Syllabus : Python et Raspberry Pi

Florence Blondiaux Jean-Martin Vlaeminck

06 août - 10 août 2018

Table des matières

Chapitre 1

A quoi s'attendre ?

Durant ce stage, nous allons aborder les points suivant :

1. Nous allons étudier le langage de programmation **PYTHON**. Ce langage va permettre aux étudiants d'interagir avec le raspberry pi et sera utile pour atteindre les objectifs fixés par le stage. Nous reprendrons les bases puis nous approfondirons la matière progressivement. De nombreux exercices accompagneront l'élève durant cet apprentissage.

2. Nous découvrirons également comment intéragir avec le **raspberry pi**. Nous verrons les bases de linux, le coeur du système d'exploitation du raspberry.

Vous disposez chacun d'un raspberry pi permettant de résoudre les exercices proposés. Cependant n'hésitez pas à dialoguer avec vos voisins concernant votre solution. En informatique, nous appelons cette technique le **pair programming**. Une telle méthode de travail permet à la fois de rendre l'apprentissage plus agréable et plus efficace.

Syllabus

Ce syllabus est destiné à des élèves de 12 à 18 ans assistant aux stages d'été organisé par Technofutur TIC. Ce syllabus est un complément aux slides et aux discussions proposées durant les séances de ce stage. Il doit être complété par l'élève selon ses notes personnelles et les exercices réalisés durant la semaine.

Le syllabus a été écrit sous forme d'activités. Chaque activité peut être vue comme un exercice permettant de découvrir la matière. Cela permet d'apprendre la théorie de façon

ludique et pratique.

1.1 Plan de la semaine

Ce plan est donné à titre informatif, il pourra être modifié selon les circonstances et les envies du groupe.

Lundi	Mardi	Mercredi	Jeudi	Vendredi
Introduction au Raspberry, Premier programme	Python Types Fonctions	Python Conditions Boucles	Coder un jeu : PyGame	Station Météo

Chapitre 2

Présentation de Python et installation des outils

2.1 Présentation de Python

Python est un langage de programmation inventé par Guido van Rossum en 1990. Son nom est une référence à la série télévisée Monty Python, dont van Rossum était fan.

Au fil des années, le langage a évolué. En 2000, on voit apparaître `Python 2.0`, et en 2008, `Python 3.0`. Actuellement, deux version de Python coexistent : `Python 2.7`, et `Python 3.6`. Nous utiliserons dans ce cours la dernière version.

Python est un langage très utilisé. Sa syntaxe simple en fait un des langages les plus appréciés pour écrire des *scripts*. Il est aussi utilisé dans le monde scientifique, dans des domaines tels que la science des données et l'intelligence artificielle.

Dans ce cours, on utilisera le Python pour écrire des programmes, qu'on pourra exécuter sur un Raspberry Pi.

2.2 Qu'est-ce qu'un programme ?

Les ordinateurs sont des outils très puissants, pour peu qu'on arrive à leur communiquer ce qu'on aimerait obtenir. On entend souvent dire qu'un ordinateur ne comprend qu'un langage très basique : le langage binaire, une suite de 0 et de 1. Du point de vue des microprocesseurs, on peut voir ça ainsi : soit le courant passe, soit il ne passe pas. En combinant ces deux états d'une manière bien précise, on peut effectuer des calculs.

Cependant, écrire un quelconque programme directement en binaire, c'est quasiment mission impossible, même pour un programme très basique. Il faut passer par une étape de traduction, et c'est ici que ça devient intéressant ! L'idée générale est d'écrire du code dans un langage de programmation, et de le transformer en langage machine/binaire.

Il y a deux manières de traduire du code.

— *La compilation :* On convertit immédiatement le code en langage machine. L'avantage, c'est que l'ordinateur comprend directement. Par contre, il faudra recommencer cette étape pour chaque système d'exploitation différents (Windows, macOS, Linux, ...).

— *L'interprétation :* on va ici passer par un intermédiaire : *l'interpréteur*. C'est un programme qui lit et traduit le code en temps réel. C'est un processus plus lent, mais qui

offre un grand avantage : on ne doit plus faire plusieurs versions pour chaque système d'exploitation, puisque c'est l'interpréteur qui se charge de la traduction ! On dit d'un tel langage qu'il est *multi-plateformes*.

Il existe une autre manière de classer les langages de programmation. Au plus ils ressemblent à du langage machine, au plus ils sont *bas niveau*. Au contraire, au plus un langage de programmation se rapproche du langage humain, au plus il est *haut-niveau*.

Le Python est un langage interprété, et haut-niveau, ce qui en fait un excellent choix pour débuter en programmation : on peut l'exécuter aisément sur de nombreux systèmes d'exploitation, et le code est assez lisible. Mais il ne faut pas s'y tromper : ce n'est pas parce qu'il est haut-niveau qu'il n'est pas puissant, bien au contraire !

2.3 Environnement de développement

Le Python est interprété, il nous faut donc un interpréteur ! Il s'agit simplement d'un programme qui va lire notre code, et exécuter en temps réel les instructions qu'on lui donne.

Théoriquement, c'est le seul outil dont on a réellement besoin. Pratiquement, un développeur s'entoure d'une suite logicielle qui lui rend la vie bien plus facile. On utilisera énormément un *éditeur de code*. C'est un éditeur de texte, à la manière de Microsoft Word, mais spécialisé dans un ou plusieurs langages de programmation. Il va notamment permettre de colorer le code pour le lire plus facilement, indenter automatiquement les lignes, les numéroter, etc.

On peut tout-à-fait utiliser l'éditeur de code et l'interpréteur séparément, mais le plus souvent on essaye de combiner les deux. On va utiliser ce qu'on appelle un evironnement de développement, abrégé IDE (pour Integrated Development Environment en anglais). Pour ce cours, nous avons choisi *Spyder*.

2.3.1 Installer Spyder

Les instructions suivantes permettent d'installer l'IDE *Spyder*.

1. Se rendre sur `https://www.continuum.io/downloads`.
2. Télécharger l'installateur et suivre les instructions d'installation de celui-ci.
3. Une fois l'installation terminée, lancer le programme *Anaconda Navigator*.
4. Depuis l'écran qui apparaît, lancer *Spyder*.

2.3.2 Présentation de l'interface

L'interface est simple à prendre en main. On retrouve 3 panneaux que nous utiliserons.

1. A gauche, l'éditeur de code. C'est ici que nous écrirons le code source.
2. En haut à droite, l'explorateur de variables. Probablement plus utile au début de l'apprentissage, on peut y voir les variables déclarées, et leur type.
3. En bas à droit, la console. C'est ici que s'afficheront les résultats de nos programmes. On peut aussi directement y taper du code.

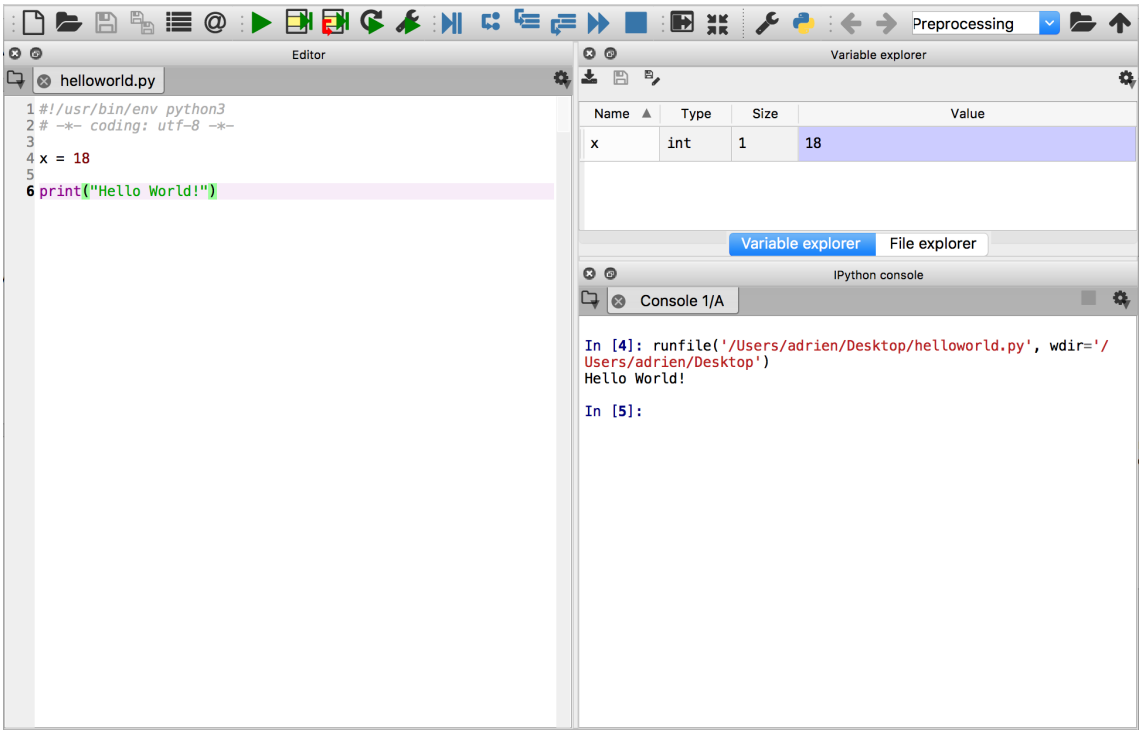


FIGURE 2.1 – Interface de Spyder

Tout en haut, on trouve la barre d'outils. Les trois boutons de gauche servent respectivement à créer un fichier, ouvrir un fichier, et sauver le fichier. Enfin, la flèche verte permet d'exécuter le programme. Concrètement, l'interpréteur Python sera lancé, il lira le contenu de l'éditeur de code, et affichera le résultat dans la console.

2.4 Premier programme

Pour vérifier que l'installation s'est déroulée correctement et que vous disposez de tous les outils nécessaires, écrivons donc un premier programme !

Ouvrez Spyder, et créez un nouveau fichier au besoin en cliquant sur le premier bouton à gauche de la barre d'outils.

Dans l'éditeur de texte, copiez-collez le bout de code suivant :

```
print("Tout fonctionne!")
```

Ensuite, lancez l'interpréteur en cliquant sur la flèche verte. Spyder vous demandera peut-être d'enregistrer le fichier. Nous vous invitons à enregistrer tous les programmes que nous écrirons cette semaine dans un dossier distinct.

Regardez à droite dans la console : une ligne de texte s'est affichée. Vous êtes maintenant normalement prêts à démarrer l'apprentissage du Python !

Chapitre 3

Programmer en Python

Maintenant que vous savez écrire un programme ainsi que l'exécuter et que vous avez écrit votre premier code, nous pouvons passer aux concepts de base de PYTHON.

3.1 Les bonnes pratiques du programmeur

1. Pas plus d'une instruction par ligne
2. Aérer son code
3. Faire *très* attention à l'indentation du texte (surtout en python).
4. Commenter son code !
5. Spécifier ses fonctions

Ces 5 concepts ne vous parlent peut-être pas encore, mais vous seront familiers très bientôt. Ces règles permettent de différencier un bon programme d'un mauvais et ne s'arrêtent pas à PYTHON mais sont utilisées dans quasiment tous les langages de programmation. Nous vous encourageons donc vivement à vous en imprégner le plus rapidement possible.

3.2 Affichage de texte

L'instruction la plus simple est l'affichage de texte dans la console. Comme vous l'aurez deviné grâce à votre fonction test, l'affichage se fait grâce à l'instruction `print`.

```
print("Le texte que je veux imprimer")
```

Il est préférable de ne pas mettre d'accents quand on imprime un texte dans la console, parce que toutes les configurations ne supportent pas leur affichage.

Pour effectuer un retour à la ligne, on peut terminer le texte par une suite de caractères spéciale : \n.

Notez déjà qu'on peut imprimer autre chose que du texte brut : ce sont des variables, dont on parlera dans le chapitre suivant.

3.3 Commentaires

Les commentaires sont des indications que le programmeur donne à quiconque voudrait lire son code. Les parties de texte commentées sont totalement ignorées par l'ordinateur lors de l'exécution d'un programme et, ainsi, n'affecte pas le code. En PYTHON, on commente une phrase en la précédant du symbole # et on commente un texte en le précédant de """ et en le terminant avec """ .

```
print "Hello"
#Cette phrase est ignoree
print "World" #Cette phrase est ignoree
"""Toute
cette
phrase
est
ignoree"""
```

3.4 Variables

Qu'est ce qu'une variable ? Les variables sont des symboles qui associent un nom à une valeur. La valeur est stockée dans la mémoire de l'ordinateur et le nom permet d'y accéder et/ou de modifier la valeur en tout temps. Une variable possède toujours un **type**. Ce dernier renseigne sur le "type" de la valeur stockée dans la variable.

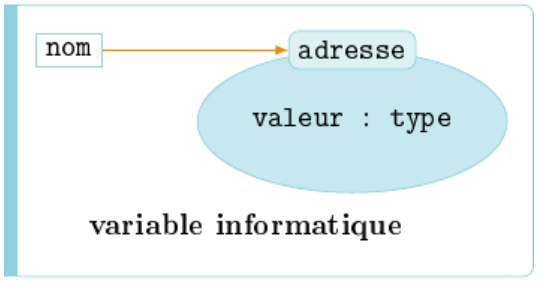


FIGURE 3.1 – Une variable

En PYTHON, les noms de variables ne peuvent pas commencer par un chiffre, ni contenir d'espaces, ni contenir d'accents.

Listing 3.1 – déclaration de variables

```
nom_de_variable = valeur
#Exemples de variables correctes
variable1 = 1 #Une variable dont le nom est variable1 et la valeur est 1
b = "salut"
variable68452 = 0.50
variable_au_nom_tres_long = True
variable_lettre = 'a'
```

3.4.1 Taille

Les variables PYTHON représentent une zone mémoire de l'ordinateur qui va stocker une certaine information. La taille d'une zone mémoire correspond à un ensemble de bits.

- 1 *bit* est le plus petit bout de donnée que l'on puisse stocker en mémoire. Il contient soit 1, soit 0.
- 1 octet (ou *byte* en anglais) correspond à 8 bits
- 1 kilooctet (Ko) correspond à 1024 octets
- 1 mégaoctet (Mo) correspond à 1024 Ko
- 1 gigaoctet (Go) correspond à 1024 Mo

3.4.2 Type natif

Python contient un ensemble de types natifs qui sont les types de données de **base**. Les types sont implicites lors de la déclaration d'une variable.

Type	Description
`boolean`	Représente *True* (vrai) ou *False* (faux)
`str`	Représente une chaîne de caractères
`int`	Représente un entier
`float`	Représente un nombre à virgule

TABLE 3.1 – Type primitif

Listing 3.2 – Types de variables

```
niveau = 'E'  #variable de type str
chaises = 12  #variable de type int
sonActif = False  #variable de type boolean
prixJeu = 12.50  #variable de type float
totalVoitures = 3453454  #variable de type int
```

Notons qu'on peut se renseigner sur le type d'une variable avec l'instruction `type(variable)`

Listing 3.3 – Instruction type

```
#Exemple
variable1 = 'a'
variable2 = 2
variable3 = 12.50
print(type(variable1)) #Affiche "str"
print(type(variable2)) #Affiche "int"
print(type(variable3)) #Affiche "float"
```

3.4.3 Cast d'une variable

Le mot *cast* signifie qu'on impose un type à une variable. On impose un type à variable de la manière suivante :

Listing 3.4 – Cast de variables

```
nom_de_la_variable = type_que_je_veux(valeur)
#Exemples
a = int(1) # a est de type int
b = float(3) # b est de type float
c = float(4.5) # x est de type float
d = int(0.5) # Que se passe-t-il ?
```

Dans quels cas de figure est-ce utile de caster un nombre ? Principalement parce qu'une opération s'effectue sur les mêmes types de données. On ne peut pas additionner "5" (la chaîne de caractères 5, un `str` !) avec 2 (un `int` !). Ici, on devrait donc d'abord caster le `str` en `int` avant l'opération. Mais il faut être prudent : que se passe-t-il si la chaîne de caractères ne peut être convertie en entier ? Par exemple :

```
int("salut") # Erreur!!!
```

3.4.4 Affectation de variables

Si vous avez bon souvenir, il était dit au tout début de la section 3.4 qu'on pouvait changer la valeur d'une variable. Pour ce faire, on égalise la variable à une nouvelle valeur.

Listing 3.5 – Réassignation de variables

```
a = 2
print(a) #Affiche 2
a = 1
a = 4
print(a) #Affiche 4
a = "arbre" #Que se passe-t-il ?
print(a) #Affiche "arbre"
```

3.4.5 Opérations arithmétiques

On peut utiliser des opérateurs arithmétiques sur nos variables. Celles-ci vont être effectuées comme des opérations d'algèbre sur une calculatrice. Les opérations disponibles sont reprises dans le tableau ci-dessous

Opération	Description
+	Addition
-	Soustraction
*	Multiplication
/	Division
**	Puissance
%	Modulo

TABLE 3.2 – Opérations arithmétiques

```
a = 4 + 6 #a vaut 10
a = ((a + a)/2)+1 #a vaut 11
b = a / 2 #b vaut 5
c = a**2 + b #c vaut 11^2 + 5 = 121 + 5 = 126
```

3.4.6 Sucre syntaxique

On dit souvent que les programmeurs sont des fainéants, et pour cause : ils cherchent à se faciliter la vie, et à réduire la taille des instructions qu'il tapent à longueur de journée ! Ceux qui ont programmé les langages tel que PYTHON ont écrit des *sucres syntaxiques* pour faciliter l'écriture et la lisibilité des programmes.

`x = x + 1`	$\iff$	`x += 1`
`x = x + 1`	$\iff$	`x += 1`
`x = x + 2`	$\iff$	`x += 2`
`x = x - 2`	$\iff$	`x -= 2`
`x = x / 2`	$\iff$	`x /= 2`
`x = x * 2`	$\iff$	`x *= 2`

TABLE 3.3 – Quelques exemples de sucres syntaxiques

Exercices

1. Ecrivez un programme qui déclare les variables nécessaires pour effectuer le calcul suivant :

$$\begin{aligned} x &= 5 \\ y &= x * 2 \\ w &= (y + x)/2 \\ y &= y * x \end{aligned}$$

2. Quelles est la valeur des différentes variables après l'exécution de cette partie de code ?

```
x = 10
y = x * 2
w = y - 10
z = y / w
```

3.5 Le type `str`

Maintenant que nous connaissons les types natifs, on peut aller un peu plus loin. Il y a quelque chose de très important à retenir : **En Python, tout est objet !** Mêmes les types

de données natifs qu'on a vu précédemment sont des objets. Nous apprendrons plus tard à définir nos propres types de données, c'est-à-dire des **objets** qui ont été déclarés dans des **classes**. Dans cette section on va s'attarder sur un type de données qu'on a déjà rencontré : le `String` (ce qui veut dire *chaîne* en anglais).

Un String est le type qui représente un texte. Tout texte entouré de guillemets est considéré comme un `String`.

Listing 3.6 – type `String`

```
#Exemples de Strings
v1 = "Ceci est une phrase"
v2 = ""
v3 = "3" #Est un String et non un int car il y a des guillemets!
v4 = "42.5"
v5 = "qpdjnqpuifnsfvsnvsdv5s1vd1s5v15xv4s5d"
print(type(v1)) #Imprime str (ce qui correspond a String)
```

3.5.1 Opérations sur les `String`

Chaque lettre du String est numérotée de par ordre croissant (commençant par zéro). Ainsi le String "Hello" possède la lettre *H* en 0, la lettre *e* en 1, etc... On accède à une lettre d'un `String` de la manière suivante :

Listing 3.7 – Accès à String

```
nom_du_string[numero_de_la_lettre] # Accede a la lettre d'un String
#Exemples
my_string = "Bonjour"
lettre_1 = my_string[0] #lettre_1 vaut 'B'
lettre_4 = my_string[3] #lettre_4 vaut 'j'
```

Attention à ne pas oublier que les numéros des lettres commencent à 0 et non à 1 ! C'est une faute très courante au début !

Des opérations courantes sur les `String` sont présentées dans le tableau ci-dessous.

Notez la syntaxe particulière des instructions lower et upper : le nom de la variable, suivi d'un point, suivi du nom de l'opération. Pour l'instant, utilisez-les telles quelles, on en reparlera dans le chapitre sur les classes et objets.

Opération	Description
len(nom_string)	Calcule la longueur d'un String
nom_string.lower()	Retourne le string en minuscule
nom_string.upper()	Retourne le string en majuscule
str(nom_variable)	Cast en String
nom_string1+nom_string2	Retourne les 2 String concaténées

3.5.2 Exercices

1. **Exercice 1 :** Concaténer les `Strings` "Hakuna Matata", "Mais quelle ", "phrase magnifique !" et affichez le résultat en majuscules ainsi que sa longueur.

2. **Exercice 2 :** Caster les Strings a = "2" et b = "125.2" en nombre pour calculer a/b. [1]

3.6 Interaction avec l'utilisateur

Pour l'instant, nous nous limitons à des programmes dans lesquels toutes les valeurs sont définies avant l'exécution du programme. Ce qui nous limite assez fortement dans le genre de problèmes que l'on peut résoudre. Nous allons ici apprendre à demander à l'utilisateur de rentrer de l'information.

On peut voir cette pratique comme lorsque Google vous demande un mot-clé à rechercher, lorsqu'un jeu vidéo vous demande un nom pour votre héros, etc...

La fonction qui permet cette opération s'appelle `input`. Par défaut, cette fonction convertit toutes les données entrées par l'utilisateur en `String` mais on peut imposer de recevoir un type en particulier en *castant* la réponse de l'utilisateur dans le type désiré.

Listing 3.8 – fonction `input`

```
answer = raw_input("ce_qu_on_affiche_a_l_utilisateur")
#Exemple
nom = input("Quel est votre nom ?")
prenom = input("Quel est votre prenom ?")
age = int(input("Quel est votre age ?"))
print "Bonjour " + nom + " " + prenom + " vous etes age de " + str(age) + " ans!"
```

3.6.1 Exercice :

Faites un convertisseur qui demande à l'utilisateur des kilomètres et qui lui renvoie ce chiffre converti en miles en le remerciant d'avoir utilisé votre programme.

Aide : 1 km = 0.621 mile

3.7 Les conditions

Nous effectuons chaque jour certaines actions plutôt que d'autres : "si je suis debout tôt, alors je vais chercher le petit déjeuner". En informatique, il est possible de faire raisonner une ordinateur d'une manière assez similaire [2].

Typiquement, les programmes informatiques utilisent des expressions dont on évalue la valeur selon certains opérateurs pour savoir si elle vaut **True** ou **False**.

L'action effectuée par l'ordinateur sera différente en fonction de la valeur (True ou False) de l'expression.

3.7.1 Outils de comparaison

Voici les opérations de comparaison disponibles en PYTHON :

1. a et b doivent être impérativement définis en String au départ !
2. Pour les conditions hein, pas pour le petit déjeuner !

Opération	Description
==	Egalité
!=	Différent
>	Plus grand
>=	Plus grand ou égal
<	Plus petit
<=	Plus petit ou égal

TABLE 3.4 – Comparaison

Listing 3.9 – Quelques comparaisons en python

```
a = 5 > 4 # a vaut True
b = 0.5 >= 1 # b vaut False
c = 1 > "salut" # Que se passe t il ?
```

3.7.2 Opérateurs logiques

Ce n'est pas tout ! On peut encore combiner ces expressions avec des opérateurs logiques.

Typiquement,on pourrait vouloir que deux conditions soient respectées, qu'au moins l'une des deux soit respectée ou qu'une condition ne soit pas respectée. Ceci est possible via les opérateurs ET, OU et NON.

Table logique Chaque opérateur logique peut être défini par une table logique :

a \ b	True	False
True	True	True
False	True	False

(a) OU

a \ b	True	False
True	True	False
False	False	False

(b) ET

a	
True	False
False	True

(c) NON

TABLE 3.5 – Table logique des opérateurs logiques

En python En PYTHON, les opérateurs sont représentés à l'aide de `and`, `or`, `not`.

a OU b	a or b
a ET b	a and b
NON a	not a

TABLE 3.6 – Opérateurs logiques en PYTHON

Combinaisons

Les opérateurs logiques ainsi expressions conditionnelles peuvent être combinés à l'infini pour former des expressions plus complexes. Cependant, comme pour les opérations

mathématiques, il existe un ordre de priorité : d'abord `not`, puis `and`, puis `or`. Pour éviter les ambiguïtés et améliorer la lisibilité d'une expression complexe, on peut (et c'est conseillé !) y mettre des parenthèses.

Listing 3.10 – Expressions booléennes

```
a = True
b = False
c = a or b #c est vrai
d = a and b #d est faux
e = not b and a #e est vrai
f = (not (a and b)) or (not ((not d) and (a or e))) #f vaut True
```

Exercices : Evaluez les expressions suivantes :

Listing 3.11 – Exercices d'expressions booléennes

```
bool_one = False or not True and True
bool_two = False and not True or True
bool_three = True and not (False or False)
bool_four = not not True or False or not True
bool_five = False or not (True and True)
```

Le `if`

On peut représenter l'ensemble des exécutions possibles d'un programme comme un arbre de décision. Selon la valeur de certaines expressions, l'exécution du programme va prendre une direction ou l'autre...

En résumé, certaines parties du code sont exécutées sous certaines conditions !

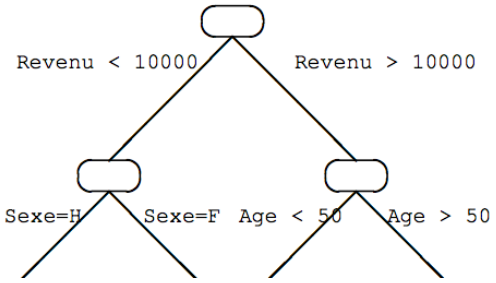


FIGURE 3.2 – Arbre binaire de décision

Afin de représenter ce choix, PYTHON contient la clause `if` suivi d'une une expression booléenne et de `:`. Si la condition est vraie, le programme exécute toutes les instructions *indentées*[3] qui suivent. Dans le cas contraire, ces instructions ne sont pas exécutées.

Listing 3.12 – l'instruction if

```
a = 10
if a > 5:
```

3. Une expression indentée est une expression précédée d'une tabulation

```
    print("Cette instruction est executee")
print("fin du code")
```

Il est aussi possible de donner un choix alternatif grâce à l'instruction `else:`. Si l'expression dans le `if` est `False`, le programme exécute alors le code *indenté* qui suit le `else`.

```
a = 10
if a < 5:
    print("Cette instruction n'est pas executee")
else:
    print("Cette instruction est executee")
print("fin du code")
```

3.7.3 Le `elif`

Vous pouvez construire des clauses `if` plus complexes en rajoutant des `elif`. Attention, à la première condition vraie rencontrée, on quitte la condition ! Les expressions suivantes ne sont même pas évaluées.

Listing 3.13 – utilisation de elif

```
x = 200
if x > 10:
    print("A")
elif x > 100:
    print("B")
else:
    print("C")
# Seul 'A' est imprime, pas 'B' !!!
```

La clause **elif** permet d'augmenter l'embranchement (c'est-à-dire le nombre de choix possibles) de notre arbre de décisions !

3.7.4 Code mort

Lorsque vous utilisez des conditions, faites attention à ne pas créer du *code mort*, c'est-à-dire du code qui ne sera jamais exécuté.

Listing 3.14 – Exemple de code mort

```
if False:
    print("Ce code n'est jamais execute!")
```

3.7.5 Exercice

Écrivez un code qui demande à l'utilisateur d'entrer un chiffre et imprime "Félicitations, votre chiffre est un multiple de 7" si le chiffre est divisible par 7. Et "Désolé, votre chiffre n'est pas un multiple de 7" sinon.

3.8 Boucles

3.8.1 La boucle `while`

La boucle `while` est un outil très puissant en programmation. En français, cela correspond à dire *tant que cette condition est vraie, j'exécute le code suivant*. En informatique, cela s'écrit avec le mot `while` suivi d'une expression booléenne et de `:`.

Fonctionnement

1. On évalue l'expression booléenne
2. Si l'expression est **False** : on ignore le code indenté qui suit le **while**.
3. Si l'expression est **True** : on exécute le code indenté qui suit **while** et retour à l'étape 1.

Listing 3.15 – Exemple de boucle while

```
"""Affiche tous les nombres de la table de 9 plus petits que 100"""
i = 0
while i <= 100:
        print("i vaut " + str(i))
        i = i+9 # Mise a jour de i
print("Fini")
```

Attention à la boucle infinie !

Comme vous avez pu le voir, la (ou les) variable de l'expression booléenne est mise à jour dans la boucle `while`. Dans le cas contraire, on ne sortira jamais de la boucle puisque la valeur de la condition de celle-ci ne sera jamais changée. Cet oubli de mise à jour crée ce qu'on appelle des boucles infinies et sont une erreur très courante. Ces dernières, en fonction de l'endroit où elle surgissent, peuvent faire planter votre programme.

Listing 3.16 – Exemple de boucle infinie

```
i = 0
while i < 10:
        print "Je suis dans la boucle"
print "Ce message ne s'affichera jamais"
```

3.8.2 La boucle `for`

La boucle `for` est, elle aussi, un réel passe-partout[4] du programmeur. En PYTHON, elle permet d'appliquer un code (*indenté*) à chaque élément d'une structure de données[5]. Elle commence toujours par le premier élément de la structure et finit par le dernier.

4. Attention pas le même que celui de Fort-Boyard !
5. Par exemple un String ou une liste

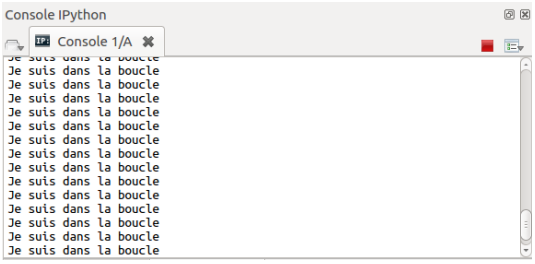


FIGURE 3.3 – Le programme ne s'arrête jamais

Pour ce faire, on écrit : for i in ma_structure. Ici i (notez qu'on pourrait très bien choisir un autre nom de variable) est appelé *itérateur*, c'est-à-dire que c'est lui qui prendra successivement la première valeur de la structure de données, puis la seconde, et ainsi de suite.

Listing 3.17 – la boucle for

```
"""Utilise une boucle for pour imprimer un string"""
s = "Hello World"
for i in s:
        print i
print "done"
```

3.8.3 Le `break`

On peut, si on le désire, sortir à tout moment d'une boucle. Pour ce faire, on utilise le mot clé **break** à l'endroit où l'ont veut sortir.

```
i = 0
while(i<100):
        if i != 50:
                print i
                i = i+1
        else:
                break
```

Attention : un `break` est considéré comme une sortie "sale" d'une boucle. On ne l'utilise donc que lorsque cela est absolument nécessaire !

3.8.4 Exercices :

1. **Exercice 1 :** Améliorez votre convertisseur de l'exercice 3.6.1 pour qu'il convertisse les unités de l'utilisateur jusqu'à ce que celui entre le mot "stop".
2. **Exercice 2 :** Implémentez un programme qui prend un texte en entrée (fourni par l'utilisateur) et imprime chaque voyelle du texte. (Essayez avec un texte très long trouvé sur Internet...)
3. **Exercice 3 :** Implémentez un programme qui prend un texte en entrée (fourni par l'utilisateur) et qui imprime **True** si le texte contient un palindrome et **False** sinon.

(**Aide :** un palindrome est un mot qui peut se lire dans les 2 sens. *Exemple* : kayak, abcba, ...)

3.9 Fonctions

Une fonction est un outil informatique qui encapsule une portion de code (une séquence d'instructions) et qui effectue une tâche bien spécifique (calcul, avertissement, affichage, ...).

L'utilisation d'une fonction se nomme *appel d'une fonction*. Chaque appel de fonction contient des **arguments** qui sont les informations avec laquelle la fonction va travailler.

Ce concept n'est pas nouveau, vous connaissez déjà plein de fonctions ! Par exemple, la fonction `print()` qui prend en argument un `str` et l'imprime dans la console.

Une fonction peut renvoyer une valeur, résultat de son exécution. Cette valeur s'appelle **la valeur de retour**. Autre exemple, la fonction `input` qui prend en argument un `str` (qui est la question posée à l'utilisateur) et retourne un `str` (qui est la réponse de l'utilisateur).

3.9.1 Créer ses propres fonctions

Utiliser des fonctions toutes faites c'est bien. En créer soi-même, c'est mieux ! En PYTHON, la création de fonction se fait par le code `def nom_de_la_fonction(argument1, argument2, ...):`. Le code indenté qui suit cette instruction est le *corps* de la fonction et correspond aux instructions que celle-ci exécutera à chaque appel. Pour renvoyer une valeur, on utilise le mot clé `return` suivi de la valeur qu'on souhaite renvoyer. Enfin, pour appeler une fonction, on utilise le nom de la fonction suivi des arguments qu'on veut lui passer entre parenthèses.

Listing 3.18 – Exemple 1 : La fonction addition

```
def addition(arg1, arg2): #Definition de la fonction addition
        return arg1+arg2
print(addition(3,4)) #Affiche 7
```

Listing 3.19 – Exemple 2 : La fonction reverse

```
def reverse(my_string): #Definition de la fonction qui inverse un String
        toReturn = ""
        for i in my_string:
                toReturn = i + toReturn
        return toReturn
print(reverse("esarhp eugnol zessa enU")) #Affiche Une assez longue phrase
```

3.9.2 Exercices :

1. Créez les fonctions `soustraction(arg1, arg2)`, `multiplication(arg1, arg2)`, `division(arg1, arg2)`, et `power5(arg)` qui, comme leurs noms l'indique font res-

pectivement les opérations de soustraction, multiplication, division, mise à la puissance de 5.

2. Créez la fonction string_length (my_string) qui affiche la longueur du texte en argument.

3.10 Portée des variables

Les variables sont déclarées dans leur "bloc". Une variable déclarée dans une fonction est distincte des autres variables déclarées en dehors du corps de la fonction, et n'est plus accessible quand la fonction se termine. On appelle ce concept la **portée des variables**.

Dans l'exemple ci-dessous, on déclare deux variables appelées `x`, **mais ce ne sont pas les mêmes !** On s'en rend compte en exécutant le code : le `x` déclaré hors de la fonction n'est pas modifié après l'appel de la fonction.

```
x = 10

def fonction():
    x = 20
    print(x)

fonction() # Imprime 20
print(x) # Imprime 10
```

On peut aussi utiliser utiliser le mot-clef `global` pour élargir la portée d'une variable, et la rendre globale. Ainsi, dans l'exemple suivant, quand Python rencontre le mot-clef `global` dans la fonction, il va regarder non seulement si `x` a été déclaré dans la fonction, mais aussi dans les "blocs" supérieurs. Dans ce cas, il trouve `x` déclaré à un niveau supérieur, et la fonction y a pleinement accès et peut la modifier.

```
x = 10
print(x) # Imprime 10

def fonction():
    global x
    x += 10
    print(x)

fonction() # Imprime 20
fonction() # Imprime 30
print(x) # Imprime 30
```

Les variables globales paraissent pratiques : on la déclare une fois, et on peut l'utiliser partout. Cependant, dans un grand programme, elles deviennent bien vite une source de confusion, et ce n'est pas considéré comme une bonne pratique. On vous conseille donc de les utiliser avec modération.

3.11 Modules

On a déjà vu quelques fonctions inclues de base dans PYTHON. Mais tout n'est pas présent, loin de là : un programme peut être utilisé pour afficher une interface graphique, travailler avec des images, accéder au web, faire des calculs mathématiques avancés, lire de l'audio ou de la vidéo, faire un jeu 3D... Ces quelques exemples montrent à quel point on peut se servir d'un seul langage pour faire à peu près tout et n'importe quoi !

Alors comment commencer ? Chaque développeur pourrait bien sûr écrire de zéro chaque fonction donc il a besoin. Mais ce serait un travail immense, et on risque de faire des erreurs. Ainsi, les développeurs se sont organisés : ils ont regroupé des fonctions déjà écrites dans ce qu'on appelle des *modules*. Il suffit d'importer un modules pour avoir accès à toutes ses fonctions !

Voici comment on importe un module, par exemple pour avoir accès à des fonctions mathématiques plus avancées :

```
import math # importation du module math

x = math.ceil(8.728)
print(x) # imprime le premier entier superieur ou egal au parametre
```

Pour utiliser les fonctions du module, on écrit module.fonction().

Et si on n'a pas envie de taper à chaque fois le nom du module ? On peut importer directement la fonction !

```
from math import ceil # importe juste la fonction ceil

x = ceil(8.728)
print(x) # imprime le premier entier superieur ou egal au parametre
```

On peut aussi importer toutes les fonctions avec la ligne from module import $*$.

3.12 Listes

Une liste est une structure de données permettant de regrouper des données de manière à pouvoir y accéder librement.

Listing 3.20 – Exemples de listes

```
jeuxVideos = ['HeartStone', 'LoL', 'WoW', 'ClubPenguin']
chiffres = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Accéder à un élément d'une liste se fait de la manière que pour un `String` (on commence à **0**). En effet, un `String` est en réalité une liste de caractères !

Listing 3.21 – Accès à une liste

```
chiffres = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

chiffre0 = chiffres[0]      #chiffre0 vaut 0
chiffre3 = chiffres[3]      #chiffre3 vaut 3
chiffres3a6 = chiffres[3:7] #chiffres3a6 vaut [3 4 5 6]
```

On peut modifier une liste de plusieurs façons : changer, supprimer, ajouter un ou plusieurs éléments.

Listing 3.22 – Modification de liste

```
my_list = ['a', 'b', 'c', 'd', 'e']

#Modification
my_list[3] = 'f'    #['a', 'b', 'c', 'f', 'e']
#Suppression
del my_list[3]      #['a', 'b', 'c', 'e']
my_list.remove('e') #['a', 'b', 'c']
#Ajout
my_list.append('d') #['a', 'b', 'c', 'd']
my_list + ['e']     #['a', 'b', 'c', 'd', 'e']
```

Les listes sont également assorties de quelques opérations très pratiques et souvent employées.

Listing 3.23 – Opérations sur les listes

```
my_list = ['a', 'b', 'c', 'd', 'e']

#Longueur de la liste
longueur = len(my_list)  #longueur vaut 5
#Presence d un element
'd' in my_list #vaut True
'f' in my_list #vaut False
#Parcourir une liste
for x in my_list : print (x) #imprime abcde
#Retourner une liste
my_list.reverse() #my_list vaut ['e', 'd', 'c', 'b', 'a']
#Mettre une liste dans l ordre croissant
my_list.sort() #my_list vaut ['a', 'b', 'c', 'd', 'e']
```

3.13 Classes et objets

Avant d'avancer plus en profondeur, il est utile de définir quelques mots de vocabulaire.

— Une classe est une définition d'un concept. Elle peut être agrémentée d'attributs et de fonctions pour la caractériser.

— Un objet est une instance d'une classe, c'est-à-dire que son comportement est défini par la classe.

Si l'on prend l'exemple d'un véhicule, son plan de fabrication et de fonctionnement correspond à la classe. Une voiture concrète, qui possède ses propres caractéristiques, correspond à un objet de cette classe. Et on peut construire plusieurs voitures à partir d'un seul plan, et les personnaliser en changeant ses options (attributs !).

Une classe possède des attributs qui sont eux-mêmes des objets (de différents types), et des méthodes qui sont des fonctions qui s'appliquent sur les objets de cette classe. Le mot-clef

`self` permet de référencer l'objet sur lequel on est en train de travailler. Il est implicitement passé en argument à toutes les méthodes. Le constructeur est une méthode particulière de la classe : il permet de créer un nouvel objet.

Listing 3.24 – Exemple de classe

```
class Vehicule:

    """Attributs"""

    #Valeurs par defaut
    nombreRoues = 4
    nombrePortes = 5
    nombrePlaces = 5
    couleur = 'noir'
    kilometres = 0

    """Methodes"""

    #Constructeur, permet de creer un objet vehicule avec un certain prix
    def __init__(self, prix):
        self.prix = prix

    #Modificateur, permet de changer la couleur
    def setColor(self, couleur):
        self.couleur = couleur

    #Calculateur de TVA (taxe de 21%)
    def tva(self):
        return self.prix * 0.21

    #Faire rouler le vehicule
    def rouler(self, distance)
        self.kilometres += distance
```

On peut ensuite utiliser cette classe pour créer une voiture, et s'en servir.

Listing 3.25 – Exemple de classe

```
voiture = Vehicule(10000)              #Cree une voiture au prix de 10 000 euros
facture = voiture.prix + voiture.tva() #Total a payer
voiture.rouler(200)                    #Roule sur 200km
voiture.setColor('red')                #Peint la voiture en rouge
voiture.rouler(150)                    #Roule sur 150km
print(voiture.distance)                #Affiche 350
```

Chapitre 4

Conclusion

Durant cette semaine, vous avez appris les bases du Python : variables, conditions, boucles, fonctions, listes, etc. On a donné un introduction à la programmation orienté-objet. Libre à vous désormais de poursuivre cet apprentissage ! Le Python est un des langages les plus utilisés, ce qui veut dire qu'il y a aussi énormément de ressources en ligne pour apprendre par soi-même.

On pense notamment à l'excellent OpenClassrooms [1], ou à ce tutoriel sur Développez.com [2].

Vous avez aussi découvert le Raspberry Pi. Étonnant petit ordinateur, il n'en est pas moins capable ! D'autres curieux tels que vous ont utilisé leur Raspberry Pi dans des projets surprenants ! Certains sont listés sur Instructables [3], avec des explications pas-à-pas.

Enfin, nous espérons que vous avez apprécié ce stage, et que vous continuerez à explorer les infinies possibilités de l'informatique !

1. https://openclassrooms.com/courses/apprenez-a-programmer-en-python
2. https://python.developpez.com/cours/apprendre-python3/
3. http://www.instructables.com/id/Raspberry-Pi-Projects/

Python 3 Cheat Sheet

Latest version on :
https://perso.limsi.fr/pointal/python:memento

Base Types

integer, float, boolean, string, bytes

```
int 783  0 -192  0b010 0o642 0xF3
          zero   binary  octal  hexa
float 9.23 0.0  -1.7e-6
                      ×10^-6
bool True False
str "One\nTwo"   escaped new line
'I\'m'   escaped '
"""X\tY\tZ
1\t2\t3"""   escaped tab
bytes b"toto\xfe\775"
        hexadecimal  octal
```

Multiline string:

immutables

Container Types

- **ordered sequences**, fast index access, repeatable values

```
list [1,5,9]   ["x",11,8.9]   ["mot"]   []
tuple (1,5,9)   11,"y",7.4   ("mot",)   ()
```

Non modifiable values (immutables) — *expression with only comas →* `tuple`

```
str bytes   (ordered sequences of chars / bytes)
""   b""
```

- **key containers**, no *a priori* order, fast key access, each key is unique

```
dictionary dict {"key":"value"}   dict(a=3,b=4,k="v")   {}
(key/value associations) {1:"one",3:"three",2:"two",3.14:"π"}
collection set {"key1","key2"}   {1,9,3,0}   set()
```

keys=hashable values (base types, immutables…) `frozenset` *immutable set* *empty*

Identifiers

for variables, functions, modules, classes… names

`a…zA…Z_` followed by `a…zA…Z_0…9`

- diacritics allowed but should be avoided
- language keywords forbidden
- lower/UPPER case discrimination

```
☺ a toto x7 y_max BigOne
☹ 8y and for
```

Variables assignment

`=`

assignment ⇔ **binding** of a *name* with a *value*

1) evaluation of right side expression value
2) assignment in order with left side names

```
x=1.2+8+sin(y)
a=b=c=0   assignment to same value
y,z,r=9.2,-7.6,0   multiple assignments
a,b=b,a   values swap
a,*b=seq
*a,b=seq   unpacking of sequence in item and list
x+=3   increment ⇔ x=x+3
x-=2   decrement ⇔ x=x-2
x=None   « undefined » constant value
del x   remove name x
```

and `*=` `/=` `%=` …

Conversions

`type(`*expression*`)`

```
int("15") → 15
int("3f",16) → 63   can specify integer number base in 2nd parameter
int(15.56) → 15   truncate decimal part
float("-11.24e8") → -1124000000.0
round(15.56,1)→15.6   rounding to 1 decimal (0 decimal → integer number)
bool(x)  False for null x, empty container x , None or False x ; True for other x
str(x)→"…"   representation string of x for display (cf. formatting on the back)
chr(64)→'@'  ord('@')→64   code ↔ char
repr(x)→"…"   literal representation string of x
bytes([72,9,64]) → b'H\t@'
list("abc") → ['a','b','c']
dict([(3,"three"),(1,"one")]) → {1:'one',3:'three'}
set(["one","two"]) → {'one','two'}
```

separator `str` and *sequence of* `str` → *assembled* `str`

```
':'.join(['toto','12','pswd']) → 'toto:12:pswd'
```

`str` *splitted on whitespaces →* `list` of `str`

```
"words with  spaces".split() → ['words','with','spaces']
```

`str` *splitted on separator* `str` → `list` *of* `str`

```
"1,4,8,2".split(",") → ['1','4','8','2']
```

sequence of one type → `list` of another type *(via list comprehension)*

```
[int(x) for x in ('1','29','-3')] → [1,29,-3]
```

Sequence Containers Indexing

for lists, tuples, strings, bytes…

```
negative index   -5  -4  -3  -2  -1
positive index    0   1   2   3   4
lst=[10, 20, 30, 40, 50]
positive slice    0   1   2   3   4   5
negative slice   -5  -4  -3  -2  -1
```

Items count

`len(lst)→5`

index from 0
(here from 0 to 4)

Individual access to **items** via `lst[`*index*`]`

```
lst[0]→10   ⇒ first one
lst[-1]→50   ⇒ last one
lst[1]→20
lst[-2]→40
```

On mutable sequences (`list`), remove with `del lst[3]` *and modify with assignment* `lst[4]=25`

Access to **sub-sequences** via `lst[`*start slice*`:`*end slice*`:`*step*`]`

```
lst[:-1]→[10,20,30,40]   lst[::-1]→[50,40,30,20,10]   lst[1:3]→[20,30]   lst[:3]→[10,20,30]
lst[1:-1]→[20,30,40]   lst[::-2]→[50,30,10]   lst[-3:-1]→[30,40]   lst[3:]→[40,50]
lst[::2]→[10,30,50]   lst[:]→[10,20,30,40,50]   shallow copy of sequence
```

Missing slice indication → from start / up to end.
On mutable sequences (`list`), remove with `del lst[3:5]` *and modify with assignment* `lst[1:4]=[15,25]`

Boolean Logic

Comparisons : `<` `>` `<=` `>=` `==` `!=` (≤ ≥ = ≠)
(boolean results)

`a and b` logical and *both simultaneously*

`a or b` logical or *one or other or both*

pitfall : `and` *and* `or` *return value of* `a` *or of* `b` *(under shortcut evaluation).*
⇒ *ensure that* `a` *and* `b` *are booleans.*

`not a` logical not

`True` `False` True and False constants

Statements Blocks

parent statement:
indentation !
statement block 1…
parent statement:
statement block2…
next statement after block 1

configure editor to insert 4 spaces in place of an indentation tab.

Modules/Names Imports

module `truc` ⇔ *file* `truc.py`

```
from monmod import nom1,nom2 as fct
```

→*direct access to names, renaming with* `as`

```
import monmod
```

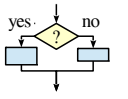
→*access via* `monmod.nom1` …

modules and packages searched in *python path* (cf `sys.path`)

Conditional Statement

statement block executed only if a condition is true

`if` *logical condition*`:`
statements block



Can go with several *elif*, *elif*... and only one final *else*. Only the block of first true condition is executed.

```
if age<=18:
    state="Kid"
elif age>65:
    state="Retired"
else:
    state="Active"
```

with a var `x`*:*

```
if bool(x)==True:  ⇔ if x:
if bool(x)==False: ⇔ if not x:
```

Maths

floating numbers… approximated values

Operators: `+` `-` `*` `/` `//` `%` `**`
Priority `(…)` × ÷ ↑ ↑ a^b
integer ÷ ÷ remainder

`@` → matrix × *python3.5+*`numpy`

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
pow(4,3)→64.0
```

usual order of operations

angles in radians

```
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
sqrt(81)→9.0   √
log(e**2)→2.0
ceil(12.5)→13
floor(12.5)→12
```

modules `math`, `statistics`, `random`, `decimal`, `fractions`, `numpy`, *etc. (cf. doc)*

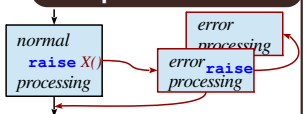
Exceptions on Errors

Signaling an error:

```
raise ExcClass(…)
```

Errors processing:

```
try:
    normal procesising block
except Exception as e:
    error processing block
```



`finally` *block for final processing in all cases.*

Conditional Loop Statement

*statements block executed **as long as** condition is true*

```
while logical condition:
    statements block
```


```
s = 0          # initializations before the loop
i = 1          # condition with a least one variable value (here i)
while i <= 100:
    s = s + i**2
    i = i + 1  # make condition variable change !
print("sum:",s)
```

Algo: $s = \sum_{i=1}^{i=100} i^2$

beware of infinite loops!

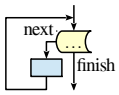
Loop Control

```
break      immediate exit
continue   next iteration
else block for normal loop exit.
```

Iterative Loop Statement

*statements block executed **for each** item of a container or iterator*

```
for var in sequence:
    statements block
```



Go over sequence's **values**

```
s = "Some text"   # initializations before the loop
cnt = 0
for c in s:       # c = loop variable, assignment managed by for statement
    if c == "e":
        cnt = cnt + 1
print("found",cnt,"'e'")
```

Algo: count number of `e` in the string.

loop on dict/set ⇔ loop on keys sequences
use *slices* to loop on a subset of a sequence

Go over sequence's **index**

- modify item at index
- access items around index (before / after)

```
lst = [11,18,9,12,23,4,17]
lost = []
for idx in range(len(lst)):
    val = lst[idx]
    if val > 15:
        lost.append(val)
        lst[idx] = 15
print("modif:",lst,"-lost:",lost)
```

Algo: limit values greater than 15, memorizing of lost values.

good habit : don't modify loop variable

Go simultaneously over sequence's **index** and **values**:

```
for idx,val in enumerate(lst):
```

Display

```
print("v=",3,"cm :",x,",",y+4)
```

items to display : literal values, variables, expressions

`print` options:

- `sep=" "` items separator, default space
- `end="\n"` end of print, default new line
- `file=sys.stdout` print to file, default standard output

Input

```
s = input("Instructions:")
```

`input` always returns a **string**, convert it to required type (cf. boxed *Conversions* on the other side).

Generic Operations on Containers

`len(c)` → items count
`min(c)` `max(c)` `sum(c)`
`sorted(c)` → `list` sorted *copy*
`val in c` → boolean, membership operator `in` (absence `not in`)
`enumerate(c)` → *iterator* on (index, value)
`zip(c1,c2…)` → *iterator* on tuples containing c_i items at same index
`all(c)` → `True` if **all** `c` items evaluated to true, else `False`
`any(c)` → `True` if **at least one** item of `c` evaluated true, else `False`

*Note: For dictionaries and sets, these operations use **keys**.*

*Specific to **ordered sequences containers** (lists, tuples, strings, bytes…)*

`reversed(c)` → *inversed iterator* `c*5`→ duplicate `c+c2`→ concatenate
`c.index(val)` → *position* `c.count(val)` → *events count*

```
import copy
copy.copy(c) → shallow copy of container
copy.deepcopy(c) → deep copy of container
```

Operations on Lists

modify original list

`lst.append(val)`	add item at end
`lst.extend(seq)`	add sequence of items at end
`lst.insert(idx,val)`	insert item at index
`lst.remove(val)`	remove first item with value *val*
`lst.pop([idx])` →*value*	remove & return item at index *idx* (default last)
`lst.sort()` `lst.reverse()`	sort / reverse liste *in place*

Operations on Dictionaries

```
d[key]=value        d.clear()
d[key] → value      del d[key]
d.update(d2)        update/add associations
d.keys()
d.values()          →iterable views on keys/values/associations
d.items()
d.pop(key[,default]) → value
d.popitem() → (key,value)
d.get(key[,default]) → value
d.setdefault(key[,default]) →value
```

Operations on Sets

Operators:

- `|` → union (vertical bar char)
- `&` → intersection
- `-` `^` → difference/symmetric diff.
- `<` `<=` `>` `>=` → inclusion relations

Operators also exist as methods.

```
s.update(s2)  s.copy()
s.add(key)    s.remove(key)
s.discard(key)  s.clear()
s.pop()
```

Files

storing data on disk, and reading it back

```
f = open("file.txt","w",encoding="utf8")
```

- file **variable** for operations
- **name** of file on disk (+path…)
- opening **mode**
  - `'r'` read
  - `'w'` write
  - `'a'` append
  - …`'+'` `'x'` `'b'` `'t'`
- **encoding** of chars for *text files*: `utf8` `ascii` `latin1` …

cf. modules `os`, `os.path` and `pathlib`

writing

```
f.write("coucou")
f.writelines(list of lines)
```

reading

read empty string if end of file

```
f.read([n])        → next chars
                   if n not specified, read up to end !
f.readlines([n])   → list of next lines
f.readline()       → next line
```

*text mode `t` by default (read/write `str`), possible binary mode `b` (read/write `bytes`). **Convert from/to required type** !*

`f.close()` dont forget to **close the file** after use !

`f.flush()` write cache
`f.truncate([size])` resize

reading/writing progress sequentially in the file, modifiable with:

`f.tell()` →*position*
`f.seek(position[,origin])`

Very common: opening with a guarded block (automatic closing) and reading loop on lines of a text file:

```
with open(…) as f:
    for line in f :
        # processing of line
```

Integer Sequences

`range([start,] end [,step])`

start default 0, *end* not included in sequence, *step* signed, default 1

`range(5)` → 0 1 2 3 4
`range(3,8)` → 3 4 5 6 7
`range(2,12,3)` → 2 5 8 11
`range(20,5,-5)` → 20 15 10
`range(len(seq))` → *sequence of index of values in seq*

range provides an immutable sequence of int constructed as needed

Function Definition

function name (identifier)
named parameters

```
def fct(x,y,z):
    """documentation"""
    # statements block, res computation, etc.
    return res
```

`fct`

result value of the call, if no computed result to return: `return None`

parameters and all variables of this block exist only *in* the block and *during* the function call (think of a "black box")

Advanced: `def fct(x,y,z,*args,a=3,b=5,**kwargs):`

**args variable positional arguments (→`tuple`), default values,*
***kwargs variable named arguments (→`dict`)*

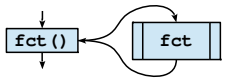
Function Call

```
r = fct(3,i+2,2*i)
```

storage/use of returned value
one argument per parameter

this is the use of function name *with parentheses* which does the call

*Advanced: *sequence **dict*



Operations on Strings

```
s.startswith(prefix[,start[,end]])   s.strip([chars])
s.endswith(suffix[,start[,end]])
s.count(sub[,start[,end]])           s.partition(sep) → (before,sep,after)
s.index(sub[,start[,end]])           s.find(sub[,start[,end]])
s.is…()  tests on chars categories (ex. s.isalpha())
s.upper()  s.lower()  s.title()  s.swapcase()
s.casefold()  s.capitalize()  s.center([width,fill])
s.ljust([width,fill])  s.rjust([width,fill])  s.zfill([width])
s.encode(encoding)  s.split([sep])  s.join(seq)
```

Formatting

formating directives — values to format

```
"modele{} {} {}".format(x,y,r) → str
"{selection:formatting!conversion}"
```

- **Selection** :
  `2`
  `nom`
  `0.nom`
  `4[key]`
  `0[2]`
- **Formatting** :
  fill char *alignment* *sign* *mini width*.*precision~maxwidth* *type*
  `< > ^ =` `+ - space` `0` at start for filling with 0
  integer: `b` binary, `c` char, `d` decimal (default), `o` octal, `x` or `X` hexa…
  float: `e` or `E` exponential, `f` or `F` fixed point, `g` or `G` appropriate (default),
  string: `s` … `%` percent
- **Conversion** : `s` (readable text) or `r` (literal representation)

Examples

```
"{:+2.3f}".format(45.72793)
→'+45.728'
"{1:>10s}".format(8,"toto")
→'      toto'
"{x!r}".format(x="I'm")
→'"I\'m"'
```