = os.listdir(".")
for entry in entries:
  print(str(entry))


files = [f for f in os.listdir() if os.path.isfile(f)]
```

File browsing example

Browse a directory:

```
import os

# Print files & dirs
entries = os.listdir(".")
for entry in entries:
  print(str(entry))

# Print files only
files = [f for f in os.listdir() if os.path.isfile(f)]
for f in files:
  print(str(f))
```

Part VII

Python's Main Concepts

Built-in types

- Numerics
- Sequences
- Mappings
- Classes
- Instances
- Exceptions

7: Python's Main Concepts

1 Numeric types

2 Sequence types

3 Mapping types

4 Other types

Numeric types

- int
- float
- complex

Numeric types

- int
  - booleans - *(subtype of int)*
- float
- complex

Numeric types

- int
  - booleans - *(subtype of int)*
- float
- complex
  - *(pair of floats for real and imaginary parts)*

Numeric types

Type	Examples		
int	0	42	1337
float	0.0	1.8e30	1.
complex	3j	2J	5+7j

"1.8e30" is equivalent to "$1{,}8 \times 10^{30}$"

"5j" is equivalent in french to "5i"

Numeric operations

Operation	Result
x + y	sum of x and y
x − y	difference of x and y
x ∗ y	product of x and y
x / y	quotient of x and y
x // y	floored quotient of x and y
x % y	remainder of x/y
pow(x, y)	x to the power y
x ∗ ∗ y	x to the power y
−x	x negated
+x	x unchanged
abs(x)	absolute value or magnitude of x

"/" gives a float as a result

"//" gives an integer as a result (euclidean division)

Numeric operations

Operation	Result
abs(x)	absolute value or magnitude of x
floor(x)	the largest integer not greater than x
ceil(x)	the smallest integer greater than or equal to x

```
import math

Absolue = abs(-42.7)  # 42.7
ParDefaut1 = math.floor(3.2)  # 3
ParDefaut2 = math.floor(3.7)  # 3
ParExces1 = math.ceil(5.2)  # 6
ParExces2 = math.ceil(5.7)  # 6
```

Numeric operations

Constructors and type conversion:

Operation	Result
int(x)	*x* converted to integer
float(x)	*x* converted to floating point
complex(re, im)	a complex number with real part *re*, imaginary part *im*. (*im* defaults to zero)

```
entier = int(42.7)        # 42
flottant = float(3)       # 3.0
complexe1 = complex(4)        # 4+0j
complexe2 = complex(5, 6)   # 5+6j
```

Other numeric types

Booleans values:

- **`False`** (equivalent to **`int(0)`**)
- **`True`** (equivalent to **`int(1)`**)

More details on types

https://docs.python.org/3/library/stdtypes.html

7: Python's Main Concepts

1. Numeric types
2. Sequence types
   - Sequence operations
   - Lists
   - Tuples
   - Ranges
3. Mapping types
4. Other types

Sequence types

- list
- tuple
- range

Sequence types

- list
- tuple
- range
- str - *(text sequence type)*

Sequence types

- list
- tuple
- range
- str - *(text sequence type)*
- *binary sequence types*

Sequence types

- list
- tuple
- range
- str - *(text sequence type)*
- *binary sequence types*

index begins at 0

7: Python's Main Concepts

1. Numeric types

2. Sequence types
   - Sequence operations
   - Lists
   - Tuples
   - Ranges

3. Mapping types

4. Other types

Sequence operations

Sequences do have common operations:

- Access
- Search
- Concatenation
- min/max/len
- ...

Sequence operations

Operation	Result
x in s	**`True`** if an item of s is equal to x, else **`False`**
x not in s	**`False`** if an item of s is equal to x, else **`True`**
s + t	the concatenation of s and t
s * n	equivalent to adding s to itself n times
s[i]	access to the ith item of s (first index: 0)
s[i:j]	slice of s from i (included) to j (excluded)
s[i:j:k]	slice of s from i to j (excluded) with step k
len(x)	length of s
min(x)	smallest item of s
max(x)	largest item of s

s and t are sequences of the same type

n, i, j and k are integers

x is an arbitrary object

Sequence operations

Access to items:

```
mylist = [ 42, 1337, 42, 666, 15 ]
```

Sequence operations

Access to items:

```
mylist = [ 42, 1337, 42, 666, 15 ]
print(len(mylist)) # length: 5
```

Sequence operations

Access to items:

```
mylist = [ 42, 1337, 42, 666, 15 ]
print(len(mylist)) # length: 5
print(min(mylist)) # smallest: 15
print(max(mylist)) # biggest: 1337
```

Sequence operations

Access to items:

```
mylist = [ 42, 1337, 42, 666, 15 ]
print(len(mylist)) # length: 5
print(min(mylist)) # smallest: 15
print(max(mylist)) # biggest: 1337
print(42 in mylist)  # test presence 42: True
print(mylist[3])       # 4th item: 666
```

Sequence operations

Access to items:

```
mylist = [ 42, 1337, 42, 666, 15 ]
print(len(mylist)) # length: 5
print(min(mylist)) # smallest: 15
print(max(mylist)) # biggest: 1337
print(42 in mylist)  # test presence 42: True
print(mylist[3])        # 4th item: 666
print(mylist[1:3])      # slice: 1337, 42
```

Sequence operations

Access to items:

```
mylist = [ 42, 1337, 42, 666, 15 ]
print(len(mylist)) # length: 5
print(min(mylist)) # smallest: 15
print(max(mylist)) # biggest: 1337
print(42 in mylist)  # test presence 42: True
print(mylist[3])       # 4th item: 666
print(mylist[1:3])     # slice: 1337, 42
print(mylist[0:4:2])   # slice: 42, 42
```

Sequence operations

Operation	Result
s[i]	access to the *i*th item of *s* [first index: 0]
s[i:j]	slice beginning at *i* ending before *j* [ends at *j* - *1*]
s[i:j:k]	slice beginning at *i* ending before *j* by steps of *k*
s.count(x)	total number of occurrences of *x* in *s*
s.index(x)	index of the first occurrence of *x* in *s*
s.index(x, i)	index of the first occurrence of *x* in *s* at or after index *i*
s.index(x, i, j)	index of the first occurrence of *x* in *s* at or after index *i* and before index *j*

s and *t* are sequences of the same type

n, *i*, *j* and *k* are integers

x is an arbitrary object

Sequence operations

Slicing tips (1):

- s[:3] is equivalent to s[0:3]
  (prints everything from start until 3)

Sequence operations

Slicing tips (1):

- s[:3] is equivalent to s[0:3]
  (prints everything from start until 3)
- s[5:] is equivalent to s[5:*(last index + 1)*]
  (prints everything from 5)

Sequence operations

Slicing tips (1):

- s[:3] is equivalent to s[0:3]
  (prints everything from start until 3)
- s[5:] is equivalent to s[5:*(last index + 1)*]
  (prints everything from 5)
- s[:] is equivalent to s[0:*(last index + 1)*]
  (prints everything)

Sequence operations

Slicing tips (1):

- s[:3] is equivalent to s[0:3]
  (prints everything from start until 3)
- s[5:] is equivalent to s[5:*(last index + 1)*]
  (prints everything from 5)
- s[:] is equivalent to s[0:*(last index + 1)*]
  (prints everything)
- s[::2] is equivalent to s[0:*(last index + 1)*:2]
  (prints everything by steps of 2)

Sequence operations

Slicing tips (1):

- s[:3] is equivalent to s[0:3]
  (prints everything from start until 3)
- s[5:] is equivalent to s[5:*(last index + 1)*]
  (prints everything from 5)
- s[:] is equivalent to s[0:*(last index + 1)*]
  (prints everything)
- s[::2] is equivalent to s[0:*(last index + 1)*:2]
  (prints everything by steps of 2)

(last index + 1) is equivalent to *len(s)*

Sequence operations

Slicing (1): Positive indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                          [0] [1] [2] [3] [4]
print(mylist)





#                          [0] [1] [2] [3] [4]
```

Sequence operations

Slicing (1): Positive indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                           [0] [1] [2] [3] [4]
print(mylist)            # 42, 37, 42, 66, 15
print(mylist[:])






#                           [0] [1] [2] [3] [4]
```

Sequence operations

Slicing (1): Positive indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                         [0] [1] [2] [3] [4]
print(mylist)           # 42, 37, 42, 66, 15
print(mylist[:])        # 42, 37, 42, 66, 15
print(mylist[1:4])




#                         [0] [1] [2] [3] [4]
```

Sequence operations

Slicing (1): Positive indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                           [0] [1] [2] [3] [4]
print(mylist)             # 42, 37, 42, 66, 15
print(mylist[:])          # 42, 37, 42, 66, 15
print(mylist[1:4])        #     37, 42, 66
print(mylist[2:])




#                           [0] [1] [2] [3] [4]
```

Sequence operations

Slicing (1): Positive indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                          [0] [1] [2] [3] [4]
print(mylist)           # 42, 37, 42, 66, 15
print(mylist[:])        # 42, 37, 42, 66, 15
print(mylist[1:4])      #     37, 42, 66
print(mylist[2:])       #         42, 66, 15
print(mylist[:3])



#                          [0] [1] [2] [3] [4]
```

Sequence operations

Slicing (1): Positive indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                          [0] [1] [2] [3] [4]
print(mylist)            # 42, 37, 42, 66, 15
print(mylist[:])         # 42, 37, 42, 66, 15
print(mylist[1:4])       #     37, 42, 66
print(mylist[2:])        #         42, 66, 15
print(mylist[:3])        # 42, 37, 42
print(mylist[::2])

#                          [0] [1] [2] [3] [4]
```

Sequence operations

Slicing (1): Positive indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                     [0] [1] [2] [3] [4]
print(mylist)       # 42, 37, 42, 66, 15
print(mylist[:])    # 42, 37, 42, 66, 15
print(mylist[1:4])  #     37, 42, 66
print(mylist[2:])   #         42, 66, 15
print(mylist[:3])   # 42, 37, 42
print(mylist[::2])  # 42,     42,     15
print(mylist[1:5:3])
#                     [0] [1] [2] [3] [4]
```

Sequence operations

Slicing (1): Positive indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                              [0] [1] [2] [3] [4]
print(mylist)              # 42, 37, 42, 66, 15
print(mylist[:])           # 42, 37, 42, 66, 15
print(mylist[1:4])         #     37, 42, 66
print(mylist[2:])          #         42, 66, 15
print(mylist[:3])          # 42, 37, 42
print(mylist[::2])         # 42,     42,     15
print(mylist[1:5:3])       #     37,         15
#                              [0] [1] [2] [3] [4]
```

Sequence operations

Slicing tips (2):

you can go back from the last item thanks to negative values

Sequence operations

Slicing tips (2):

you can go back from the last item thanks to negative values

- s[-1] is equivalent to s[*(last index + 1)*] *(last item)*

Sequence operations

Slicing tips (2):

you can go back from the last item thanks to negative values

- s[-1] is equivalent to s[*(last index + 1)*] *(last item)*
- s[-4:] prints the last 4 items

Sequence operations

Slicing tips (2):

you can go back from the last item thanks to negative values

- s[-1] is equivalent to s[*(last index + 1)*] *(last item)*
- s[-4:] prints the last 4 items
- s[:-1] prints all of the items, except the last one

Sequence operations

Slicing tips (2):

you can go back from the last item thanks to negative values

- s[-1] is equivalent to s[*(last index + 1)*] *(last item)*
- s[-4:] prints the last 4 items
- s[:-1] prints all of the items, except the last one
- s[-4:-1] prints the last 4 items, except the last one *(stop at -1)*

Sequence operations

Slicing tips (2):

you can go back from the last item thanks to negative values

- s[-1] is equivalent to s[*(last index + 1)*] *(last item)*
- s[-4:] prints the last 4 items
- s[:-1] prints all of the items, except the last one
- s[-4:-1] prints the last 4 items, except the last one *(stop at -1)*
- s[2:-1] prints all the items from index 2, except the last one

Sequence operations

Slicing tips (2):

you can go back from the last item thanks to negative values

- s[-1] is equivalent to s[*(last index + 1)*] *(last item)*
- s[-4:] prints the last 4 items
- s[:-1] prints all of the items, except the last one
- s[-4:-1] prints the last 4 items, except the last one *(stop at -1)*
- s[2:-1] prints all the items from index 2, except the last one
- s[-4:3] prints all the items from index -4 to index 2 *(stop at 3)*

Sequence operations

Slicing tips (2):

you can go back from the last item thanks to negative values

- s[-1] is equivalent to s[*(last index + 1)*] *(last item)*
- s[-4:] prints the last 4 items
- s[:-1] prints all of the items, except the last one
- s[-4:-1] prints the last 4 items, except the last one *(stop at -1)*
- s[2:-1] prints all the items from index 2, except the last one
- s[-4:3] prints all the items from index -4 to index 2 *(stop at 3)*

don't forget that the second index is not selected/it is the limit

Sequence operations

Slicing (2): Negative indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                           [0] [1] [2] [3] [4]
print(mylist[:])         # 42, 37, 42, 66, 15




#                           [0] [1] [2] [3] [4]
```

Sequence operations

Slicing (2): Negative indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                      [0] [1] [2] [3] [4]
print(mylist[:])     # 42, 37, 42, 66, 15
print(mylist[-1])




#                      [0] [1] [2] [3] [4]
```

Sequence operations

Slicing (2): Negative indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                      [0] [1] [2] [3] [4]
print(mylist[:])     # 42, 37, 42, 66, 15
print(mylist[-1])    #                 15
print(mylist[:-1])



#                     [-5][-4][-3][-2][-1]
```

Sequence operations

Slicing (2): Negative indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                      [0] [1] [2] [3] [4]
print(mylist[:])      # 42, 37, 42, 66, 15
print(mylist[-1])     #                 15
print(mylist[:-1])    # 42, 37, 42, 66
print(mylist[2:-1])


#                     [-5][-4][-3][-2][-1]
```

Sequence operations

Slicing (2): Negative indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                      [0] [1] [2] [3] [4]
print(mylist[:])      # 42, 37, 42, 66, 15
print(mylist[-1])     #                 15
print(mylist[:-1])    # 42, 37, 42, 66
print(mylist[2:-1])   #         42, 66
print(mylist[-4:-1])

#                     [-5][-4][-3][-2][-1]
```

Sequence operations

Slicing (2): Negative indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                      [0] [1] [2] [3] [4]
print(mylist[:])      # 42, 37, 42, 66, 15
print(mylist[-1])     #                 15
print(mylist[:-1])    # 42, 37, 42, 66
print(mylist[2:-1])   #         42, 66
print(mylist[-4:-1])  #     37, 42, 66
print(mylist[-4:4])
#                     [-5][-4][-3][-2][-1]
```

Sequence operations

Slicing (2): Negative indexes

```
mylist = [ 42, 37, 42, 66, 15 ]
#                        [0] [1] [2] [3] [4]
print(mylist[:])       # 42, 37, 42, 66, 15
print(mylist[-1])      #                 15
print(mylist[:-1])     # 42, 37, 42, 66
print(mylist[2:-1])    #         42, 66
print(mylist[-4:-1])   #     37, 42, 66
print(mylist[-4:4])    #     37, 42, 66
#                       [-5][-4][-3][-2][-1]
```

Sequence operations

Beware of **references**!

Sequence operations

Beware of **references**!

Python does not copy everything

Sequence operations

Beware of **references**!

Python does not copy everything
It "*creates bindings between a target and an object*"

Sequence operations

Beware of **references**!

Python does not copy everything
It "*creates bindings between a target and an object*"

Modifying a sequence will change the assigned values in other sequences where it is referenced

Sequence operations

`id()` is a builtin that gives the identifier of an object

Sequence operations

`id()` is a builtin that gives the identifier of an object
(in a specific context, it gives the memory address of the object)

Sequence operations

`id()` is a builtin that gives the identifier of an object
(in a specific context, it gives the memory address of the object)

Sequence operations

`id()` is a builtin that gives the identifier of an object
(in a specific context, it gives the memory address of the object)

```
L1 = [ 42, 37, 42 ]       #  [0] [1] [2] [3] [4]
L2 = L1                   #  42, 37, 42








                          #  [0] [1] [2] [3] [4]
```

Sequence operations

`id()` is a builtin that gives the identifier of an object
(in a specific context, it gives the memory address of the object)

```
L1 = [ 42, 37, 42 ]      #  [0] [1] [2] [3] [4]
L2 = L1                  #  42, 37, 42
print(id(L1))
print(id(L2))
print(L1 == L2)
print(id(L1) == id(L2))



                         #  [0] [1] [2] [3] [4]
```

Sequence operations

id() is a builtin that gives the identifier of an object
(in a specific context, it gives the memory address of the object)

```
L1 = [ 42, 37, 42 ]      #  [0] [1] [2] [3] [4]
L2 = L1                  #  42, 37, 42
print(id(L1))
print(id(L2))
print(L1 == L2)          #  True
print(id(L1) == id(L2)) #  True



                         #  [0] [1] [2] [3] [4]
```

Sequence operations

`id()` is a builtin that gives the identifier of an object
(in a specific context, it gives the memory address of the object)

```
L1 = [ 42, 37, 42 ]       #  [0] [1] [2] [3] [4]
L2 = L1                   #  42, 37, 42
print(id(L1))
print(id(L2))
print(L1 == L2)           #  True
print(id(L1) == id(L2))  #  True
L2[0] = 88
print(L1)
print(L2)
                          #  [0] [1] [2] [3] [4]
```

Sequence operations

`id()` is a builtin that gives the identifier of an object
(in a specific context, it gives the memory address of the object)

```
L1 = [ 42, 37, 42 ]      #  [0] [1] [2] [3] [4]
L2 = L1                  #  42, 37, 42
print(id(L1))
print(id(L2))
print(L1 == L2)          #  True
print(id(L1) == id(L2))  #  True
L2[0] = 88
print(L1)                #  88, 37, 42
print(L2)                #  88, 37, 42
                         #  [0] [1] [2] [3] [4]
```

Sequence operations

Lists are not copied, only their references are used

```
L1 = [ 42, 37, 42 ]      #  [0] [1] [2] [3] [4]
L2 = L1                  #  42, 37, 42
print(id(L1))
print(id(L2))
print(L1 == L2)          #  True
print(id(L1) == id(L2))  #  True
L2[0] = 88
print(L1)                #  88, 37, 42
print(L2)                #  88, 37, 42
                         #  [0] [1] [2] [3] [4]
```

Sequence operations

```
L1 = [ 2, 7, [4, 3] ]    #  [0] [1] [2] [3] [4]











                         #  [0] [1] [2] [3] [4]
```

Sequence operations

```
L1 = [ 2, 7, [4, 3] ]    #  [0] [1] [2] [3] [4]
L2 = L1.copy()           #   2   7 [4,3]
print(L1 == L2)
print(id(L1) == id(L2))




                         #  [0] [1] [2] [3] [4]
```

Sequence operations

```
L1 = [ 2, 7, [4, 3] ]    #  [0] [1] [2] [3] [4]
L2 = L1.copy()           #   2   7 [4,3]
print(L1 == L2)          #  True
print(id(L1) == id(L2))  #  False
L2[0] = 8





                         #  [0] [1] [2] [3] [4]
```

Sequence operations

```
L1 = [ 2, 7, [4, 3] ]   #  [0] [1] [2] [3] [4]
L2 = L1.copy()          #   2   7 [4,3]
print(L1 == L2)         #  True
print(id(L1) == id(L2)) #  False
L2[0] = 8
print(L1)
print(L2)



                        #  [0] [1] [2] [3] [4]
```

Sequence operations

```
L1 = [ 2, 7, [4, 3] ]     #  [0] [1] [2] [3] [4]
L2 = L1.copy()            #   2   7 [4,3]
print(L1 == L2)           #  True
print(id(L1) == id(L2))   #  False
L2[0] = 8
print(L1)                 #   2   7 [4,3]
print(L2)                 #   8   7 [4,3]
L2[2][0] = 9
print(L1)
print(L2)
                          #  [0] [1] [2] [3] [4]
```

Sequence operations

```
L1 = [ 2, 7, [4, 3] ]    #  [0] [1] [2] [3] [4]
L2 = L1.copy()           #   2   7 [4,3]
print(L1 == L2)          #  True
print(id(L1) == id(L2))  #  False
L2[0] = 8
print(L1)                #   2   7 [4,3]
print(L2)                #   8   7 [4,3]
L2[2][0] = 9
print(L1)                #   2   7 [9,3]
print(L2)                #   8   7 [9,3]
                         #  [0] [1] [2] [3] [4]
```

Sequence operations

Module *copy*

Sequence operations

Module *copy*

- **Shallow copy**: creates a new sequence and inserts references to each contained object of the original sequence

Sequence operations

Module *copy*

- **Shallow copy**: creates a new sequence and inserts references to each contained object of the original sequence
- **Deep copy**: creates a new sequence and recursively inserts copies of each object of the original sequence

Sequence operations

Module *copy*

- **Shallow copy**: creates a new sequence and inserts references to each contained object of the original sequence
- **Deep copy**: creates a new sequence and recursively inserts copies of each object of the original sequence

`NewList1 = list(MyList)` works as a shallow copy

Sequence operations

Module *copy*

- **Shallow copy**: creates a new sequence and inserts references to each contained object of the original sequence
- **Deep copy**: creates a new sequence and recursively inserts copies of each object of the original sequence

`NewList1 = list(MyList)` works as a shallow copy
`NewList2 = MyList[:]` works as a shallow copy

Sequence operations

Module *copy*

- **Shallow copy**: creates a new sequence and inserts references to each contained object of the original sequence
- **Deep copy**: creates a new sequence and recursively inserts copies of each object of the original sequence

`NewList1 = list(MyList)` works as a shallow copy
`NewList2 = MyList[:]` works as a shallow copy
`NewList3 = MyList.copy()` works as a shallow copy

Sequence operations

Major difficulties concern containers containing containers

```
import copy

L1 = [ [1,1], [2,2], [3,3] ]
```

Sequence operations

Major difficulties concern containers containing containers

```
import copy

L1 = [ [1,1], [2,2], [3,3] ]
L2 = copy.copy(L1)      # shallow copy
L3 = copy.deepcopy(L1) # deep copy
```

Sequence operations

Major difficulties concern containers containing containers

```
import copy

L1 = [ [1,1], [2,2], [3,3] ]
L2 = copy.copy(L1)      # shallow copy
L3 = copy.deepcopy(L1) # deep copy
L1[1][1] = 9
print(L2)
print(L3)
```

Sequence operations

Major difficulties concern containers containing containers

```
import copy

L1 = [ [1,1], [2,2], [3,3] ]
L2 = copy.copy(L1)      # shallow copy
L3 = copy.deepcopy(L1) # deep copy
L1[1][1] = 9
print(L2)    # [ [1,1], [2,9], [3,3] ]
print(L3)    # [ [1,1], [2,2], [3,3] ]
```

Sequence operations

Major difficulties concern containers containing containers

```
import copy

L1 = [ [1,1], [2,2], [3,3] ]
L2 = copy.copy(L1)      # shallow copy
L3 = copy.deepcopy(L1) # deep copy
L1[1][1] = 9
print(L2)    # [ [1,1], [2,9], [3,3] ]
print(L3)    # [ [1,1], [2,2], [3,3] ]
L1[1] = [0,0]
print(L2)
```

Sequence operations

Major difficulties concern containers containing containers

```
import copy

L1 = [ [1,1], [2,2], [3,3] ]
L2 = copy.copy(L1)      # shallow copy
L3 = copy.deepcopy(L1) # deep copy
L1[1][1] = 9
print(L2)    # [ [1,1], [2,9], [3,3] ]
print(L3)    # [ [1,1], [2,2], [3,3] ]
L1[1] = [0,0]
print(L2)    # [ [1,1], [2,9], [3,3] ]
```

7: Python's Main Concepts

1 Numeric types

2 Sequence types
- Sequence operations
- Lists
- Tuples
- Ranges

3 Mapping types

4 Other types

Lists

- mutable (can be modified after creation)

Lists

- mutable (can be modified after creation)
  - *beware of shallow and deep copies*

Lists

- mutable (can be modified after creation)
  - *beware of shallow and deep copies*
- ordered (only the functions of reorganization change the order)

Lists

- mutable (can be modified after creation)
  - *beware of shallow and deep copies*
- ordered (only the functions of reorganization change the order)
- same value can be recorded multiple times

Lists

- mutable (can be modified after creation)
  - *beware of shallow and deep copies*
- ordered (only the functions of reorganization change the order)
- same value can be recorded multiple times
- different types of items are allowed in the same list

Lists

Creation of a list:

```
vide = []
```

Lists

Creation of a list:

```
vide = []
initialisee1 = [ 42 ]
initialisee2 = [ 42, 1337, 666 ]
```

Lists

Creation of a list:

```
vide = []
initialisee1 = [ 42 ]
initialisee2 = [ 42, 1337, 666 ]
comprehension = [ x for x in range(0, 10) ]
```

Lists

Creation of a list:

```
vide = []
initialisee1 = [ 42 ]
initialisee2 = [ 42, 1337, 666 ]
comprehension = [ x for x in range(0, 10) ]
constructeurVide = list()
constructeur1 = list("abc")
constructeur2 = list( (58, "abc", 58) )
```

Lists

Creation of a list:

```
vide = []
initialisee1 = [ 42 ]
initialisee2 = [ 42, 1337, 666 ]
comprehension = [ x for x in range(0, 10) ]
constructeurVide = list()
constructeur1 = list("abc")
constructeur2 = list( (58, "abc", 58) )
```

- list initialized by comprehension or with a constructor can use any *iterable* object

Lists operations

Operation	Result
list.append(x)	add an item x to the end of the list
list.insert(pos, x)	insert an item x at position *pos*
	(items are pushed back/+1 applied to indexes)
list.extend(L)	concatenates lists : L added at the end of *list*
	(any iterable objects can be concatenated)
list.pop(pos)	remove the item at position *pos*
	(if no parameter given, it removes the last one)
list.remove(x)	remove the first item x found in the list
list.copy()	copy the list
list.reverse()	reverse the order of the list
list.sort()	sort alphabetically the list
	(add the parameter $reverse = True$ to reverse)

x is an arbitrary object, *pos* is an integer, and L is a list

Lists

```
mylist = [ 42, 37, 42 ] # [0] [1] [2] [3] [4]
mylist.append(66)
```

Lists

```
mylist = [ 42, 37, 42 ] # [0] [1] [2] [3] [4]
mylist.append(66)       # 42, 37, 42, 66
mylist.insert(1, 99)
```

Lists

```
mylist = [ 42, 37, 42 ] # [0] [1] [2] [3] [4]
mylist.append(66)       # 42, 37, 42, 66
mylist.insert(1, 99)    # 42, 99, 37, 42, 66
mylist.remove(42)
```

Lists

```
mylist = [ 42, 37, 42 ] # [0] [1] [2] [3] [4]
mylist.append(66)       # 42, 37, 42, 66
mylist.insert(1, 99)    # 42, 99, 37, 42, 66
mylist.remove(42)       # 99, 37, 42, 66
mylist.reverse()
```

Lists

```
mylist = [ 42, 37, 42 ] # [0] [1] [2] [3] [4]
mylist.append(66)       # 42, 37, 42, 66
mylist.insert(1, 99)    # 42, 99, 37, 42, 66
mylist.remove(42)       # 99, 37, 42, 66
mylist.reverse()        # 66, 42, 37, 99
mylist.sort()
```

Lists

```
mylist = [ 42, 37, 42 ] # [0] [1] [2] [3] [4]
mylist.append(66)        # 42, 37, 42, 66
mylist.insert(1, 99)     # 42, 99, 37, 42, 66
mylist.remove(42)        # 99, 37, 42, 66
mylist.reverse()         # 66, 42, 37, 99
mylist.sort()            # 37, 42, 66, 99
mylist.pop()
```

Lists

```
mylist = [ 42, 37, 42 ] # [0] [1] [2] [3] [4]
mylist.append(66)        # 42, 37, 42, 66
mylist.insert(1, 99)     # 42, 99, 37, 42, 66
mylist.remove(42)        # 99, 37, 42, 66
mylist.reverse()         # 66, 42, 37, 99
mylist.sort()            # 37, 42, 66, 99
mylist.pop()             # 37, 42, 66
mylist.pop(1)
```

Lists

```
mylist = [ 42, 37, 42 ] # [0] [1] [2] [3] [4]
mylist.append(66)        # 42, 37, 42, 66
mylist.insert(1, 99)     # 42, 99, 37, 42, 66
mylist.remove(42)        # 99, 37, 42, 66
mylist.reverse()         # 66, 42, 37, 99
mylist.sort()            # 37, 42, 66, 99
mylist.pop()             # 37, 42, 66
mylist.pop(1)            # 37, 66
l2 = mylist.copy()
l2.append(88)
mylist.extend(l2)
```

Lists

```
mylist = [ 42, 37, 42 ] # [0] [1] [2] [3] [4]
mylist.append(66)       # 42, 37, 42, 66
mylist.insert(1, 99)    # 42, 99, 37, 42, 66
mylist.remove(42)       # 99, 37, 42, 66
mylist.reverse()        # 66, 42, 37, 99
mylist.sort()           # 37, 42, 66, 99
mylist.pop()            # 37, 42, 66
mylist.pop(1)           # 37, 66
l2 = mylist.copy()  # 37, 66
l2.append(88)       # 37, 66, 88
mylist.extend(l2)       # 37, 66, 37, 66, 88
```

7: Python's Main Concepts

1 Numeric types

2 Sequence types
- Sequence operations
- Lists
- Tuples
- Ranges

3 Mapping types

4 Other types

Tuples

- immutable (when created, cannot be modified)

Tuples

- immutable (when created, cannot be modified)
- ordered

Tuples

- immutable (when created, cannot be modified)
- ordered
- same value can be recorded multiple times

Tuples

- immutable (when created, cannot be modified)
- ordered
- same value can be recorded multiple times
- different types of items are allowed in the same tuple

Tuples

Creation of a tuple:

```
vide = ()
```

Tuples

Creation of a tuple:

```
vide = ()
initialisee1 = ( 42, )
initialisee2 = ( 42, 1337, 666 )
```

Tuples

Creation of a tuple:

```
vide = ()
initialisee1 = ( 42, )
initialisee2 = ( 42, 1337, 666 )
constructeurVide = tuple()
constructeur1 = tuple("abc",)
constructeur2 = tuple( (58, "abc", 58) )
```

Tuples

Creation of a tuple:

```
vide = ()
initialisee1 = ( 42, )
initialisee2 = ( 42, 1337, 666 )
constructeurVide = tuple()
constructeur1 = tuple("abc",)
constructeur2 = tuple( (58, "abc", 58) )
```

- Tuple with only 1 element must have a comma (,)

Tuples

Creation of a tuple:

```
vide = ()
initialisee1 = ( 42, )
initialisee2 = ( 42, 1337, 666 )
constructeurVide = tuple()
constructeur1 = tuple("abc",)
constructeur2 = tuple( (58, "abc", 58) )
```

- Tuple with only 1 element must have a comma (,)
- Tuples can be initialized with any iterable object

Tuples

Creating a tuple requires commas OR the iterable property

Tuples

Creating a tuple requires commas OR the iterable property
Parenthesis are mandatory when a tuple is used as an argument or when creating an empty tuple

Tuples

Creating a tuple requires commas OR the iterable property
Parenthesis are mandatory when a tuple is used as an argument or when creating an empty tuple

Tuples

Creating a tuple requires commas OR the iterable property

Parenthesis are mandatory when a tuple is used as an argument or when creating an empty tuple

```
my_tuple = tuple('a', 2)      # ('a', 2)
```

Tuples

Creating a tuple requires commas OR the iterable property
Parenthesis are mandatory when a tuple is used as an argument or when creating an empty tuple

```
my_tuple = tuple('a', 2)        # ('a', 2)
nb_tuple = tuple([1, 2, 3])
```

Tuples

Creating a tuple requires commas OR the iterable property
Parenthesis are mandatory when a tuple is used as an argument or when creating an empty tuple

```
my_tuple = tuple('a', 2)        # ('a', 2)
nb_tuple = tuple([1, 2, 3])  # (1, 2, 3)
str_tuple = tuple('abc')
```

Tuples

Creating a tuple requires commas OR the iterable property
Parenthesis are mandatory when a tuple is used as an argument or when creating an empty tuple

```
my_tuple = tuple('a', 2)       # ('a', 2)
nb_tuple = tuple([1, 2, 3])  # (1, 2, 3)
str_tuple = tuple('abc')      # ('a', 'b', 'c')
f('a', 'b', 'c')
f( ('a','b','c') )
```

Tuples

Creating a tuple requires commas OR the iterable property
Parenthesis are mandatory when a tuple is used as an argument or when creating an empty tuple

```
my_tuple = tuple('a', 2)       # ('a', 2)
nb_tuple = tuple([1, 2, 3])  # (1, 2, 3)
str_tuple = tuple('abc')       # ('a', 'b', 'c')
f('a', 'b', 'c')    # Calls f with 3 arguments
f( ('a','b','c') ) # Calls f with 1 argument
```

7: Python's Main Concepts

1. Numeric types
2. Sequence types
   - Sequence operations
   - Lists
   - Tuples
   - Ranges
3. Mapping types
4. Other types

Ranges

- immutable (when created, cannot be modified)

Ranges

- immutable (when created, cannot be modified)
- ordered

Ranges

- immutable (when created, cannot be modified)
- ordered
- same value can be recorded multiple times

Ranges

- immutable (when created, cannot be modified)
- ordered
- same value can be recorded multiple times
- only for integers numbers

Ranges

- immutable (when created, cannot be modified)
- ordered
- same value can be recorded multiple times
- only for integers numbers
- always takes the same amount of memory

Ranges

- immutable (when created, cannot be modified)
- ordered
- same value can be recorded multiple times
- only for integers numbers
- always takes the same amount of memory
  - only 3 parameters are used (**`start`**, **`stop`**, **`step`**)

Ranges

Ranges are defined at least by the *stop* boundary

Ranges

Ranges are defined at least by the *stop* boundary
Start defines the first value to take, by default it is 0
Step defines the step between each value, by default it is 1

Ranges

Ranges are defined at least by the *stop* boundary
Start defines the first value to take, by default it is 0
Step defines the step between each value, by default it is 1

Ranges

Ranges are defined at least by the *stop* boundary
Start defines the first value to take, by default it is 0
Step defines the step between each value, by default it is 1

```
list(range(6))
```

Ranges

Ranges are defined at least by the *stop* boundary
Start defines the first value to take, by default it is 0
Step defines the step between each value, by default it is 1

```
list(range(6))          # 0 1 2 3 4 5
list(range(1, 6))
```

Ranges

Ranges are defined at least by the *stop* boundary
Start defines the first value to take, by default it is 0
Step defines the step between each value, by default it is 1

```
list(range(6))          # 0 1 2 3 4 5
list(range(1, 6))       #   1 2 3 4 5
list(range(0, 6, 2))
```

Ranges

Ranges are defined at least by the *stop* boundary
Start defines the first value to take, by default it is 0
Step defines the step between each value, by default it is 1

```
list(range(6))          # 0 1 2 3 4 5
list(range(1, 6))       #   1 2 3 4 5
list(range(0, 6, 2))    # 0   2   4
```

Ranges

Main range formulae: $r[i] = \text{start} + \text{step} \cdot i$

Ranges

Main range formulae: $r[i] = \text{start} + \text{step} \cdot i$
Constraint: $i \geq 0$ and $r[i] < \text{stop}$

Ranges

Main range formulae: $r[i] = \text{start} + \text{step} \cdot i$
Constraint: $i \geq 0$ and $r[i] < \text{stop}$
Ranges can go backward with negative steps!

Ranges

Main range formulae: $r[i] = \text{start} + \text{step} \cdot i$
Constraint: $i \geq 0$ and $r[i] < \text{stop}$
Ranges can go backward with negative steps!
Constraint (negative step): $i \geq 0$ and $r[i] > \text{stop}$

Ranges

Main range formulae: $r[i] = \text{start} + \text{step} \cdot i$

Constraint: $i \geq 0$ and $r[i] < \text{stop}$

Ranges can go backward with negative steps!

Constraint (negative step): $i \geq 0$ and $r[i] > \text{stop}$

Ranges

Main range formulae: $r[i] = \text{start} + \text{step} \cdot i$
Constraint: $i \geq 0$ and $r[i] < \text{stop}$
Ranges can go backward with negative steps!
Constraint (negative step): $i \geq 0$ and $r[i] > \text{stop}$

```
list(range(-5, 0))
```

Ranges

Main range formulae: $r[i] = \text{start} + \text{step} \cdot i$
Constraint: $i \geq 0$ and $r[i] < \text{stop}$
Ranges can go backward with negative steps!
Constraint (negative step): $i \geq 0$ and $r[i] > \text{stop}$

```
list(range(-5, 0))      # -5 -4 -3 -2 -1
list(range(0, -5, -1))
```

Ranges

Main range formulae: $r[i] = \text{start} + \text{step} \cdot i$
Constraint: $i \geq 0$ and $r[i] < \text{stop}$
Ranges can go backward with negative steps!
Constraint (negative step): $i \geq 0$ and $r[i] > \text{stop}$

```
list(range(-5, 0))      # -5 -4 -3 -2 -1
list(range(0, -5, -1))  #  0 -1 -2 -3 -4
list(range(5,  0, -1))
```

Ranges

Main range formulae: $r[i] = \text{start} + \text{step} \cdot i$

Constraint: $i \geq 0$ and $r[i] < \text{stop}$

Ranges can go backward with negative steps!

Constraint (negative step): $i \geq 0$ and $r[i] > \text{stop}$

```
list(range(-5, 0))       # -5 -4 -3 -2 -1
list(range(0, -5, -1))  #  0 -1 -2 -3 -4
list(range(5,  0, -1))  #  5  4  3  2  1
list(range(0))
```

Ranges

Main range formulae: $r[i] = \text{start} + \text{step} \cdot i$
Constraint: $i \geq 0$ and $r[i] < \text{stop}$
Ranges can go backward with negative steps!
Constraint (negative step): $i \geq 0$ and $r[i] > \text{stop}$

```
list(range(-5, 0))      # -5 -4 -3 -2 -1
list(range(0, -5, -1)) #  0 -1 -2 -3 -4
list(range(5,  0, -1)) #  5  4  3  2  1
list(range(0))          # []  step+  0 == stop
list(range(1,  0))
```

Ranges

Main range formulae: $r[i] = \text{start} + \text{step} \cdot i$

Constraint: $i \geq 0$ and $r[i] < \text{stop}$

Ranges can go backward with negative steps!

Constraint (negative step): $i \geq 0$ and $r[i] > \text{stop}$

```
list(range(-5, 0))      # -5 -4 -3 -2 -1
list(range(0, -5, -1))  #  0 -1 -2 -3 -4
list(range(5,  0, -1))  #  5  4  3  2  1
list(range(0))          # []  step+  0 == stop
list(range(1,  0))      # []  step+  1 > stop
list(range(0, -1))
```

Ranges

Main range formulae: $r[i] = \text{start} + \text{step} \cdot i$

Constraint: $i \geq 0$ and $r[i] < \text{stop}$

Ranges can go backward with negative steps!

Constraint (negative step): $i \geq 0$ and $r[i] > \text{stop}$

```
list(range(-5, 0))       # -5 -4 -3 -2 -1
list(range(0, -5, -1))   #  0 -1 -2 -3 -4
list(range(5,  0, -1))   #  5  4  3  2  1
list(range(0))           # []  step+  0 == stop
list(range(1,  0))       # []  step+  1 > stop
list(range(0, -1))       # []  step+  0 > stop
```

Ranges

Ranges are a type, and can be compared with `==` and `!=`

Ranges

Ranges are a type, and can be compared with **`==`** and **`!=`**
They are equal if they represent the same sequence of values

Ranges

Ranges are a type, and can be compared with **`==`** and **`!=`**
They are equal if they represent the same sequence of values
(**`start`**, **`stop`**, and **`step`** might be differents)

Ranges

Ranges are a type, and can be compared with **==** and **!=**

They are equal if they represent the same sequence of values (**start**, **stop**, and **step** might be differents)

```
r0 = range(0)
r1 = range(0, 5, 1)
r2 = range(0, 3, 2)
r3 = range(0, 4, 2)
```

Ranges

Ranges are a type, and can be compared with **==** and **!=**
They are equal if they represent the same sequence of values
(**start**, **stop**, and **step** might be differents)

```
r0 = range(0)           # []
r1 = range(0, 5, 1)     # 0 1 2 3 4
r2 = range(0, 3, 2)     # 0     3
r3 = range(0, 4, 2)     # 0     3
```

Ranges

Ranges are a type, and can be compared with **==** and **!=**
They are equal if they represent the same sequence of values
(**start**, **stop**, and **step** might be differents)

```
r0 = range(0)             # []
r1 = range(0, 5, 1)       # 0 1 2 3 4
r2 = range(0, 3, 2)       # 0     3
r3 = range(0, 4, 2)       # 0     3
print(r0 == r1)
```

Ranges

Ranges are a type, and can be compared with **==** and **!=**
They are equal if they represent the same sequence of values (**start**, **stop**, and **step** might be differents)

```
r0 = range(0)            # []
r1 = range(0, 5, 1)      # 0 1 2 3 4
r2 = range(0, 3, 2)      # 0     3
r3 = range(0, 4, 2)      # 0     3
print(r0 == r1)     # False
print(r1 == r2)
```

Ranges

Ranges are a type, and can be compared with **==** and **!=**
They are equal if they represent the same sequence of values
(**start**, **stop**, and **step** might be differents)

```
r0 = range(0)             # []
r1 = range(0, 5, 1)       # 0 1 2 3 4
r2 = range(0, 3, 2)       # 0     3
r3 = range(0, 4, 2)       # 0     3
print(r0 == r1)     # False
print(r1 == r2)     # False
print(r2 == r3)
```

Ranges

Ranges are a type, and can be compared with **==** and **!=**
They are equal if they represent the same sequence of values
(**start**, **stop**, and **step** might be differents)

```
r0 = range(0)              # []
r1 = range(0, 5, 1)        # 0 1 2 3 4
r2 = range(0, 3, 2)        # 0     3
r3 = range(0, 4, 2)        # 0     3
print(r0 == r1)     # False
print(r1 == r2)     # False
print(r2 == r3)     # True
```

7: Python's Main Concepts

1 Numeric types

2 Sequence types

3 Mapping types
- Dictionnaries

4 Other types

Mapping types

- dict *(dictionnary)*

Mapping types

- dict *(dictionnary)*
- "*maps hashable values to arbitrary objects*"

Mapping types

- dict *(dictionnary)*
- "*maps hashable values to arbitrary objects*"
  - keys = hashable values

Mapping types

- dict *(dictionnary)*
- "*maps hashable values to arbitrary objects*"
  - keys = hashable values
  - keys must be hashable (a list can't be a key)

Mapping types

- dict *(dictionnary)*
- "*maps hashable values to arbitrary objects*"
  - keys = hashable values
  - keys must be hashable (a list can't be a key)
  - if two values are equal, they refer to the same key

Mapping types

- dict *(dictionnary)*
- "*maps hashable values to arbitrary objects*"
  - keys = hashable values
  - keys must be hashable (a list can't be a key)
  - if two values are equal, they refer to the same key
- mutable

7: Python's Main Concepts

1. Numeric types
2. Sequence types
3. Mapping types
   - Dictionnaries
4. Other types

Dictionnaries

Creation of a dict:

Dictionnaries

Creation of a dict:

```
vide = dict()
dict1 = {"one": 1, "two": 2, "three": 3}
```

Dictionnaries

Creation of a dict:

```
vide = dict()
dict1 = {"one": 1, "two": 2, "three": 3}
dict2 = dict(one=1, two=2, three=3)
```

Dictionnaries

Creation of a dict:

```
vide = dict()
dict1 = {"one": 1, "two": 2, "three": 3}
dict2 = dict(one=1, two=2, three=3)
dict3 = dict({'three': 3, 'one': 1, 'two': 2})
```

Dictionnaries

Creation of a dict:

```
vide = dict()
dict1 = {"one": 1, "two": 2, "three": 3}
dict2 = dict(one=1, two=2, three=3)
dict3 = dict({'three': 3, 'one': 1, 'two': 2})
dict4 = dict({'one': 1, 'three': 3}, two=2)
```

Dictionnaries

Creation of a dict:

```
vide = dict()
dict1 = {"one": 1, "two": 2, "three": 3}
dict2 = dict(one=1, two=2, three=3)
dict3 = dict({'three': 3, 'one': 1, 'two': 2})
dict4 = dict({'one': 1, 'three': 3}, two=2)
dict5 = dict([('two',2),('one',1), ('three',3)])
```

Dicts operations

Operation	Result
list(d)	return a list of all the keys of *d*
len(d)	number of items in *d*
d.get(key, default)	like **`d[key]`**, but return *default* if *key* not found
d.keys()	return the dictionnary's keys
iter(d)	return an iterator over the keys of *d*
d[key] = value	set *value* at key *key*
del d[key]	Remove the value at **`d[key]`** (Raise a KeyError if *key* not found)
d.pop(key)	remove *key* and return the associated value
d.clear()	remove all items from the dictionnary
d.copy()	shallow copy of the dictionnary *d*

Dicts operations

```
dict1 = {"one": 1, "two": 2, "three": 3}
dict1["one"]
dict1["one"] + dict1["two"]
len(dict1)
```

Dicts operations

```
dict1 = {"one": 1, "two": 2, "three": 3}
dict1["one"]            # 1
dict1["one"] + dict1["two"] # 3
len(dict1)              # 3
list(dict1)
```

Dicts operations

```
dict1 = {"one": 1, "two": 2, "three": 3}
dict1["one"]          # 1
dict1["one"] + dict1["two"] # 3
len(dict1)            # 3
list(dict1)           # ['one','two','three']
```

Dicts operations

```
dict1 = {"one": 1, "two": 2, "three": 3}
dict1["one"]          # 1
dict1["one"] + dict1["two"] # 3
len(dict1)            # 3
list(dict1)           # ['one','two','three']
dict1.get("one")
dict1.get("A", 42)
```

Dicts operations

```
dict1 = {"one": 1, "two": 2, "three": 3}
dict1["one"]           # 1
dict1["one"] + dict1["two"] # 3
len(dict1)             # 3
list(dict1)            # ['one','two','three']
dict1.get("one")       # 1
dict1.get("A", 42) # 42
dict1["A"] = 8
dict1.get("A", 42)
```

Dicts operations

```
dict1 = {"one": 1, "two": 2, "three": 3}
dict1["one"]          # 1
dict1["one"] + dict1["two"] # 3
len(dict1)            # 3
list(dict1)           # ['one','two','three']
dict1.get("one")      # 1
dict1.get("A", 42) # 42
dict1["A"] = 8
dict1.get("A", 42) # 8
```

Dicts operations

Operation	**Result**
d | e	new dict with merged keys and values from *d* and *e* (values of *e* erase those from *d* if same key)
d |= e	like **`d	e`** except that *d* is updated (values of *e* erase those from *d* if same key)

Dicts operations

```
dict1 = {"one": 1, "two": 2}
dict2 = {"one": 4, "six": 6}
```

Dicts operations

```
dict1 = {"one": 1, "two": 2}
dict2 = {"one": 4, "six": 6}
dict1["one"] + dict2["one"]
```

Dicts operations

```
dict1 = {"one": 1, "two": 2}
dict2 = {"one": 4, "six": 6}
dict1["one"] + dict2["one"]    # 5
NewD = dict1 | dict2
```

Dicts operations

```
dict1 = {"one": 1, "two": 2}
dict2 = {"one": 4, "six": 6}
dict1["one"] + dict2["one"]    # 5
NewD = dict1 | dict2
NewD["two"]
NewD["six"]
NewD["one"]
```

Dicts operations

```
dict1 = {"one": 1, "two": 2}
dict2 = {"one": 4, "six": 6}
dict1["one"] + dict2["one"]    # 5
NewD = dict1 | dict2
NewD["two"]    # 2
NewD["six"]    # 6
NewD["one"]    # 4
```

7: Python's Main Concepts

1 Numeric types

2 Sequence types

3 Mapping types

4 Other types
- Set and FrozenSet
- Text sequence type

Other Built-in types

- Set (and FrozenSet)
- Text sequence type - **`str`**
- Binary sequence type

More details on types

`https://docs.python.org/3/library/stdtypes.html`

Other Built-in types

- Set (and FrozenSet)
- Text sequence type - **`str`**
- Binary sequence type
  - *(we won't see it)*

More details on types

`https://docs.python.org/3/library/stdtypes.html`

7: Python's Main Concepts

1 Numeric types

2 Sequence types

3 Mapping types

4 Other types
- Set and FrozenSet
- Text sequence type

Set

- mutable

Set

- mutable
  - *FrozenSet are immutables*

Set

- mutable
  - *FrozenSet are immutables*
- unordered

Set

- mutable
  - *FrozenSet are immutables*
- unordered
  - *sequences cannot be used (index, slicing, ...)*

Set

- mutable
  - *FrozenSet are immutables*
- unordered
  - *sequences cannot be used (index, slicing, ...)*
- distinct values only

Set

- mutable
  - *FrozenSet are immutables*
- unordered
  - *sequences cannot be used (index, slicing, ...)*
- distinct values only
- only hashable objects can be added

Set

```
set1 = {'world', 'hello'}
set2 = set( [4, 42, 6] )
set3 = {x for x in 'abracadabra'
         if x not in 'abc'}
set3
```

Set

```
set1 = {'world', 'hello'}  # {'hello', 'world'}
set2 = set( [4, 42, 6] )
set3 = {x for x in 'abracadabra'
        if x not in 'abc'}
set3
```

Set

```
set1 = {'world', 'hello'}  # {'hello', 'world'}
set2 = set( [4, 42, 6] )   # {42, 4, 6}
set3 = {x for x in 'abracadabra'
          if x not in 'abc'}
set3
```

Set

```python
set1 = {'world', 'hello'}  # {'hello', 'world'}
set2 = set( [4, 42, 6] )   # {42, 4, 6}
set3 = {x for x in 'abracadabra'
          if x not in 'abc'}
set3                       # {'d', 'r'}
```

Set

```python
set1 = {'world', 'hello'}  # {'hello', 'world'}
set2 = set( [4, 42, 6] )   # {42, 4, 6}
set3 = {x for x in 'abracadabra'
         if x not in 'abc'}
set3                       # {'d', 'r'}
```

Main usages: membership testing, removing duplicates from a sequence, mathematical operations (union, intersection, ...)

Set operations

Operation	Result
add(e)	add an element to a set
remove(e)	remove an element from a set
len(s)	number of elements in set *s*
x in s	test *x* for membership in *s*
copy()	shallow copy of set
isdisjoint(s2)	test if no element are in common with *s2*
issubset(s2)	test if every element is in the set *s2*
issuperset(s2)	test if every element of *s2* is in the set
	new set with elements:
union(s2)	- from the set and *s2*
intersection(s2)	- common to the set and *s2*
difference(s2)	- in the set that are not in *s2*
symmetric_difference(s2)	- in either the set or *s2* but not both

Set operations

```
s1 = { 1, 2, 3, 4 }
s2 = { 3, 4, 5, 6 }
s3 = { 1, 2 }
s4 = { 1, 2, 3, 4, 5}
```

Set operations

```
s1 = { 1, 2, 3, 4 }
s2 = { 3, 4, 5, 6 }
s3 = { 1, 2 }
s4 = { 1, 2, 3, 4, 5}
s2.isdisjoint(s1)
```

Set operations

```
s1 = { 1, 2, 3, 4 }
s2 = { 3, 4, 5, 6 }
s3 = { 1, 2 }
s4 = { 1, 2, 3, 4, 5}
s2.isdisjoint(s1)       # False
s2.isdisjoint(s3)
```

Set operations

```
s1 = { 1, 2, 3, 4 }
s2 = { 3, 4, 5, 6 }
s3 = { 1, 2 }
s4 = { 1, 2, 3, 4, 5}
s2.isdisjoint(s1)      # False
s2.isdisjoint(s3)      # True
s3.issubset(s1)
```

Set operations

```
s1 = { 1, 2, 3, 4 }
s2 = { 3, 4, 5, 6 }
s3 = { 1, 2 }
s4 = { 1, 2, 3, 4, 5}
s2.isdisjoint(s1)       # False
s2.isdisjoint(s3)       # True
s3.issubset(s1)         # True
s3.issuperset(s1)
```

Set operations

```
s1 = { 1, 2, 3, 4 }
s2 = { 3, 4, 5, 6 }
s3 = { 1, 2 }
s4 = { 1, 2, 3, 4, 5}
s2.isdisjoint(s1)       # False
s2.isdisjoint(s3)       # True
s3.issubset(s1)         # True
s3.issuperset(s1)       # False
s4.issubset(s1)
```

Set operations

```
s1 = { 1, 2, 3, 4 }
s2 = { 3, 4, 5, 6 }
s3 = { 1, 2 }
s4 = { 1, 2, 3, 4, 5}
s2.isdisjoint(s1)       # False
s2.isdisjoint(s3)       # True
s3.issubset(s1)         # True
s3.issuperset(s1)       # False
s4.issubset(s1)         # False
s4.issuperset(s1)
```

Set operations

```
s1 = { 1, 2, 3, 4 }
s2 = { 3, 4, 5, 6 }
s3 = { 1, 2 }
s4 = { 1, 2, 3, 4, 5}
s2.isdisjoint(s1)       # False
s2.isdisjoint(s3)       # True
s3.issubset(s1)         # True
s3.issuperset(s1)       # False
s4.issubset(s1)         # False
s4.issuperset(s1)       # True
```

7: Python's Main Concepts

1 Numeric types

2 Sequence types

3 Mapping types

4 Other types
- Set and FrozenSet
- Text sequence type

Text sequence type

- immutable (when created, cannot be modified)

Text sequence type

- immutable (when created, cannot be modified)
- ordered

Text sequence type

- immutable (when created, cannot be modified)
- ordered
- same value can be recorded multiple times

Text sequence type

- immutable (when created, cannot be modified)
- ordered
- same value can be recorded multiple times
- only **`str`** objets can be used inside

Text sequence type

Three ways to write literals:

- Single quotes: `'Text'`

Text sequence type

Three ways to write literals:

- Single quotes: `'Text'`
  - *(may embed double quotes* `'The "only" Text'` *)*

Text sequence type

Three ways to write literals:

- Single quotes: `'Text'`
  - *(may embed double quotes* `'The "only" Text'` *)*
- Double quotes: `"Text"`

Text sequence type

Three ways to write literals:

- Single quotes: `'Text'`
  - *(may embed double quotes* `'The "only" Text'` *)*
- Double quotes: `"Text"`
  - *(may embed single quotes* `"The 'only' Text"` *)*

Text sequence type

Three ways to write literals:

- Single quotes: `'Text'`
  - *(may embed double quotes* `'The "only" Text'` *)*
- Double quotes: `"Text"`
  - *(may embed single quotes* `"The 'only' Text"` *)*
- Triple quotes: `'''Text'''` or `"""Text"""`

Text sequence type

Three ways to write literals:

- Single quotes: `'Text'`
  - *(may embed double quotes* `'The "only" Text'` *)*
- Double quotes: `"Text"`
  - *(may embed single quotes* `"The 'only' Text"` *)*
- Triple quotes: `'''Text'''` or `"""Text"""`
  - *(triple quotes can be on multiple lines)*

Text sequence type

As strings are immutables, you must:

Text sequence type

As strings are immutables, you must:

- concatenate strings into a new string

Text sequence type

As strings are immutables, you must:

- concatenate strings into a new string
- extract substrings with slices

Text sequence type

As strings are immutables, you must:

- concatenate strings into a new string
- extract substrings with slices

`str()` constructor converts native types into strings

Text sequence type

```python
A = str('My String')
B = str("A sentence")
C = str('''The giant
text with a lot of
lines''')
```

Text sequence type

```
A = str('My String')
B = str("A sentence")
C = str('''The giant
text with a lot of
lines''')
D = B[0:2] + A[3:10]

print(D)
```

Text sequence type

```
A = str('My String')
B = str("A sentence")
C = str('''The giant
text with a lot of
lines''')
D = B[0:2] + A[3:10]

print(D)         #  "A String"
```

Text sequence type

Operation	Result
`str.isprintable()`	**True** if only printable characters and more than one char
`str.isascii()`	**True** if only ASCII characters and more than one char
`str.isalpha()`	**True** if only alphabetic characters and more than one char
`str.isalnum()`	**True** if only alphanumeric characters and more than one char
`str.isdigit()`	**True** if only numbers characters and more than one char
`str.isnumeric()`	**True** if only numeric characters and more than one char

Text sequence type

```
"abc42".isprintable()   #  True
"abc42".isascii()       #  True
"abc42".isalpha()       # False
"abc".isalpha()         #  True
"abc42".isalnum()       #  True
"42".isdigit()          #  True
"42.2".isdigit()        # False
"-42".isdigit()         # False
"-42".isnumeric()       # False
```

Text sequence type

Operation	Result
`str.lower()`	Put the string into lowercase
`str.upper()`	Put the string into uppercase
`str.capitalize()`	1st character is capitalized, other are lowercased
`str.title()`	Same as capitalized, but on each word
`str.swapcase()`	Reverse upper to lower case characters, and lower to upper
`str.isspace()`	**`True`** if there are only whitespaces and at least one character
`str.isupper()`	**`True`** if all characters are uppercase and at least one character

Don't forget: a string is immutable!

All of the string methods *always* return a copy of the string

Text sequence type

```
A = str('tHis is A TExT wIth')
print(A.capitalize())
```

Text sequence type

```python
A = str('tHis is A TExT wIth')
print(A.capitalize()) # "This is a text with"
print(A.title())
```

Text sequence type

```python
A = str('tHis is A TExT wIth')
print(A.capitalize()) # "This is a text with"
print(A.title())      # "THis Is A TExT WIth"
print(A.swapcase())
```

Text sequence type

```
A = str('tHis is A TExT wIth')
print(A.capitalize()) # "This is a text with"
print(A.title())      # "THis Is A TExT WIth"
print(A.swapcase())   # "ThIS IS a teXt WiTH"
```

Text sequence type

```
A = str('tHis is A TExT wIth')
print(A.capitalize()) # "This is a text with"
print(A.title())      # "THis Is A TExT WIth"
print(A.swapcase())   # "ThIS IS a teXt WiTH"

B = str("    ")
print(B.isspace())
print(B.isupper())
```

Text sequence type

```
A = str('tHis is A TExT wIth')
print(A.capitalize()) # "This is a text with"
print(A.title())      # "THis Is A TExT WIth"
print(A.swapcase())   # "ThIS IS a teXt WiTH"

B = str("   ")
print(B.isspace())   # True
print(B.isupper())   # False
```

Text sequence type

Operation	Result
`str.`**`removeprefix(p)`**	If the string begins with prefix *p*, then it is removed
`str.`**`removesuffix(s)`**	If the string ends with suffix *s*, then it is removed
`str.`**`lstrip(`*`[chars]`*`)`**	Remove leading characters present in *chars*. If no argument, remove whitespaces.
`str.`**`rstrip(`*`[chars]`*`)`**	Remove trailing characters present in *chars*. If no argument, remove whitespaces.
`str.`**`strip(`*`[chars]`*`)`**	Apply **`lstrip()`** and **`rstrip()`**

Text sequence type

```python
'TestHook'.removeprefix('Test')
```

Text sequence type

```
'TestHook'.removeprefix('Test')      # Hook
'BaseTestCase'.removeprefix('Test')
```

Text sequence type

```
'TestHook'.removeprefix('Test')       # Hook
'BaseTestCase'.removeprefix('Test')   # BaseTestCase

'Arthur: three!'.removeprefix('Arthur: ')
```

Text sequence type

```
'TestHook'.removeprefix('Test')       # Hook
'BaseTestCase'.removeprefix('Test')  # BaseTestCase

'Arthur: three!'.removeprefix('Arthur: ')  # three!
'Arthur: three!'.lstrip('Arthur: ')
```

Text sequence type

```
'TestHook'.removeprefix('Test')       # Hook
'BaseTestCase'.removeprefix('Test')   # BaseTestCase

'Arthur: three!'.removeprefix('Arthur: ')  # three!
'Arthur: three!'.lstrip('Arthur: ')        # ee!

'www.example.com'.lstrip('cmowz.')
```

Text sequence type

```
'TestHook'.removeprefix('Test')      # Hook
'BaseTestCase'.removeprefix('Test')  # BaseTestCase

'Arthur: three!'.removeprefix('Arthur: ')  # three!
'Arthur: three!'.lstrip('Arthur: ')        # ee!

'www.example.com'.lstrip('cmowz.')  # 'example.com'
'   spacious   '.lstrip()
```

Text sequence type

```
'TestHook'.removeprefix('Test')       # Hook
'BaseTestCase'.removeprefix('Test')   # BaseTestCase

'Arthur: three!'.removeprefix('Arthur: ')  # three!
'Arthur: three!'.lstrip('Arthur: ')        # ee!

'www.example.com'.lstrip('cmowz.')  # 'example.com'
'   spacious   '.lstrip()           # 'spacious   '

'Monty Python'.rstrip(' Python')
```

Text sequence type

```
'TestHook'.removeprefix('Test')      # Hook
'BaseTestCase'.removeprefix('Test')  # BaseTestCase

'Arthur: three!'.removeprefix('Arthur: ')  # three!
'Arthur: three!'.lstrip('Arthur: ')        # ee!

'www.example.com'.lstrip('cmowz.')  # 'example.com'
'   spacious   '.lstrip()           # 'spacious   '

'Monty Python'.rstrip(' Python')          # M
'Monty Python'.removesuffix(' Python')
```

Text sequence type

```
'TestHook'.removeprefix('Test')       # Hook
'BaseTestCase'.removeprefix('Test')  # BaseTestCase

'Arthur: three!'.removeprefix('Arthur: ')  # three!
'Arthur: three!'.lstrip('Arthur: ')        # ee!

'www.example.com'.lstrip('cmowz.')  # 'example.com'
'   spacious   '.lstrip()           # 'spacious   '

'Monty Python'.rstrip(' Python')        # M
'Monty Python'.removesuffix(' Python')  # Monty
```

Text sequence type

Operation	Result
`str.find(sub)` ***find(sub,start,end)***	Find 1st index of substring *sub* within the slice **[start:end]**, or return **−1**
`str.index(sub)` ***index(sub,start,end)***	Like **find**, but raise **ValueError** instead of returning **−1**
`str.replace(old,new)` ***replace(old,new,count)***	Replace all (or the *count* firsts) of the occurrencies of *old* by *new*

Text sequence type

```
"ballo" in "C'est ballo"
```

Text sequence type

```
"ballo" in "C'est ballo"        # True
"C'est ballo".find("ballo")
```

Text sequence type

```
"ballo" in "C'est ballo"        # True
"C'est ballo".find("ballo")     #  6
"alalala".find("la", 2, 5)
```

Text sequence type

```
"ballo" in "C'est ballo"        # True
"C'est ballo".find("ballo")     #  6
"alalala".find("la", 2, 5)      #  3
"ah".find("b")
```

Text sequence type

```
"ballo" in "C'est ballo"          # True
"C'est ballo".find("ballo")       #  6
"alalala".find("la", 2, 5)        #  3
"ah".find("b")                    # -1
"alalala".index("la", 2, 5)
```

Text sequence type

```
"ballo" in "C'est ballo"          # True
"C'est ballo".find("ballo")       #  6
"alalala".find("la", 2, 5)        #  3
"ah".find("b")                    # -1
"alalala".index("la", 2, 5)       #  3
"ah".index("b")
```

Text sequence type

```
"ballo" in "C'est ballo"          # True
"C'est ballo".find("ballo")       #  6
"alalala".find("la", 2, 5)        #  3
"ah".find("b")                    # -1
"alalala".index("la", 2, 5)       #  3
"ah".index("b")                   # [Exception]
"ahehih".replace("ih", "oh")
```

Text sequence type

```
"ballo" in "C'est ballo"         # True
"C'est ballo".find("ballo")      #  6
"alalala".find("la", 2, 5)       #  3
"ah".find("b")                   # -1
"alalala".index("la", 2, 5)      #  3
"ah".index("b")                  # [Exception]
"ahehih".replace("ih", "oh")     #  "ahehoh"
"hohoho".replace("ho", "ha")
```

Text sequence type

```
"ballo" in "C'est ballo"          # True
"C'est ballo".find("ballo")       #  6
"alalala".find("la", 2, 5)        #  3
"ah".find("b")                    # -1
"alalala".index("la", 2, 5)       #  3
"ah".index("b")                   # [Exception]
"ahehih".replace("ih", "oh")      #  "ahehoh"
"hohoho".replace("ho", "ha")      #  "hahaha"
"hohoho".replace("ho", "ha",2)
```

Text sequence type

```
"ballo" in "C'est ballo"       # True
"C'est ballo".find("ballo")    #  6
"alalala".find("la", 2, 5)     #  3
"ah".find("b")                 # -1
"alalala".index("la", 2, 5)    #  3
"ah".index("b")                # [Exception]
"ahehih".replace("ih", "oh")   #  "ahehoh"
"hohoho".replace("ho", "ha")   #  "hahaha"
"hohoho".replace("ho", "ha",2)#  "hahaho"
```

Text sequence type

Operation	**Result**
`str.join(iterable)`	Separate each element of *iterate* with the calling string
`str.partition(sep)`	Separate the string in a 3-tuple: - the substring before the 1st *sep* - *sep* - the substring after the 1st *sep*
`str.split(sep,maxsplit)` *`split(sep=None,maxsplit=-1)`*	Split the string into a list of words Split *maxsplit* times (-1 for all) *(Whitespaces are treated differently)*
`str.splitlines(keepends)` *`splitlines(keepends=False)`*	Split a string by lines *(see manual for line boundaries)*

Text sequence type

```
"|".join("alo alo")
```

Text sequence type

```
"|".join("alo alo")        # 'a|l|o| |a|l|o'
"alo alo".partition(" ")
```

Text sequence type

```
"|".join("alo alo")        # 'a|l|o| |a|l|o'
"alo alo".partition(" ") # ('alo', ' ', 'alo')
"a b c".partition(" ")
```

Text sequence type

```
"|".join("alo alo")        # 'a|l|o| |a|l|o'
"alo alo".partition(" ") # ('alo', ' ', 'alo')
"a b c".partition(" ")   # ('a', ' ', 'b c')
"a b c".split()
```

Text sequence type

```
"|".join("alo alo")          # 'a|l|o| |a|l|o'
"alo alo".partition(" ") # ('alo', ' ', 'alo')
"a b c".partition(" ")   # ('a', ' ', 'b c')
"a b c".split()          # ['a', 'b', 'c']
"abcbdbe".split()
```

Text sequence type

```
"|".join("alo alo")          # 'a|l|o| |a|l|o'
"alo alo".partition(" ")    # ('alo', ' ', 'alo')
"a b c".partition(" ")      # ('a', ' ', 'b c')
"a b c".split()             # ['a', 'b', 'c']
"abcbdbe".split()           # ['abcbdbe']
"abcbdbe".split("b", 1)
```

Text sequence type

```
"|".join("alo alo")          # 'a|l|o| |a|l|o'
"alo alo".partition(" ")     # ('alo', ' ', 'alo')
"a b c".partition(" ")       # ('a', ' ', 'b c')
"a b c".split()              # ['a', 'b', 'c']
"abcbdbe".split()            # ['abcbdbe']
"abcbdbe".split("b", 1)      # ['a', 'cbdbe']
"abcbdbe".split("b", -1)
```

Text sequence type

```
"|".join("alo alo")         # 'a|l|o| |a|l|o'
"alo alo".partition(" ") # ('alo', ' ', 'alo')
"a b c".partition(" ")   # ('a', ' ', 'b c')
"a b c".split()          # ['a', 'b', 'c']
"abcbdbe".split()        # ['abcbdbe']
"abcbdbe".split("b", 1)  # ['a', 'cbdbe']
"abcbdbe".split("b", -1) # ['a','c','d','e']
'::1:2:'.split(':')
```

Text sequence type

```
"|".join("alo alo")          # 'a|l|o| |a|l|o'
"alo alo".partition(" ")  # ('alo', ' ', 'alo')
"a b c".partition(" ")     # ('a', ' ', 'b c')
"a b c".split()              # ['a', 'b', 'c']
"abcbdbe".split()            # ['abcbdbe']
"abcbdbe".split("b", 1)    # ['a', 'cbdbe']
"abcbdbe".split("b", -1)  # ['a','c','d','e']
'::1:2:'.split(':')         # ['','','1','2', '']
"   a   b c   ".split()
```

Text sequence type

```
"|".join("alo alo")         # 'a|l|o| |a|l|o'
"alo alo".partition(" ")  # ('alo', ' ', 'alo')
"a b c".partition(" ")    # ('a', ' ', 'b c')
"a b c".split()           # ['a', 'b', 'c']
"abcbdbe".split()         # ['abcbdbe']
"abcbdbe".split("b", 1)   # ['a', 'cbdbe']
"abcbdbe".split("b", -1)  # ['a','c','d','e']
'::1:2:'.split(':')       # ['','','1','2', '']
"   a   b c   ".split()   # ['a', 'b', 'c']
```

Text sequence type

Operation	Result
`str.center(width)`	Center the string with leading and trailing whitespaces and make its length equal to *fill*
`str.zfill(width)`	Put leading 0 to fill the string and make its length equal to *fill* (eventually add − as a prefix)
`str.ljust(width)` *`ljust(width,fillchar)`*	Left justify for length *width*, and fill with *fillchar* characters
`str.rjust(width)` *`rjust(width,fillchar)`*	Right justify like **`str.ljust()`**

Text sequence type

```
"abc".center(7)
```

Text sequence type

```
"abc".center(7)    # '  abc  '
"abc".center(2)
```

Text sequence type

```
"abc".center(7)    # '  abc  '
"abc".center(2)    # 'abc'
"abc".center(4)
```

Text sequence type

```
"abc".center(7)    # '  abc  '
"abc".center(2)    # 'abc'
"abc".center(4)    # 'abc '
"  abc".center(7)
```

Text sequence type

```
"abc".center(7)   # '  abc  '
"abc".center(2)   # 'abc'
"abc".center(4)   # 'abc '
"  abc".center(7) # '   abc '
"abc".rjust(7)
"abc".ljust(7)
```

Text sequence type

```
"abc".center(7)    # '  abc  '
"abc".center(2)    # 'abc'
"abc".center(4)    # 'abc '
"  abc".center(7)  # '   abc '
"abc".rjust(7)     # '    abc'
"abc".ljust(7)     # 'abc    '
"42".zfill(5)
```

Text sequence type

```
"abc".center(7)   # '  abc  '
"abc".center(2)   # 'abc'
"abc".center(4)   # 'abc '
"  abc".center(7) # '   abc '
"abc".rjust(7)    # '    abc'
"abc".ljust(7)    # 'abc    '
"42".zfill(5)     # '00042'
"42.3".zfill(5)
```

Text sequence type

```python
"abc".center(7)    # '  abc  '
"abc".center(2)    # 'abc'
"abc".center(4)    # 'abc '
"  abc".center(7)  # '   abc '
"abc".rjust(7)     # '    abc'
"abc".ljust(7)     # 'abc    '
"42".zfill(5)      # '00042'
"42.3".zfill(5)    # '042.3'
```

Thank you for your attention