Langage Python

ISUP, Sorbonne Université

Etienne Guével
Ingénieur de Recherche - SCAI
etienne.guevel@sorbonne-universite.fr

Septembre-Novembre 2025

Contenu du cours

Partie 1 – Fondamentaux

- Syntaxe, structures de contrôle
- Types, structures de données

Partie 1

Fondamentaux

Concepts de base : syntaxe et types

Syntaxe

La syntaxe de Python repose sur une série d'instructions et des mots clés bien précis

```
>>> a = "Hello, world"
>>> print(a)
"Hello, world"
```

Syntaxe

La syntaxe de Python repose sur une série d'instructions et des mots clés bien précis

```
>>> a = "Hello, world"
>>> print(a)
"Hello, world"
```

- a est une **variable** et "Hello, world" est sa **valeur**

Syntaxe

La syntaxe de Python repose sur une série d'instructions et des mots clés bien précis

```
>>> a = "Hello, world"
>>> print(a)
"Hello, world"
```

- a est une **variable** et "Hello, world" est sa **valeur**
- la variable a est un objet

Syntaxe

La syntaxe de Python repose sur une série d'instructions et des mots clés bien précis

```
>>> a = "Hello, world"
>>> print(a)
"Hello, world"
```

- a est une **variable** et "Hello, world" est sa **valeur**
- la variable a est un objet
- la variable a a un type

```
>>> print(type(a))
str
```

Syntaxe

Les noms de variables sont libres à l'exception de certains mots réservés :

```
def, return, class, global, else, in
```

Liste complète ici

Conventions

- Ne pas mettre de caractère accentués ni de caractère non ASCII et préférer l'anglais
- Choisir des noms de variables qui soient compréhensibles
- Vous pouvez choisir des noms en plusieurs mots s'il n'est pas trop long, séparer les mots par un "_", ou mettre des majuscules.

Syntaxe

La syntaxe de Python repose sur une série d'instructions et des mots clés bien précis.

```
>>> a = "Hello, world"
>>> print(a)
"Hello, world"
```

Syntaxe

La syntaxe de Python repose sur une série d'instructions et des mots clés bien précis.

```
>>> a = "Hello, world"
>>> print(a)
"Hello, world"
```

Le même texte aurait pu être affiché d'une façon différente.

```
>>> a = "Hello,"
>>> b = "world"
>>> print(a, b)
"Hello, world"
```

Syntaxe

La syntaxe de Python repose sur une série d'instructions et des mots clés bien précis.

```
>>> a = "Hello, world"
>>> print(a)
"Hello, world"
```

Le même texte aurait pu être affiché d'une façon différente.

```
>>> a = "Hello,"
>>> b = "world"
>>> print(a, b)
"Hello, world"
```

Ou encore de cette façon.

```
>>> a = "Hello, "
>>> b = "world"
>>> print(a + b)
"Hello, world"
```

Types de base : types numériques

- Entier, `int`

```
>>> a = 1
```

- Flottant, `float`

```
>>> a = 1.1
```

- Booléen, `bool`

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

Opérations sur les types numériques

Opérations élémentaires :

```
>>> 10 + 4
14
>>> 10 - 4
6
>>> 10 * 4
40
>>> 10 ** 4
10000
>>> 10 / 4
2.5
>>> 10 / float(4)
2.5
>>> 7 // 3
2
>>> 7 % 3
1
```

Types de base : types itérables

Types itérables, i.e. des séquences

- Liste, `list`

```
>>> a = [1, 2, 3]
```

- Tuple, `tuple`

```
>>> a = (1, 2, 3)
```

- Dictionnaires, `dict`

```
>>> a = {"key1": 1, "key2": 2, "key3": 3}
```

Structures de contrôle

Structures de contrôle

Instruction if, elif, else

```
>>> if [condition1]:
...     [instructions]
... elif [condition2]:
...     [instructions]
... else:
...     [instructions]
```

Remarque : `elif` et `else` sont optionnels

Opérations sur les booléens et comparaisons

Opérations :

```
>>> a = True
>>> b = a and False    # idem que b = a & False
>>> c = not a
>>> d = bool(0)
>>> e = bool(1)
```

Comparaisons :

```
>>> 5 > 3
>>> 5 >= 3
>>> 5 != 3
>>> 5 == 5
>>> 5 > 3 and 6 > 3
>>> 5 > 3 or 5 < 3
>>> not False
>>> False or not False and True
```

Instruction if, elif, else

Exemple :

```
>>> if i == 0:
...     print("i equals 0")
```

Remarque : `elif` et `else` sont optionnels

Instruction if, elif, else

Exemple :

```
>>> i = 0
>>> condition = i != 0
>>> if not condition:
...     print("i equals 0")
```

Remarque : `elif` et `else` sont optionnels

Structures de contrôle

Il est possible de définir une variable selon certaines conditions.

```
>>> if not i:
...     a = 1
... else:
...     a = 2
```

```
>>> a = 1 if not i else 2
```

Structures de contrôle

Boucle while

```
>>> while [condition]:
...     [instructions]
```

Exemple :

```
>>> i = 0
>>> while i < 10:
...     print(i)
...     i += 1
```

Boucle while

```
>>> while [condition]:
...     [instructions]
```

Exemple :

```
>>> i = 0
>>> while i < 10:
...     print(i)
```

La boucle ne s'arrêtra que quand l'ordinateur plantera

Fonctions

Syntaxe

Précedemment nous avons utilisé print plusieurs fois.

```
>>> a = "Hello, world"
>>> print(a)
"Hello, world"
```

Syntaxe

Précedemment nous avons utilisé print plusieurs fois.

```
>>> a = "Hello, world"
>>> print(a)
"Hello, world"
```

Et de façons différentes.

```
>>> a = "Hello,"
>>> b = "world"
>>> print(a, b)
"Hello, world"
```

Syntaxe

Précedemment nous avons utilisé print plusieurs fois.

```
>>> a = "Hello, world"
>>> print(a)
"Hello, world"
```

Et de façons différentes.

```
>>> a = "Hello,"
>>> b = "world"
>>> print(a, b)
"Hello, world"
```

En fait **print est une fonction avec des arguments**.

```
print(*objects, sep=' ', end='\n', file=None, flush=
    False)
```

Fonctions

Mots clés pour définir une fonction : `def` et `return`

```
>>> def fct(a, b, c):
...     # a, b et c sont des arguments positionnels
...     d = (a + b) * c
...     return d

>>> fct(1, 2, 3)
```

- `fct` est le nom de notre fonction
- a, b et c sont les arguments (**nommés** ou **positionnels**)
- d est le retour de la fonction

Fonctions

Mots clés pour définir une fonction : `def` et `return`

```
>>> def fct(a=1, b=1, c=1):
...     # a, b et c sont des arguments nommes
...     d = (a + b) * c
...     return d

>>> fct()
```

- `fct` est le nom de notre fonction
- a, b et c sont les arguments (**nommés** ou **positionnels**)
- d est le retour de la fonction

Fonctions

Mots clés pour définir une fonction : `def` et `return`

```
>>> def fct(a, b, c=None):
...     # a et b sont des arguments positionnels, c est un
    argument nomme
...     if not c:
...         d = a + b
...     else:
...         d = (a + b) * c
...     return d

>>> fct(1, 2)
```

- `fct` est le nom de notre fonction
- a, b et c sont les arguments (**nommés** ou **positionnels**)
- d est le retour de la fonction

Listes

Listes

Une liste est un contenant permettant de concaténer différent objects :

```
>>> my_list = [True, 2, "3", 4]
```

Il n'y a pas de restrictions sur les objects contenus dans les listes, ils peuvent être de différents types et même être des listes !

Boucler sur une liste

```
>>> for item in my_list:
...     print(item)
```

Boucler sur une liste

```
>>> for item in my_list:
...     print(item)
```

Avec le mot clé enumerate

```
>>> for i, item in enumerate(my_list):
...     print(i, item)
```

Listes

Une liste est une séquence d'objets potentiellement de types différents :

```
>>> my_list = [True, 2, "3", 4]
```

Accès par indice :

```
my_list[start:stop:step]
```

Les indices commencent à 0 et peuvent être négatifs

Listes

Par exemple :

```
>>> my_list[0]
True

>>> my_list[0:2]
[True, 2]

>>> my_list[0:4:2]
[True, "3"]

>>> my_list[-1]
4
>>> my_list[::-1]
[4, "3", 2, True]
```

Listes

Autres opérations utiles :

```
>>> print(2 in my_list)
True

>>> list(range(4))
[0, 1, 2, 3]

>>> my_list + [10, 11]
[True, 2, "3", 4, 10, 11]

>>> my_list * 2
[True, 2, "3", 4, True, 2, "3", 4]
```

Opérations sur les listes

- Remplacer un élément
- Remplacer une sous-séquence (slicing)
- Supprimer des éléments
- Concaténer deux listes
- Répéter les élements d'une liste
- Ajout d'un élement en fin de liste

Méthodes et attributs d'une liste

Méthodes :

- `append`: ajout d'un élément.
- `clear`: vider la liste.
- `copy`: copier la liste.
- `count`: nombre de fois où l'élément apparaît.
- `index`: position où l'élément apparaît.
- `extend`: ajout d'une liste.
- `insert`: mettre un object à la position i.
- `pop`: supprimer l'élément à la position i.
- `remove`: supprimer l'élément i.
- `reverse`: inverser la liste.
- `sort`: trier la liste.

```
my_list = [1, 2, 3]
my_list.<nom de la methode>(<arguments>)
my_list.append(1)
```

Compréhension de liste

Exemple : calculer le carré de chaque élément d'une liste d'entiers

```
>>> my_list = [1, 2, 3, 4]
>>>
>>>
```

Compréhension de liste

Exemple : calculer le carré de chaque élément d'une liste d'entiers

```
>>> my_list = [1, 2, 3, 4]
>>> new_list = []
>>> for item in my_list:
...     new_list.append(item**2)
```

Compréhension de liste

Exemple : calculer le carré de chaque élément d'une liste d'entiers

```
>>> my_list = [1, 2, 3, 4]
>>> new_list = []
>>> for item in my_list:
...     new_list.append(item**2)
```

Via une liste de compréhension :

```
>>> my_list = [1, 2, 3, 4]
>>> new_list = [item**2 for item in my_list]
```

Plus rapide, plus lisible !

Chaînes de caractères

Chaînes de caractères

Plusieurs manières d'écrire

```
>>> s = "une chaine de caracteres"
>>> s = 'une chaine de caracteres'
>>> s = """une chaine
de caracteres"""
```

Accès aux éléments

```
>>> s = "python"
>>> print(s[0])
"p"
>>> print(s[1:3])
"yt"
>>> print(s[1:6:2])
"yhn"
```

Chaînes de caractères

Boucler sur un chaine de caractères

```
>>> for item in "python":
...     print(item)
```

Concaténer plusieurs chaînes de caractères

```
>>> s = "une chaine" + "de" + "caracteres"
>>> print(s)
"une chainedecaracteres"
```

f-strings

Formatter une chaîne de caractères

```
>>> pi = 3.14159
>>> print(f"pi = {pi}")
pi = 3.14159
>>> print(f"pi = {pi:.2f}")
pi = 3.14
>>> print(f"pi = {pi:8.2f}")
pi =     3.14
```

Formatter une chaîne de caractères

```
coordinates = [(53.123, 26.589), (20.720, 70.447), (80.294,
    50.382)]

print(f"{'Index':>10} | {'Latitude':<9} | {'Longitude':<9}")
for i, (lat, lon) in enumerate(coordinates):
    print(f"{i:10} | {lat:<9.3f} | {lon:<9.3f}")
```

Outputs :

```
     Index | Latitude  | Longitude
         0 | 53.123    | 26.589
         1 | 20.720    | 70.447
         2 | 80.294    | 50.382
```

Chaînes de caractères

Quelques méthodes utiles :

- `lower`
- `upper`
- `join`
- `replace`
- `split`

Chaînes de caractères

```
>>> a = "    Hello, etienne guevel   "
>>> a = a.strip()
>>> a = a.title()
>>> print(a)
Hello, Etienne Guevel
```

Chaînes de caractères

```
>>> a = "    Hello, etienne guevel   "
>>> a = a.strip()
>>> a = a.title()
>>> print(a)
Hello, Etienne Guevel
```

Les méthodes peuvent être chaînées !

```
>>> a = "    Hello, etienne guevel   "
>>> a = a.strip().title()
>>> print(a)
Hello, Etienne Guevel
```

Typage

Langage dynamiquement typé

Dynamique vs Statique

- Dynamique : le type est déterminé au **moment de l'exécution** et peut **changer**
- Statique : le type est fixé en début de programme

Inférence de type

Python détermine automatiquement le type d'une variable

```
>>> a = 1
>>> print(type(a))
int
>>> a = "hello"
>>> print(type(a))
str
```

Typage fort

- le type d'un objet est déterminé par l'ensemble de ses caractéristiques
- En particulier, une même opération peut fonctionner sur **objets de type différents**, si tant est que les opérations soient valables

```
>>> 5 + 4
9
>>> "titi" + "toto"
"tititoto"
```

Typage fort

Corolaire

Python **interdit** des opérations ayant peu de sens et **ne cherche pas à convertir** lui même.

Par exemple :

- **On ne peut pas** ajouter une chaîne de caractère et un entier
- **On peut** multiplier une chaîne de caractère et un entier

```
>>> "titi" * 2
"tititititi"

>>> "titi" + 2
TypeError: can only concatenate str (not "int") to str
```

Tuples

Tuples

Un tuple est une séquence immutable d'objets potentiellement de types différents :

```
>>> my_tuple = (True, 2, "3", 4)
>>> my_tuple = (1,)
```

Accès par indice :

```
>>> my_tuple[0]
True
```

Comparaison avec les tuples

- ~~Remplacer un élément~~
- ~~Remplacer une sous-séquence (slicing)~~
- ~~Supprimer des éléments~~
- Concaténer deux tuples
- Répeter les élements d'un tuple
- ~~Ajout d'un élement en fin de tuple~~

Boucler sur un tuple

```
>>> for item in my_tuple:
...     print(item)
```

Boucler sur un tuple

```
>>> for item in my_tuple:
...     print(item)
```

Avec le mot clé `range`

```
>>> for i in range(len(my_tuple)):
...     print(my_tuple[i])
```

Boucler sur un tuple

```
>>> for item in my_tuple:
...     print(item)
```

Avec le mot clé range

```
>>> for i in range(len(my_tuple)):
...     print(my_tuple[i])
```

Avec le mot clé enumerate

```
>>> for i, item in enumerate(my_tuple):
...     print(i, item)
```

Dictionnaires

Dictionnaires

Un dictionnaire fonctionne avec le paradigme clé / valeur

```
>>> Larousse["python"]
Serpent monstrueux de la mythologie grecque, qui rendait ses
oracles a Delphes.
Il fut tue par Apollon qui etablit a Delphes son propre
oracle.
```

Dictionnaires

Un dictionnaire est une séquence mutable selon le paradigme clé/valeurs :

```
>>> my_dict = {"key1": 1, "key2": 2}
>>> my_dict = dict(key1=1, key2=2)
```

Accès par clé :

```
>>> my_dict["key1"]
1
>>> my_dict.get("key1")
1
```

Dictionnaires

Un dictionnaire est une séquence mutable selon le paradigme clé/valeurs :

```
>>> my_dict = {"key1": 1, "key2": 2}
>>> my_dict = dict(key1=1, key2=2)
```

Accès par clé :

```
>>> my_dict["key1"]
1
>>> my_dict.get("key1")
1
```

Que se passe-t-il si la clé n'existe pas ?

```
>>> my_dict["key3"]
?
>>> my_dict.get("key3")
?
```

Dictionnaires

Ajouter un élément :

```
>>> my_dict["new_key"] = new_value
```

Dictionnaires

Ajouter un élément :

```
>>> my_dict["new_key"] = new_value
```

Utiliser la méthode `update`

```
>>> my_dict.update({"new_key": new_value})
>>> my_dict |= {"other_key": other_value} #since python 3.9
>>> my_dict["new_key"]
new_value
```

Opérations sur les dictionnaires

Opérations

- Ajouter un élément
- Remplacer un élément
- Supprimer des éléments
- Concaténer deux dictionnaires

Méthodes associées

- `get`
- `keys`
- `values`
- `items`
- `clear`
- `pop`
- `update`

Boucler sur un dictionnaire

```
>>> for key in my_dict:
...     print(key, my_dict[key])
```

Boucler sur un dictionnaire

```
>>> for key in my_dict:
...     print(key, my_dict[key])
```

```
>>> for value in my_dict.values():
...     print(value)
```

Boucler sur un dictionnaire

```
>>> for key in my_dict:
...     print(key, my_dict[key])
```

```
>>> for value in my_dict.values():
...     print(value)
```

```
>>> for key, value in my_dict.items():
...     print(key, value)
```

Ensembles

Ensemble

Un ensemble est une séquence **mutable** contenant des éléments **ordonnés** et **uniques**. Un ensemble vide est créé par `set()`.

```
>>> a = {1, 2, 3, 3, 3, 3}
>>> print(a)
{1, 2, 3}
```

Intérêt des ensembles :

- Tests d'appartenance d'un élément à une séquence
- Suppression de doublons

Ensemble

Des opérations sont possibles sur les sets :

```
>>> french_fruits = {"apple", "pear", "banana", "peach"}
>>> american_fruits = {"banana", "peach", "pineapple", "
    avocado"}
>>> french_fruits & american_fruits
{"banana", "peach"}
>>> french_fruits | american_fruits
{"apple", "pear", "banana", "peach", "pineapple", "avocado"}
>>> french_fruits - american_fruits
{"apple", "pear"}
>>> french_fruits ^ american_fruits
{"apple", "pear", "pineapple", "avocado"}
```

Unpacking

Unpacking

```
>>> a, b = [1, 2]
>>> print(a, b)
1 2
```

Unpacking

Unpacking

```
>>> a, b = [1, 2]
>>> print(a, b)
1 2
```

Unpacking plus complexe

```
>>> name, (taille, poids) = ["Antoine", (1.75, 90)]
>>> print(name, taille, poids)
Antoine 1.75 90
```

Unpacking

Unpacking

```
>>> a, b = [1, 2]
>>> print(a, b)
1 2
```

Unpacking plus complexe

```
>>> name, (taille, poids) = ["Antoine", (1.75, 90)]
>>> print(name, taille, poids)
Antoine 1.75 90
```

Unpacking avec une taille inconnue

```
>>> start, *_, end = list(range(10))
>>> print(start, end)
0 9
```

Mutabilité

Mutable vs immutable

Objet mutable

Un objet mutable peut être modifié après sa création

- list
- dict
- set

Objet immutable

Un objet immutable ne peut être modifié après sa création

- int, float, bool
- str
- tuple
- byte

Mutable vs immutable

Immutabilité : modification d'un entier

```
>>> a = 1
>>> id(a)
xxxxxxx560
>>> a += 1
>>> id(a)
xxxxxxx592
```

Que se passe-t-il ici ?

Mutable vs immutable

Immutabilité : modification d'un entier

```
>>> a = 1
>>> id(a)
xxxxxxx560
>>> a += 1
>>> id(a)
xxxxxxx592
```

Que se passe-t-il ici ?

Copie implicite

Mutable vs immutable

Mutabilité : modification d'une liste

```
>>> a = [1, 2]
>>> id(a)
xxxxxxx328
>>> a[0] += 1
>>> id(a)
xxxxxxx328
```

Mutable vs immutable

Autre exemple, `int`

```
>>> a = 1
>>> b = a
>>> b is a
True
>>> b += 1
>>> b
2


>>> b is a
False
```

Mutable vs immutable

Autre exemple, `list`

```
>>> a = [1, 2]
>>> b = a
>>> b is a
True
>>> b[0] += 1
>>> b
[2, 2]
>>> a
[2, 2]
>>> b is a
True
```

En bref

- Python gère les objets mutables et immutables différemment
- Les objets mutables sont très intéressants si on a besoin de changer leur structures (taille, type, etc.) mais cela peut être dangereux
- Les objets immutables sont plus rapides d'accès
- Les objets immutables doivent être préférés si on veux que l'objet reste le même tout au long de l'exécution
- Modifier un objet immutable est plus couteux car il nécessite une copie (explicite ou non)

Portée des variables

Portée des variables

Comme dans les autres langages, il existe deux types de variables en python :

- locales : des variables définies dans une fonction
- globales : des variables définies en dehors des fonctions

```
>>> val = 0
>>> def sum(a, b):
...
...
...     val = a + b
...     return val

>>> sum(1, 2)
???


>>> print(val)
???
```

Portée des variables

Comme dans les autres langages, il existe deux types de variables en python :

- locales : des variables définies dans une fonction
- globales : des variables définies en dehors des fonctions

```
>>> val = 0
>>> def sum(a, b):
...
...
...     val = a + b
...     return val

>>> sum(1, 2)
3


>>> print(val)
0
```

Portée des variables

Comme dans les autres langages, il existe deux types de variables en python :

- locales : des variables définies dans une fonction
- globales : des variables définies en dehors des fonctions

```
>>> val = 0
>>> def sum(a, b):
...
...     val += 1
...     total_sum = a + b + val
...     return total_sum

>>> sum(1, 2)
UnboundLocalError: local variable "val" referenced before
    assignment

>>> print(val)
0
```

Portée des variables

Comme dans les autres langages, il existe deux types de variables en python :

- locales : des variables définies dans une fonction
- globales : des variables définies en dehors des fonctions

```
>>> val = 0
>>> def sum(a, b):
...     global val
...     val += 1
...     total_sum = a + b + val
...     return total_sum

>>> sum(1, 2)
4


>>> print(val)
1
```

Modules et imports

Modules et imports

Définition

Un module est un fichier python contenant un ensemble d'instructions (fonctions, classes, etc.)

Un module peut être

- Dans une bibliothèque standard de Python (exemple: `random`).

Modules et imports

Définition

Un module est un fichier python contenant un ensemble d'instructions (fonctions, classes, etc.)

Un module peut être

- Dans une bibliothèque standard de Python (exemple: `random`).
- Inclus dans un package ou une librairie installée (exemple: `numpy`).

Modules et imports

Définition

Un module est un fichier python contenant un ensemble d'instructions (fonctions, classes, etc.)

Un module peut être

- Dans une bibliothèque standard de Python (exemple: `random`).
- Inclus dans un package ou une librairie installée (exemple: `numpy`).
- Créé localement, par exemple un fichier `mon_module.py`

Modules et imports

Définition

Un module est un fichier python contenant un ensemble d'instructions (fonctions, classes, etc.)

Un module peut être

- Dans une bibliothèque standard de Python (exemple: `random`).
- Inclus dans un package ou une librairie installée (exemple: `numpy`).
- Créé localement, par exemple un fichier `mon_module.py`

Permet

1. Utiliser les fonctionnalités d'un module dans un autre
2. Structurer un programme python en plusieurs fichiers (implémentation **modulaire**)
3. Utiliser des packages open-source

Modules et imports

Exemple : `random`

```
>>> import random
>>> random.randint(1, 10)
4
>>> random.choice(["apple", "banana", "cherry"])
"banana"
```

Modules et imports

Exemple : `random`

```
>>> import random
>>> random.randint(1, 10)
4
>>> random.choice(["apple", "banana", "cherry"])
"banana"
```

Utilisation de `from` pour importer des fonctions spécifiques:

```
>>> from random import randint, choice
>>> randint(1, 10)
4
>>> choice(["apple", "banana", "cherry"])
"banana"
```

Modules et imports

Exemple : `random`

```
>>> import random
>>> random.randint(1, 10)
4
>>> random.choice(["apple", "banana", "cherry"])
"banana"
```

Utilisation de `from` pour importer des fonctions spécifiques:

```
>>> from random import randint, choice
>>> randint(1, 10)
4
>>> choice(["apple", "banana", "cherry"])
"banana"
```

Autres modules : collections, math, copy, typing, etc.

Installation de modules

- Utiliser un gestionnaire de paquets (pip, conda, etc.)
- Exemple: `pip install numpy`
- Utiliser Anaconda navigator

Utilisation des modules importés

```
# On peut importer un module avec un alias
>>> import numpy as np
>>> np.sqrt([1, 2, 3])
array([1.         , 1.41421356, 1.73205081])
>>> from numpy import sqrt as square_root
```

Modules et imports

Exemple : `my_module.py`

```
def prod(a, b):
    return a * b

def add(a, b):
    return a + b
```

Modules et imports

Exemple : `my_module.py`

```
def prod(a, b):
    return a * b

def add(a, b):
    return a + b
```

```
>>> import my_module
>>> my_module.add(1, 2)
3
>>> my_module.prod(1, 2)
2
```

Modules et imports

Exemple : `my_module.py`

```
def prod(a, b):
    return a * b

def add(a, b):
    return a + b
```

```
>>> import my_module as mod
>>> mod.add(1, 2)
3
>>> mod.prod(1, 2)
2
```

Modules et imports

Exemple : `my_module.py`

```
def prod(a, b):
    return a * b

def add(a, b):
    return a + b
```

```
>>> from my_module import add
>>> add(1, 2)
3
>>> prod(1, 2)
NameError: name "prod" is not defined
```

Modules et imports

Exemple : `my_module.py`

```
def prod(a, b):
    return a * b

def add(a, b):
    return a + b
```

```
>>> from my_module import add, prod
>>> add(1, 2)
3
>>> prod(1, 2)
2
```

Modules et imports

Exemple : `my_module.py`

```
def prod(a, b):
    return a * b

def add(a, b):
    return a + b
```

```
>>> from my_module import *     # NE SURTOUT PAS UTILISER
>>> add(1, 2)
3
>>> prod(1, 2)
2
```

Pourquoi proscrire l'utilisation de `import *` ?

Exemple : la fonction `sqrt` existe dans plusieurs libraries :

- Dans `math` : calcule la racine carré pour un scalaire
- Dans `numpy` : calcule la racine carré pour un scalaire ou pour chaque élément d'un tableau

```
>>> from numpy import *
>>> from math import *
>>> print(sqrt([1, 2, 3]))
TypeError: must be real number, not list
```

Structurer un programme

Structure en plusieurs fichiers

- Un ou plusieurs modules contenant les fonctionnalités implémentées
- Un programme principal

```
my_project/
  - module1.py
  - module2.py
  - main.py
```

Structurer un programme

module1.py

```
import math
import sys

def prod(a, b):
    return a * b

def add(a, b):
    return a + b
```

main.py

```
from module1 import add, prod
print(add(1, 2))
print(prod(1, 2))
```