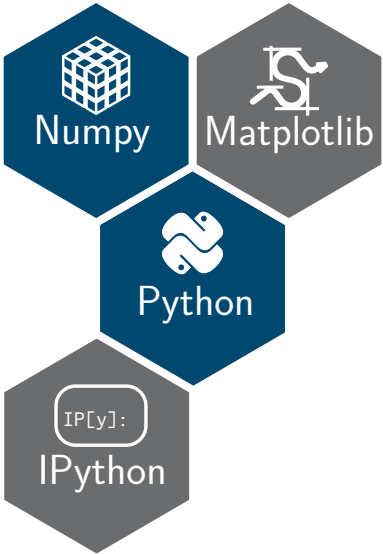


Option « Programmation en Python »

Variables, structure conditionnelle et autres boucles

Déclaration de variables

- En raison du **typage dynamique**, Python permet de déclarer des variables sans en préciser le type (entier, nombre flottant, ...)

```
In [1]: i = 2

In [2]: x = 10.5

In [3]: s = "Une noisette, j'la casse entre mes fesses tu vois... JCVD"
```

```
In [4]: %whos
Variable   Type     Data/Info
-----------------------------
i          int      2
s          str      Une noisette, j'la casse <...>es fesses tu vois... JCVD
x          float    10.5
```

~~Déclaration~~ Initialisation de variables

- En raison du **typage dynamique**, Python permet de déclarer des variables sans en préciser le type (entier, nombre flottant, ...)

```
In [1]: i = 2

In [2]: x = 10.5

In [3]: s = "Une noisette, j'la casse entre mes fesses tu vois... JCVD"
```

```
In [4]: %whos
Variable   Type     Data/Info
-----------------------------
i          int      2
s          str      Une noisette, j'la casse <...>es fesses tu vois... JCVD
x          float    10.5
```

~~Déclaration~~ Initialisation de variables

- En raison du **typage dynamique**, Python permet de déclarer des variables sans en préciser le type (entier, nombre flottant, ...)

```
In [1]: i = 2

In [2]: x = 10.5

In [3]: s = "Une noisette, j'la casse entre mes fesses tu vois... JCVD"
```

```
In [4]: %whos
Variable   Type     Data/Info
----------------------------
i          int      2
s          str      Une noisette, j'la casse <...>es fesses tu vois... JCVD
x          float    10.5
```

Type fondamentaux

- La fonction **`type`** permet d'accéder au type d'une variable
- **Nombre entier**

```
In [1]: i = 2
In [2]: type(i)
Out[2]: int
```

- **Nombre flottant**

```
In [1]: x = 10.5
In [2]: type(x)
Out[2]: float
```

Type fondamentaux

- La fonction **type** permet d'accéder au type d'une variable
- **Nombre entier**

```
In [1]: i = 2
In [2]: type(i)
Out[2]: int
```

- **Nombre flottant**

```
In [1]: x = 10.5
In [2]: type(x)
Out[2]: float
```

Type fondamentaux

- La fonction **type** permet d'accéder au type d'une variable
- **Nombre entier**

```
In [1]: i = 2
In [2]: type(i)
Out[2]: int
```

- **Nombre flottant**

```
In [1]: x = 10.5
In [2]: type(x)
Out[2]: float
```

Nombre complexe

```
In [1]: z = 1.5 + 0.5j
In [2]: type(z)
Out[2]: complex
In [3]: z.real
Out[3]: 1.5
In [4]: z.imag
Out[4]: 0.5
```

⚠ On notera que le nombre complexe i est noté `j` ou `J` en Python

```
In [1]: j = 5

In [2]: 2 + 5*j
Out[2]: 27

In [3]: 2 + 5j
Out[3]: (2+5j)
```

- **Booléen**

```
In [1]: b = 3 > 4
In [2]: type(b)
Out[2]: bool
In [3]: b
Out[3]: False
```

Conversion explicite

- Nombre entier → nombre flottant

```
In [1]: i = 2
In [2]: i
Out[2]: 2
In [3]: x = float(i)
In [4]: x
Out[4]: 2.0
```

- Nombre entier → booléen

```
In [1]: b = bool(i)
In [2]: b
Out[2]: True
```

⚠ Toute valeur différente de 0 est considérée comme vraie

Conversion explicite

- Nombre entier → nombre flottant

```
In [1]: i = 2
In [2]: i
Out[2]: 2
In [3]: x = float(i)
In [4]: x
Out[4]: 2.0
```

- Nombre entier → booléen

```
In [1]: b = bool(i)
In [2]: b
Out[2]: True
```

⚠ Toute valeur différente de 0 est considérée comme vraie

Conversion explicite

- Nombre entier → nombre complexe

```
In [1]: z = complex(i)
In [2]: z
Out[2]: (2+0j)
```

- Nombre complexe → nombre flottant

```
In [1]: z = 1.5 + 0.5j
In [2]: x = float(z.imag)
In [3]: x
Out[3]: 0.5
```

Conversion explicite

- Nombre entier → nombre complexe

```
In [1]: z = complex(i)
In [2]: z
Out[2]: (2+0j)
```

- Nombre complexe → nombre flottant

```
In [1]: z = 1.5 + 0.5j
In [2]: x = float(z.imag)
In [3]: x
Out[3]: 0.5
```

Opérations & comparaisons

- Opérations arithmétiques +, -, *, /, // division entière, ** puissance, % modulo

```
In [1]: 1 + 2, 1 - 2, 1 * 2, 1 / 2
Out[1]: (3, -1, 2, 0)

In [2]: 1.0 + 2.0, 1.0 - 2.0, 1.0 * 2.0, 1.0 / 2.0
Out[2]: (3.0, -1.0, 2.0, 0.5)

In [3]: 2**2
Out[3]: 4

In [4]: 3.0 // 2.0
Out[4]: 1.0

In [5]: 3.0 % 2.0
Out[5]: 1.0
```

⚠ En Python 2.X, la division entière pouvait se faire à l'aide de l'opérateur / dès lors que des entiers étaient impliqués.
Python 3.X a introduit l'opérateur // pour lever toutes ambiguïtés.

Opérations & comparaisons

- Opérations arithmétiques +, -, *, /, // division entière, ** puissance, % modulo

```
In [1]: 1 + 2, 1 - 2, 1 * 2, 1 / 2
Out[1]: (3, -1, 2, 0)

In [2]: 1.0 + 2.0, 1.0 - 2.0, 1.0 * 2.0, 1.0 / 2.0
Out[2]: (3.0, -1.0, 2.0, 0.5)

In [3]: 2**2
Out[3]: 4

In [4]: 3.0 // 2.0
Out[4]: 1.0

In [5]: 3.0 % 2.0
Out[5]: 1.0
```

⚠ En Python 2.X, la division entière pouvait se faire à l'aide de l'opérateur / dès lors que des entiers étaient impliqués.
Python 3.X a introduit l'opérateur // pour lever toutes ambiguïtés.

- Opérations arithmétiques (suite) : `+=`, `-=`, `*=`, `/=`, `%=`, `**=`

```
In [1]: x = 1.0
In [2]: x = x + 1.5
In [3]: x += 1.5

In [4]: i = 0
In [5]: i += 1
```

⚠ Contrairement au C/C++, les opérateurs ++ et -- n'existent pas en Python.

- Opérations arithmétiques (suite) : `+=`, `-=`, `*=`, `/=`, `%=`, `**=`

```
In [1]: x = 1.0
In [2]: x = x + 1.5
In [3]: x += 1.5

In [4]: i = 0
In [5]: i += 1
```

⚠ Contrairement au C/C++, les opérateurs ++ et -- n'existent pas en Python.

Opérations & comparaisons

- Opérations booléennes : `and`, `or` et `not`

```
In [1]: True and False
Out[1]: False

In [2]: not False
Out[2]: True

In [3]: True or False
Out[3]: True
```

- Opérateur de comparaison : >, <, >=, <=, ==, !=

```
In [1]: 2 > 1, 2 < 1
Out[1]: (True, False)

In [2]: 2 > 2, 2 < 2
Out[2]: (False, False)

In [3]: 2 == 2
Out[3]: True

In [4]: 2 != 2
Out[4]: False
```

Affectation multiples et parallèles

- Python autorise **l'affectation simultanée** d'une même valeur à plusieurs variables

```
In [1]: x = y = 1.0

In [2]: x, y
Out[2]: (1.0, 1.0)
```

- Python permet également **l'affectation en parallèle** de plusieurs variables

```
In [1]: x, y = 1.0, 1.0

In [2]: x, y
Out[2]: (1.0, 1.0)
```

Affectation multiples et parallèles

- Python autorise **l'affectation simultanée** d'une même valeur à plusieurs variables

```
In [1]: x = y = 1.0

In [2]: x, y
Out[2]: (1.0, 1.0)
```

- Python permet également **l'affectation en parallèle** de plusieurs variables

```
In [1]: x, y = 1.0, 1.0

In [2]: x, y
Out[2]: (1.0, 1.0)
```

Application : échange de deux valeurs

```
In [1]: x, y = 1.0, 2.0

In [2]: x, y
Out[2]: (1.0, 2.0)

In [3]: x, y = y, x

In [4]: x, y
Out[4]: (2.0, 1.0)
```

Mot-clés réservés

- Parmi les bonnes pratiques de programmation, le nom des variables doit être le plus clair et le plus explicite possible pour le développeur comme pour un lecteur non averti

```
In [1]: planck_constant = 6.626e-34 # J.s
In [2]: pc = 6.626e-34              # J.s
In [3]: energy = pc                 # WTF !?
```

Mot-clés réservés

- Toutefois, un certain nombre de mot-clés sont réservés au langage Python

  `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, **`lambda`**, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `while`, `with`, **`yield`**

- Rien n'empêche en revanche, d'utiliser des noms de variables identiques à certaines fonctions du Python

```
In [1]: type = 666
In [2]: type(666)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-7e2d10a8adcc> in <module>()
----> 1 type(666)

TypeError: 'int' object is not callable
```

Mot-clés réservés

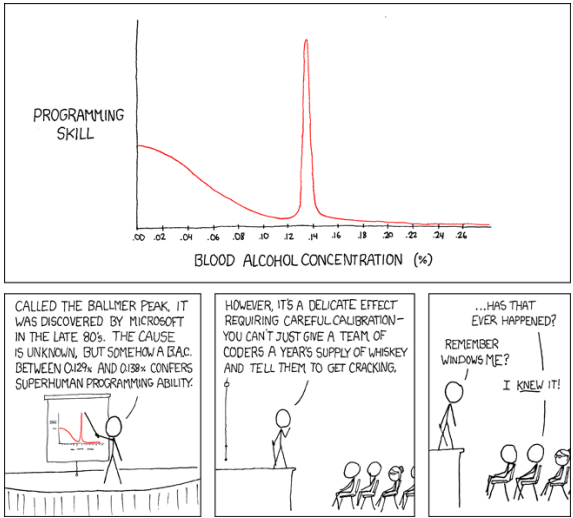
- Toutefois, un certain nombre de mot-clés sont réservés au langage Python

  `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, **`lambda`**, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `while`, `with`, **`yield`**

- Rien n'empêche en revanche, d'utiliser des noms de variables identiques à certaines fonctions du Python

```
In [1]: type = 666
In [2]: type(666)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-7e2d10a8adcc> in <module>()
----> 1 type(666)

TypeError: 'int' object is not callable
```


PROGRAMMING SKILL
.00 .02 .04 .06 .08 .10 .12 .14 .16 .18 .20 .22 .24 .26
BLOOD ALCOHOL CONCENTRATION (%)
CALLED THE BALLMER PEAK. IT WAS DISCOVERED BY MICROSOFT IN THE LATE 80'S. THE CAUSE IS UNKNOWN, BUT SOMEHOW A B.A.C. BETWEEN 0.129% AND 0.138% CONFERS SUPERHUMAN PROGRAMMING ABILITY.
HOWEVER, IT'S A DELICATE EFFECT REQUIRING CAREFUL CALIBRATION— YOU CAN'T JUST GIVE A TEAM OF CODERS A YEAR'S SUPPLY OF WHISKEY AND TELL THEM TO GET CRACKING.
...HAS THAT EVER HAPPENED?
REMEMBER WINDOWS ME?
I KNEW IT!

† xkcd comic

Structure conditionnelle

```
In [1]: test1 = False
   ...: test2 = False
   ...:
   ...: if test1:
   ...:     print("test1 est True")
   ...:
   ...: elif test2:
   ...:     print("test2 est True")
   ...:
   ...: else:
   ...:     print("test1 & test2 sont False")
```

- utilisation des mot-clés **`if/elif/else`**
- la fin de chaque condition est matérialisée par **le caractère :**
- **l'indentation (4 espaces ou une tabulation) délimite le bloc de condition**
- dans `ipython`, appuyer sur `Entrée` deux fois pour exécuter le bloc

Structure conditionnelle

```
In [1]: test1 = False
   ...: test2 = False
   ...:
   ...: if test1:
   ...:     print("test1 est True")
   ...:
   ...: elif test2:
   ...:     print("test2 est True")
   ...:
   ...: else:
   ...:     print("test1 & test2 sont False")
```

- utilisation des mot-clés **`if/elif/else`**
- la fin de chaque condition est matérialisée par **le caractère :**
- **l'indentation (4 espaces ou une tabulation) délimite le bloc de condition**
- dans `ipython`, appuyer sur `Entrée` deux fois pour exécuter le bloc

Structure conditionnelle

```
In [1]: test1 = False
   ...: test2 = False
   ...:
   ...: if test1:
   ...:     print("test1 est True")
   ...:
   ...: elif test2:
   ...:     print("test2 est True")
   ...:
   ...: else:
   ...:     print("test1 & test2 sont False")
```

- utilisation des mot-clés **`if/elif/else`**
- la fin de chaque condition est matérialisée par **le caractère :**
- **l'indentation (4 espaces ou une tabulation) délimite le bloc de condition**
- dans `ipython`, appuyer sur `Entrée` deux fois pour exécuter le bloc

Structure conditionnelle

```
In [1]: test1 = False
   ...: test2 = False
   ...:
   ...: if test1:
   ...:     print("test1 est True")
   ...:
   ...: elif test2:
   ...:     print("test2 est True")
   ...:
   ...: else:
   ...:     print("test1 & test2 sont False")
```

- utilisation des mot-clés **`if/elif/else`**
- la fin de chaque condition est matérialisée par **le caractère :**
- **l'indentation (4 espaces ou une tabulation) délimite le bloc de condition**
- dans `ipython`, appuyer sur `Entrée` deux fois pour exécuter le bloc

Structure conditionnelle

```
In [1]: test1 = False
   ...: test2 = False
   ...:
   ...: if test1:
   ...:     print("test1 est True")
   ...:
   ...: elif test2:
   ...:     print("test2 est True")
   ...:
   ...: else:
   ...:     print("test1 & test2 sont False")
```

```
bool test1 = false;
bool test2 = false;

if (test1)
  {
    cout << "test1 est True" << endl;
  }
 else if (test2)
   {
     cout << "test2 est True" << endl;
   }
 else
   {
     cout << "test1 & test2 sont False" << endl;
   }
```

- Python offre la possibilité de former des *expressions* dont l'évaluation est soumise à une condition

```
In [1]: x = 2.0

In [2]: y = x if x < 0 else x**2
In [3]: y
Out[3]: 4.0

In [4]: print("y est positif" if y > 0 else "y est négatif")
Out[4]: y est positif
```

- Pour répéter un bloc d'instructions tant qu'une condition est réalisée, Python propose la clause **while**
- Suite de Syracuse

```
In [3]: n = 27
In [4]: while n != 1:
   ...:     if n % 2:
   ...:         n = 3*n+1
   ...:     else:
   ...:         n //= 2
   ...:     print(n, end=" ")
   ...:
82 41 124 62 31 94 47 142 71 214 107 322 161 484 242 121 364 182 91 274 137 412 206 103
310 155 466 233 700 350 175 526 263 790 395 1186 593 1780 890 445 1336 668 334 167 502
251 754 377 1132 566 283 850 425 1276 638 319 958 479 1438 719 2158 1079 3238 1619 4858
2429 7288 3644 1822 911 2734 1367 4102 2051 6154 3077 9232 4616 2308 1154 577 1732 866
433 1300 650 325 976 488 244 122 61 184 92 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
```

Boucles for

- Pour répéter un certain nombre de fois un bloc d'instructions, on utilisera la construction suivante

```
for variable in objet:
    bloc de commandes
```

- Exemple :

```
In [1]: for c in "abcdef":
   ...:     print(c)
   ...:
a
b
c
d
e
f
```

- Pour répéter un certain nombre de fois un bloc d'instructions, on utilisera la construction suivante

```
for variable in objet:
    bloc de commandes
```

- Exemple :

```
In [1]: for c in "abcdef":
   ...:     print(c)
   ...:
a
b
c
d
e
f
```

- Autres exemples:

```
In [1]: for i in range(4):
   ...:     print(i)
   ...:
   0
   1
   2
   3
```

```
In [1]: for i in range(-4, 4, 2):
   ...:     print(i)
   ...:
   -4
   -2
   0
   2
```

- Autres exemples:

```
In [1]: for i in range(4):
   ...:     print(i)
   ...:
   0
   1
   2
   3
```

```
In [1]: for i in range(-4, 4, 2):
   ...:     print(i)
   ...:
   -4
   -2
   0
   2
```

Instructions `break` & `continue`

- Pour quitter une boucle `for` en cours d'exécution, on utilisera l'instruction **`break`**

```
In [1]: for i in range(-4,4,2):
   ...:     if i == 0:
   ...:         break
   ...:     print(i)
   ...:
-4
-2
```

- S'il s'agit de passer outre le bloc d'instruction suivant, on utilisera l'instruction **`continue`**

```
In [1]: for i in range(-4,4,2):
   ...:     if i == 0:
   ...:         continue
   ...:     print(i)
   ...:
-4
-2
2
```

- Pour quitter une boucle for en cours d'exécution, on utilisera l'instruction **break**

```
In [1]: for i in range(-4,4,2):
   ...:     if i == 0:
   ...:         break
   ...:     print(i)
   ...:
-4
-2
```

- S'il s'agit de passer outre le bloc d'instruction suivant, on utilisera l'instruction **continue**

```
In [1]: for i in range(-4,4,2):
   ...:     if i == 0:
   ...:         continue
   ...:     print(i)
   ...:
-4
-2
2
```

- S'il s'agit de passer outre le bloc d'instruction suivant, on utilisera l'instruction **continue**

```
In [1]: for i in range(-4,4,2):
   ...:     if i == 0:
   ...:         continue
   ...:     print(i)
   ...:
-4
-2
2
```

⚠ L'instruction `continue` est particulièrement utile afin d'éviter une trop grande imbrication d'instructions `if` successives.