

Algorithmique et programmation

Philippe Rannou

ESIR

Organisation du module

Volume horaire

44 H (sur 12 semaines) :

- sur 2 semaines \approx 2H de CM + 2H de TD + 4H de TP

Organisation du module

Volume horaire

44 H (sur 12 semaines) :

- sur 2 semaines \approx 2H de CM + 2H de TD + 4H de TP

Modalités d'évaluation

- TP(s) à rendre
- 1 DS (milieu de semestre)
- 1 DS (fin de semestre)

Organisation du module

Volume horaire

44 H (sur 12 semaines) :

- sur 2 semaines \approx 2H de CM + 2H de TD + 4H de TP

Modalités d'évaluation

- TP(s) à rendre
- 1 DS (milieu de semestre)
- 1 DS (fin de semestre)

Équipe pédagogique

- CM : Philippe Rannou
- TD : Philippe Rannou et Hélène Feuillâtre
- TP : Philippe Rannou et Hélène Feuillâtre

Objectifs

Contenu


- Base des concepts algorithmiques (variables, conditionnelles, boucles)
- Syntaxe élémentaire du langage python
- Structures de données composées (tableaux et dictionnaires)
- Notion de récursivité

Contenu

- Base des concepts algorithmiques (variables, conditionnelles, boucles)
- Syntaxe élémentaire du langage python
- Structures de données composées (tableaux et dictionnaires)
- Notion de récursivité

Compétences visées

- Concevoir, analyser et modifier des algorithmes simples
- Décomposer un problème en sous-problèmes
- Implémenter ces algorithmes en python
- Tester et déboguer des programmes
- Déterminer la complexité d'un algorithme

les exercices qui jalonnent ce cours sont signalés par un : , ils seront corrigés en cours et serviront en TDs/TPs.

Plan du cours

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 Données structurées

Quelques exemples d'algorithmes de la vie courante

Recette de cuisine

- *Éplucher et couper les potirons.*
- *Émincer l'ail et l'oignon.*
- *Faire suer l'oignon dans l'huile d'olive.*
- ...

Recette de cuisine

- *Éplucher et couper les potirons.*
- *Émincer l'ail et l'oignon.*
- *Faire suer l'oignon dans l'huile d'olive.*
- ...

Notice d'utilisation

- *Veillez à ce que les tuyaux soient bien connecté.*
- *Branchez votre machine.*
- *Ouvrez complètement le robinet.*
- ...

Quelques exemples d'algorithmes de la vie courante

Recette de cuisine

- *Éplucher et couper les potirons.*
- *Émincer l'ail et l'oignon.*
- *Faire suer l'oignon dans l'huile d'olive.*
- ...

Notice d'utilisation

- *Veillez à ce que les tuyaux soient bien connectés.*
- *Branchez votre machine.*
- *Ouvrez complètement le robinet.*
- ...

Instructions GPS

- *Prendre N137 à Pont-Péan et quitter D286.*
- *Continuer sur N137.*
- *Prendre la sortie en direction de Caen.*
- ...

Qu'est-ce qu'un algorithme ?

Ces exemples ont des caractéristiques communes :

Qu'est-ce qu'un algorithme ?

Ces exemples ont des caractéristiques communes :

- ce sont des suites d'instructions élémentaires,

Qu'est-ce qu'un algorithme ?

Ces exemples ont des caractéristiques communes :

- ce sont des suites d'instructions élémentaires,
- ils permettent de résoudre des tâches déterminées.

Qu'est-ce qu'un algorithme ?

Ces exemples ont des caractéristiques communes :

- ce sont des suites d'instructions élémentaires,
- ils permettent de résoudre des tâches déterminées.

Définition 0.1

*Un **algorithme** est une suite finie d'instructions et permettant de résoudre un problème.*

Qu'est-ce qu'un algorithme ?

Ces exemples ont des caractéristiques communes :

- ce sont des suites d'instructions élémentaires,
- ils permettent de résoudre des tâches déterminées.

Définition 0.1

*Un **algorithme** est une suite finie d'instructions et permettant de résoudre un problème.*

*Le domaine qui étudie les algorithmes est appelé **l'algorithmique**.*

Dans ce cours, nous étudierons l'algorithmique dans le cadre de l'*informatique*.

Dans ce cours, nous étudierons l'algorithmique dans le cadre de l'*informatique*.

Informatique

L'informatique est la science du traitement automatique de l'information.

Dans ce cours, nous étudierons l'algorithmique dans le cadre de l'*informatique*.

Informatique

L'informatique est la science du traitement automatique de l'information.

Elle englobe :

- l'étude des programmes immatériels qui décrivent un traitement à réaliser (logiciel, *software*).
- l'étude des machines qui exécute ce traitement (matériel, *hardware*)

Dans ce cours, nous étudierons l'algorithmique dans le cadre de l'*informatique*.

Informatique

L'informatique est la science du traitement automatique de l'information.

Elle englobe :

- l'étude des programmes immatériels qui décrivent un traitement à réaliser (logiciel, *software*).
- l'étude des machines qui exécute ce traitement (matériel, *hardware*)
- L'algorithmique fait partie du côté *logiciel* de l'informatique.

Algorithme vs Programme Informatique

Algorithme vs Programme Informatique

- En TP vous *implémenterez* des algorithmes sur ordinateur, c'est à dire que vous *traduirez* ces algorithmes en **programmes informatiques** (ici en langage python)

Algorithme vs Programme Informatique

- En TP vous *implémenterez* des algorithmes sur ordinateur, c'est à dire que vous *traduirez* ces algorithmes en **programmes informatiques** (ici en langage python)
- On pourrait donc utiliser 2 langages dans ce cours :

Algorithme vs Programme Informatique

- En TP vous *implémenterez* des algorithmes sur ordinateur, c'est à dire que vous *traduirez* ces algorithmes en **programmes informatiques** (ici en langage python)
- On pourrait donc utiliser 2 langages dans ce cours :
 - du **pseudo-code** pour écrire les algorithmes,

Algorithme vs Programme Informatique

- En TP vous *implémenterez* des algorithmes sur ordinateur, c'est à dire que vous *traduirez* ces algorithmes en **programmes informatiques** (ici en langage python)
- On pourrait donc utiliser 2 langages dans ce cours :
 - du **pseudo-code** pour écrire les algorithmes,
 - le langage python pour écrire les programmes.

Algorithme vs Programme Informatique

- En TP vous *implémenterez* des algorithmes sur ordinateur, c'est à dire que vous *traduirez* ces algorithmes en **programmes informatiques** (ici en langage python)
- On pourrait donc utiliser 2 langages dans ce cours :
 - du **pseudo-code** pour écrire les algorithmes,
 - le langage python pour écrire les programmes.

Algorithme 6 : EstPositif(x)

Entrées : x : entier

```
1 début
2   si  $x \geq 0$  alors
3     | Afficher "Oui"
4   sinon
5     | Afficher "Non"
6   fin
7 fin
```

Pseudo-code

Algorithme vs Programme Informatique

- En TP vous *implémenterez* des algorithmes sur ordinateur, c'est à dire que vous *traduirez* ces algorithmes en **programmes informatiques** (ici en langage python)
- On pourrait donc utiliser 2 langages dans ce cours :
 - du **pseudo-code** pour écrire les algorithmes,
 - le langage python pour écrire les programmes.

Algorithme 7 : EstPositif(x)

Entrées : x : entier

```
1 début
2   si  $x \geq 0$  alors
3     | Afficher "Oui"
4   sinon
5     | Afficher "Non"
6   fin
7 fin
```

Pseudo-code

```
def est_positif(x) :
    if x >= 0 :
        print("Oui")
    else :
        print("Non")
```

python

Algorithme vs Programme Informatique

- En TP vous *implémenterez* des algorithmes sur ordinateur, c'est à dire que vous *traduirez* ces algorithmes en **programmes informatiques** (ici en langage python)
- On pourrait donc utiliser 2 langages dans ce cours :
 - du **pseudo-code** pour écrire les algorithmes,
 - le langage python pour écrire les programmes.

Algorithme 8 : EstPositif(x)

Entrées : x : entier

```
1 début
2   si  $x \geq 0$  alors
3     | Afficher "Oui"
4   sinon
5     | Afficher "Non"
6   fin
7 fin
```

Pseudo-code

```
def est_positif(x) :
    if x >= 0 :
        print("Oui")
    else :
        print("Non")
```

python

⇒ Dans ce cours on n'écrira
que du python

- 1 Concepts de bases
 - Types de données
 - Entrées-sorties
 - Structures de contrôle

- 2 Structuration d'un programme

- 3 Données structurées

Formalisation

En informatique, les algorithmes sont des suites d'**instructions** qui manipulent des **données**.

En informatique, les algorithmes sont des suites d'**instructions** qui manipulent des **données**.

Données

Les **données** peuvent être de différents types :

- entier, booléen, réel, chaîne de caractères, ...
- images, vidéos, ...
- fichiers, ...
- ...

En informatique, les algorithmes sont des suites d'**instructions** qui manipulent des **données**.

Données

Les **données** peuvent être de différents types :

- entier, booléen, réel, chaîne de caractères, ...
- images, vidéos, ...
- fichiers, ...
- ...

→ Toutes ces données sont codées en binaires (avec des "0" et des "1")

Formalisation

En informatique, les algorithmes sont des suites d'**instructions** qui manipulent des **données**.

Données

Les **données** peuvent être de différents types :

- entier, booléen, réel, chaîne de caractères, ...
- images, vidéos, ...
- fichiers, ...
- ...

→ Toutes ces données sont codées en binaires (avec des "0" et des "1")

Instructions

Les **instructions** utilisées en algorithmique (if, while, for, ...) correspondent aux **briques de base** que peut faire directement l'ordinateur.

- 1 Concepts de bases
 - Types de données
 - Entrées-sorties
 - Structures de contrôle

- 2 Structuration d'un programme

- 3 Données structurées

Définition 1.1

*Toute donnée est soit une **variable** soit une **constante**:*

Définition 1.1

*Toute donnée est soit une **variable** soit une **constante**:*

- *une **variable** est une donnée dont la valeur va évoluer lors de l'exécution du programme*

Définition 1.1

*Toute donnée est soit une **variable** soit une **constante**:*

- *une **variable** est une donnée dont la valeur va évoluer lors de l'exécution du programme*
- *une **constante** va conserver la même valeur tout au long du traitement*

Définition 1.1

*Toute donnée est soit une **variable** soit une **constante**:*

- *une **variable** est une donnée dont la valeur va évoluer lors de l'exécution du programme*
- *une **constante** va conserver la même valeur tout au long du traitement*

Chaque donnée (variable comme constante) est définie par :

Définition 1.1

*Toute donnée est soit une **variable** soit une **constante**:*

- *une **variable** est une donnée dont la valeur va évoluer lors de l'exécution du programme*
- *une **constante** va conserver la même valeur tout au long du traitement*

Chaque donnée (variable comme constante) est définie par :

- un **identificateur**

Définition 1.1

*Toute donnée est soit une **variable** soit une **constante**:*

- *une **variable** est une donnée dont la valeur va évoluer lors de l'exécution du programme*
- *une **constante** va conserver la même valeur tout au long du traitement*

Chaque donnée (variable comme constante) est définie par :

- un **identificateur**
- un **type**

Définition 1.1

*Toute donnée est soit une **variable** soit une **constante**:*

- *une **variable** est une donnée dont la valeur va évoluer lors de l'exécution du programme*
- *une **constante** va conserver la même valeur tout au long du traitement*

Chaque donnée (variable comme constante) est définie par :

- un **identificateur**
- un **type**
- une **valeur**

Définition 1.1

*Toute donnée est soit une **variable** soit une **constante**:*

- *une **variable** est une donnée dont la valeur va évoluer lors de l'exécution du programme*
- *une **constante** va conserver la même valeur tout au long du traitement*

Chaque donnée (variable comme constante) est définie par :

- un **identificateur**
- un **type**
- une **valeur**

Programmation

- Dans certains langage, comme le C, Java ou C++, la gestion des variables et constantes est différenciée.

Définition 1.1

*Toute donnée est soit une **variable** soit une **constante**:*

- *une **variable** est une donnée dont la valeur va évoluer lors de l'exécution du programme*
- *une **constante** va conserver la même valeur tout au long du traitement*

Chaque donnée (variable comme constante) est définie par :

- un **identificateur**
- un **type**
- une **valeur**

Programmation

- Dans certains langage, comme le C, Java ou C++, la gestion des variables et constantes est différenciée.
- En python, toutes les données que l'on manipule sont des variables.

Définition 1.2

*L'**identificateur** correspond au nom que l'on donne à une donnée pour l'identifier (variable comme constante) :*

Définition 1.2

*L'**identificateur** correspond au nom que l'on donne à une donnée pour l'identifier (variable comme constante) :*

- composé à partir des lettres (non accentuées) de l'alphabet, des chiffres et du symbole underscore '_'*

Définition 1.2

*L'**identificateur** correspond au nom que l'on donne à une donnée pour l'identifier (variable comme constante) :*

- composé à partir des lettres (non accentuées) de l'alphabet, des chiffres et du symbole underscore '_' (les autres symboles comme l'espace ou le tiret '-' ne sont pas autorisés)*

Conventions pour les identificateurs

Conventions pour les identificateurs

Il existe plusieurs conventions de nommage suivants les langages :

- `nombredejours`
- `nombre_de_jours` : pour les variables/fonctions en python
- `NombreDeJours`
- `NOMBREDEJOURS`
- `NOMBRE_DE_JOURS` : pour les constantes en python
- ...

Conventions pour les identificateurs

Il existe plusieurs conventions de nommage suivants les langages :

- `nombredejours`
- `nombre_de_jours` : pour les variables/fonctions en python
- `NombreDeJours`
- `NOMBREDEJOURS`
- `NOMBRE_DE_JOURS` : pour les constantes en python
- ...

Noms à éviter

Conventions pour les identificateurs

Il existe plusieurs conventions de nommage suivants les langages :

- `nombredejours`
- `nombre_de_jours` : pour les variables/fonctions en python
- `NombreDeJours`
- `NOMBREDEJOURS`
- `NOMBRE_DE_JOURS` : pour les constantes en python
- ...

Noms à éviter

- `l` ("L" minuscule)

Conventions pour les identificateurs

Il existe plusieurs conventions de nommage suivants les langages :

- `nombredejours`
- `nombre_de_jours` : pour les variables/fonctions en python
- `NombreDeJours`
- `NOMBREDEJOURS`
- `NOMBRE_DE_JOURS` : pour les constantes en python
- ...

Noms à éviter

- `l` ("L" minuscule)
- `I` ("i" majuscule)

Conventions pour les identificateurs

Il existe plusieurs conventions de nommage suivants les langages :

- `nombredejours`
- `nombre_de_jours` : pour les variables/fonctions en python
- `NombreDeJours`
- `NOMBREDEJOURS`
- `NOMBRE_DE_JOURS` : pour les constantes en python
- ...

Noms à éviter

- `l` ("L" minuscule)
- `I` ("i" majuscule)
- `O` ("o" majuscule : trop proche du zéro)

Conventions pour les identificateurs

Il existe plusieurs conventions de nommage suivants les langages :

- `nombredejours`
- `nombre_de_jours` : pour les variables/fonctions en python
- `NombreDeJours`
- `NOMBREDEJOURS`
- `NOMBRE_DE_JOURS` : pour les constantes en python
- ...

Noms à éviter

- `l` ("L" minuscule)
- `I` ("i" majuscule)
- `O` ("o" majuscule : trop proche du zéro)

Pour plus d'informations sur les conventions de nommage en python :

<https://www.python.org/dev/peps/pep-0008/>

Types de données

Définition 1.3

Le **type** d'une donnée (ou plus simple type) désigne les **valeurs** que peut prendre une donnée ainsi que les **opérateurs** qui lui sont applicables.

Définition 1.3

Le **type** d'une donnée (ou plus simple type) désigne les **valeurs** que peut prendre une donnée ainsi que les **opérateurs** qui lui sont applicables.

- Les valeurs que peuvent prendre une donnée sont codées en binaire

Définition 1.3

Le **type** d'une donnée (ou plus simple type) désigne les **valeurs** que peut prendre une donnée ainsi que les **opérateurs** qui lui sont applicables.

- Les valeurs que peuvent prendre une donnée sont codées en binaire
- À la nature (numérique, caractère ...) de l'information mémorisée correspond généralement une manière de coder cette information.

Définition 1.3

Le **type** d'une donnée (ou plus simple **type**) désigne les **valeurs** que peut prendre une donnée ainsi que les **opérateurs** qui lui sont applicables.

- Les valeurs que peuvent prendre une donnée sont codées en binaire
- À la nature (numérique, caractère ...) de l'information mémorisée correspond généralement une manière de coder cette information.
- Le type d'une variable permet d'associer entre eux : la nature des informations, le codage mais aussi les limites et opérations associées.

Types de données en python

- Booléen (`bool`) :
 - valeur : `True` ou `False` (Attention aux majuscules)

Types de données en python

- Booléen (`bool`) :
 - valeur : `True` ou `False` (Attention aux majuscules)
- Entier (`int`)
 - valeur : n'importe quelle valeur entière
 - exemple : 18, -3, +326

Types de données en python

- Booléen (`bool`) :
 - valeur : `True` ou `False` (Attention aux majuscules)
- Entier (`int`)
 - valeur : n'importe quelle valeur entière
 - exemple : 18, -3, +326
- Réel (`float`)
 - valeur : n'importe quelle valeur réelle
 - exemple : 1.0, 3.14, -128.32

Types de données en python

- Booléen (`bool`) :
 - valeur : `True` ou `False` (Attention aux majuscules)
- Entier (`int`)
 - valeur : n'importe quelle valeur entière
 - exemple : 18, -3, +326
- Réel (`float`)
 - valeur : n'importe quelle valeur réelle
 - exemple : 1.0, 3.14, -128.32
- Chaîne de caractères (`string`) :
 - valeur : ensemble de lettres, chiffres, symboles ...
 - exemple : "ABCD" , "a"

Types de données en python

- Booléen (`bool`) :
 - valeur : `True` ou `False` (Attention aux majuscules)
- Entier (`int`)
 - valeur : n'importe quelle valeur entière
 - exemple : 18, -3, +326
- Réel (`float`)
 - valeur : n'importe quelle valeur réelle
 - exemple : 1.0, 3.14, -128.32
- Chaîne de caractères (`string`) :
 - valeur : ensemble de lettres, chiffres, symboles ...
 - exemple : "ABCD" , "a"

Déclaration

Dans certains langages (C, Java, C++), le typage est **explicite**, c'est à dire que le type est précisé lors de la déclaration.

Types de données en python

- Booléen (`bool`) :
 - valeur : `True` ou `False` (Attention aux majuscules)
- Entier (`int`)
 - valeur : n'importe quelle valeur entière
 - exemple : 18, -3, +326
- Réel (`float`)
 - valeur : n'importe quelle valeur réelle
 - exemple : 1.0, 3.14, -128.32
- Chaîne de caractères (`string`) :
 - valeur : ensemble de lettres, chiffres, symboles ...
 - exemple : "ABCD" , "a"

Déclaration

Dans certains langages (C, Java, C++), le typage est **explicite**, c'est à dire que le type est précisé lors de la déclaration. (Ex : `int a = 5;`)

Types de données en python

- Booléen (`bool`) :
 - valeur : `True` ou `False` (Attention aux majuscules)
- Entier (`int`)
 - valeur : n'importe quelle valeur entière
 - exemple : 18, -3, +326
- Réel (`float`)
 - valeur : n'importe quelle valeur réelle
 - exemple : 1.0, 3.14, -128.32
- Chaîne de caractères (`string`) :
 - valeur : ensemble de lettres, chiffres, symboles ...
 - exemple : "ABCD" , "a"

Déclaration

Dans certains langages (C, Java, C++), le typage est **explicite**, c'est à dire que le type est précisé lors de la déclaration. (Ex : `int a = 5;`)
En python, la plupart du temps, on ne précise pas le type des variables que l'on manipule.

Définition 1.4 (Affectation)

Définition 1.4 (Affectation)

Une **affectation** *identifiant* = *expression* est une instruction qui permet de spécifier qu'au moment de son exécution, la variable désignée par *identifiant* recevra comme nouvelle valeur le résultat de *expression*.

Définition 1.4 (Affectation)

Une **affectation** *identifiant* = *expression* est une instruction qui permet de spécifier qu'au moment de son exécution, la variable désignée par *identifiant* recevra comme nouvelle valeur le résultat de *expression*.

En python, l'affectation s'effectue avec le signe "=", et le type d'une variable peut changer :

Définition 1.4 (Affectation)

Une **affectation** *identifiant* = *expression* est une instruction qui permet de spécifier qu'au moment de son exécution, la variable désignée par *identifiant* recevra comme nouvelle valeur le résultat de *expression*.

En python, l'affectation s'effectue avec le signe "=", et le type d'une variable peut changer :

```
n = 8
```

```
n = "ABC"
```

Définition 1.4 (Affectation)

Une **affectation** *identifiant* = *expression* est une instruction qui permet de spécifier qu'au moment de son exécution, la variable désignée par *identifiant* recevra comme nouvelle valeur le résultat de *expression*.

En python, l'affectation s'effectue avec le signe "=", et le type d'une variable peut changer :

```
n = 8  
n = "ABC"
```

→ **MAIS !** C'est fortement déconseillé ! (et souvent impossible dans les autres langages)

`identifiant = expression`

`identifiant = expression`

- Une expression permet de désigner le calcul d'une nouvelle valeur à partir d'autres valeurs et d'opérations.

`identifiant = expression`

- Une expression permet de désigner le calcul d'une nouvelle valeur à partir d'autres valeurs et d'opérations.
- Les valeurs utilisées dans l'expression peuvent être des constantes, des valeurs de variables, des données littérales ...:

`identifiant = expression`

- Une expression permet de désigner le calcul d'une nouvelle valeur à partir d'autres valeurs et d'opérations.
- Les valeurs utilisées dans l'expression peuvent être des constantes, des valeurs de variables, des données littérales ...:
 - $3 + 27$ ou $x * 3,0$ si x est une variable de type réel

`identifiant = expression`

- Une expression permet de désigner le calcul d'une nouvelle valeur à partir d'autres valeurs et d'opérations.
- Les valeurs utilisées dans l'expression peuvent être des constantes, des valeurs de variables, des données littérales ...:
 - $3 + 27$ ou $x * 3,0$ si x est une variable de type réel
- Lors de l'affectation `identifiant = expression` :

`identifiant = expression`

- Une expression permet de désigner le calcul d'une nouvelle valeur à partir d'autres valeurs et d'opérations.
- Les valeurs utilisées dans l'expression peuvent être des constantes, des valeurs de variables, des données littérales ...:
 - $3 + 27$ ou $x * 3,0$ si x est une variable de type réel
- Lors de l'affectation `identifiant = expression` :
 - 1 L'expression est d'abord évaluée.

`identifiant = expression`

- Une expression permet de désigner le calcul d'une nouvelle valeur à partir d'autres valeurs et d'opérations.
- Les valeurs utilisées dans l'expression peuvent être des constantes, des valeurs de variables, des données littérales ...:
 - $3 + 27$ ou $x * 3,0$ si x est une variable de type réel
- Lors de l'affectation `identifiant = expression` :
 - 1 L'expression est d'abord évaluée.
 - 2 Le résultat de l'expression (donc la valeur) est ensuite affectée à la variable désignée par son identifiant.

`identifiant = expression`

- Une expression permet de désigner le calcul d'une nouvelle valeur à partir d'autres valeurs et d'opérations.
 - Les valeurs utilisées dans l'expression peuvent être des constantes, des valeurs de variables, des données littérales ...:
 - $3 + 27$ ou $x * 3,0$ si x est une variable de type réel
 - Lors de l'affectation `identifiant = expression` :
 - 1 L'expression est d'abord évaluée.
 - 2 Le résultat de l'expression (donc la valeur) est ensuite affectée à la variable désignée par son identifiant.
- on peut donc utiliser la valeur d'une variable dans sa propre affectation :

Affectation (2/2)

`identifiant = expression`

- Une expression permet de désigner le calcul d'une nouvelle valeur à partir d'autres valeurs et d'opérations.
- Les valeurs utilisées dans l'expression peuvent être des constantes, des valeurs de variables, des données littérales ...:
 - $3 + 27$ ou $x * 3,0$ si x est une variable de type réel
- Lors de l'affectation `identifiant = expression` :
 - 1 L'expression est d'abord évaluée.
 - 2 Le résultat de l'expression (donc la valeur) est ensuite affectée à la variable désignée par son identifiant.

→ on peut donc utiliser la valeur d'une variable dans sa propre affectation :

$$x = x + 3$$

Conversion de type en python

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Exemples :

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Exemples :

- `float(3)`

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Exemples :

- `float(3)` → 3.0

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Exemples :

- `float(3)` → 3.0
- `int(5.6)`

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Exemples :

- `float(3)` → 3.0
- `int(5.6)` → 5

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Exemples :

- `float(3)` → 3.0
- `int(5.6)` → 5
- `string(68)`

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Exemples :

- `float(3)` → 3.0
- `int(5.6)` → 5
- `string(68)` → "68"

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Exemples :

- `float(3)` → 3.0
- `int(5.6)` → 5
- `string(68)` → "68"
- `int("35")`

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Exemples :

- `float(3)` → 3.0
- `int(5.6)` → 5
- `string(68)` → "68"
- `int("35")` → 35

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Exemples :

- `float(3)` → 3.0
- `int(5.6)` → 5
- `string(68)` → "68"
- `int("35")` → 35
- `float("08.93")`

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Exemples :

- `float(3)` → 3.0
- `int(5.6)` → 5
- `string(68)` → "68"
- `int("35")` → 35
- `float("08.93")` → 8.93

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Exemples :

- `float(3)` → 3.0
- `int(5.6)` → 5
- `string(68)` → "68"
- `int("35")` → 35
- `float("08.93")` → 8.93
- `str(09.64)`

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Exemples :

- | | |
|----------------------------------|--------------------------------------|
| • <code>float(3)</code> → 3.0 | • <code>int("35")</code> → 35 |
| • <code>int(5.6)</code> → 5 | • <code>float("08.93")</code> → 8.93 |
| • <code>string(68)</code> → "68" | • <code>str(09.64)</code> → "9.64" |

Définition 1.5

*Un **opérateur** est un symbole indiquant une opération.*

Définition 1.5

*Un **opérateur** est un symbole indiquant une opération.*

- Les opérateurs sont souvent binaires (c'est-à-dire reliant **deux** opérandes) mais peuvent aussi être unaires (une seule opérande), ternaires voire n -aires.

Définition 1.5

*Un **opérateur** est un symbole indiquant une opération.*

- Les opérateurs sont souvent binaires (c'est-à-dire reliant **deux** opérandes) mais peuvent aussi être unaires (une seule opérande), ternaires voire n -aires.
- Un opérateur est généralement associé à un type de données. (le "+" a la même signification lorsqu'il est utilisé sur des entiers ou des réels, mais pas sur des chaînes de caractères).

Définition 1.5

*Un **opérateur** est un symbole indiquant une opération.*

- Les opérateurs sont souvent binaires (c'est-à-dire reliant **deux** opérandes) mais peuvent aussi être unaires (une seule opérande), ternaires voire n -aires.
- Un opérateur est généralement associé à un type de données. (le "+" a la même signification lorsqu'il est utilisé sur des entiers ou des réels, mais pas sur des chaînes de caractères).
- Trois familles d'opérateurs sont distinguées :
 - ① opérateurs arithmétiques : donnent un résultat numérique à partir d'opérandes numériques (addition, soustraction ...)
 - ② opérateurs relationnels : donnent un résultat logique (booléen) à partir d'opérandes numériques (plus grand que, égal ...)
 - ③ opérateurs logiques : donnent un résultat logique à partir d'opérandes logiques (et logique ...)

Opérateurs sur les chaînes de caractères

Une variable de type chaîne de caractères :

Opérateurs sur les chaînes de caractères

Une variable de type chaîne de caractères :

- ne peut utiliser d'opérateurs arithmétiques

Opérateurs sur les chaînes de caractères

Une variable de type chaîne de caractères :

- ne peut utiliser d'opérateurs arithmétiques
- peut utiliser les opérateurs relationnels : $>$, $>=$, $<$, $<=$, $=$ et \neq
(utilisation de l'ordre *lexicographique*, i.e. celui du dictionnaire)

Opérateurs sur les chaînes de caractères

Une variable de type chaîne de caractères :

- ne peut utiliser d'opérateurs arithmétiques
- peut utiliser les opérateurs relationnels : $>$, $>=$, $<$, $<=$, $=$ et \neq (utilisation de l'ordre *lexicographique*, i.e. celui du dictionnaire)
- possède un opérateur de concaténation : $\ll + \gg$

Opérateurs sur les chaînes de caractères

Une variable de type chaîne de caractères :

- ne peut utiliser d'opérateurs arithmétiques
- peut utiliser les opérateurs relationnels : $>$, $>=$, $<$, $<=$, $=$ et \neq (utilisation de l'ordre *lexicographique*, i.e. celui du dictionnaire)
- possède un opérateur de concaténation : $\ll + \gg$

Exemple : "abc" + "bcd" retourne "abcbcd"

Opérateurs sur les chaînes de caractères

Une variable de type chaîne de caractères :

- ne peut utiliser d'opérateurs arithmétiques
- peut utiliser les opérateurs relationnels : $>$, $>=$, $<$, $<=$, $=$ et $!=$ (utilisation de l'ordre *lexicographique*, i.e. *celui du dictionnaire*)
- possède un opérateur de concaténation : $\ll + \gg$

Exemple : "abc" + "bcd" retourne "abcbcd"

Remarque

En python, on peut effectuer l'opération `chaine*k`, avec `k` entier : cela donnera le résultat `chaine chaine ... chaine` où `chaine` est répétée `k` fois.

Opérateurs sur les booléens

Opérateurs sur les booléens

- En python, on utilise `True` et `False` pour désigner les deux valeurs booléenne.
- Les seuls opérateurs utilisables avec le type booléen sont les opérateurs logiques ("et" : `and`, "ou" : `or`, "non" : `not`) et les opérateurs relationnels `==` et `!=`

- En python, on utilise `True` et `False` pour désigner les deux valeurs booléenne.
- Les seuls opérateurs utilisables avec le type booléen sont les opérateurs logiques ("`et`" : `and`, "`ou`" : `or`, "`non`" : `not`) et les opérateurs relationnels `==` et `!=`
- Un petit rappel de logique :
 - l'expression `a or b` est vraie si l'une des variables `a` ou `b` est vraie.
 - l'expression `a and b` est vraie si les deux variables `a` et `b` sont vraies.
 - l'expression `not a` est le contraire de `a`.

Opérateurs sur les entiers

Opérateurs sur les entiers

- Les opérations arithmétiques binaires telles que $+$, $-$, $*$, $//$ (division entière) et $\%$ (modulo : reste de la division entière) sont disponibles sur les entiers et donnent toujours un entier (les opérateurs relationnels sont aussi disponibles) :

Opérateurs sur les entiers

- Les opérations arithmétiques binaires telles que $+$, $-$, $*$, $//$ (division entière) et $\%$ (modulo : reste de la division entière) sont disponibles sur les entiers et donnent toujours un entier (les opérateurs relationnels sont aussi disponibles) :

```
x = 13 // 2
```

```
# la valeur de x est donc 6
```

```
# (ici // désigne la division euclidienne)
```

```
x = 13 % 2
```

```
# la valeur de x est donc 1
```

```
# (ici % désigne le reste de la division)
```

Opérateurs sur les entiers

- Les opérations arithmétiques binaires telles que $+$, $-$, $*$, $//$ (division entière) et $\%$ (modulo : reste de la division entière) sont disponibles sur les entiers et donnent toujours un entier (les opérateurs relationnels sont aussi disponibles) :

```
x = 13 // 2
```

```
# la valeur de x est donc 6
```

```
# (ici // désigne la division euclidienne)
```

```
x = 13 % 2
```

```
# la valeur de x est donc 1
```

```
# (ici % désigne le reste de la division)
```

Remarques :

- Dans certains langages (C, Java, C++), une variable de type entier ne peut prendre qu'un nombre fini de valeurs qui est fonction du nombre d'octets nécessaires à son codage.

Opérateurs sur les entiers

- Les opérations arithmétiques binaires telles que $+$, $-$, $*$, $//$ (division entière) et $\%$ (modulo : reste de la division entière) sont disponibles sur les entiers et donnent toujours un entier (les opérateurs relationnels sont aussi disponibles) :

```
x = 13 // 2
```

```
# la valeur de x est donc 6
```

```
# (ici // désigne la division euclidienne)
```

```
x = 13 % 2
```

```
# la valeur de x est donc 1
```

```
# (ici % désigne le reste de la division)
```

Remarques :

- Dans certains langages (C, Java, C++), une variable de type entier ne peut prendre qu'un nombre fini de valeurs qui est fonction du nombre d'octets nécessaires à son codage. Un entier qui serait codé sur 1 octet (8 bits) ne pourrait prendre que 256 valeurs (2^8) de 0 à 255 par exemple.

Opérateurs sur les entiers

- Les opérations arithmétiques binaires telles que $+$, $-$, $*$, $//$ (division entière) et $\%$ (modulo : reste de la division entière) sont disponibles sur les entiers et donnent toujours un entier (les opérateurs relationnels sont aussi disponibles) :

```
x = 13 // 2
```

```
# la valeur de x est donc 6
```

```
# (ici // désigne la division euclidienne)
```

```
x = 13 % 2
```

```
# la valeur de x est donc 1
```

```
# (ici % désigne le reste de la division)
```

Remarques :

- Dans certains langages (C, Java, C++), une variable de type entier ne peut prendre qu'un nombre fini de valeurs qui est fonction du nombre d'octets nécessaires à son codage. Un entier qui serait codé sur 1 octet (8 bits) ne pourrait prendre que 256 valeurs (2^8) de 0 à 255 par exemple. En python il n'y a pas de limite.

Opérateurs sur les réels

- Les opérations arithmétiques binaires telles que $+$, $-$, $*$, $/$ sont disponibles sur les réels et donnent toujours un réel.

Opérateurs sur les réels

- Les opérations arithmétiques binaires telles que $+$, $-$, $*$, $/$ sont disponibles sur les réels et donnent toujours un réel.
- Les opérateurs relationnels sont aussi disponibles sur les réels **mais** les opérateurs $==$ et $!=$ sont **absolument** à éviter !

Opérateurs sur les réels

- Les opérations arithmétiques binaires telles que $+$, $-$, $*$, $/$ sont disponibles sur les réels et donnent toujours un réel.
- Les opérateurs relationnels sont aussi disponibles sur les réels **mais** les opérateurs $==$ et $!=$ sont **absolument** à éviter !

Remarques :

- Une variable de type réel permet la représentation des nombres avec une certaine précision qui est fonction du nombre d'octets utilisées pour son codage.

Opérateurs sur les réels

- Les opérations arithmétiques binaires telles que $+$, $-$, $*$, $/$ sont disponibles sur les réels et donnent toujours un réel.
- Les opérateurs relationnels sont aussi disponibles sur les réels **mais** les opérateurs `==` et `!=` sont **absolument** à éviter !

Remarques :

- Une variable de type réel permet la représentation des nombres avec une certaine précision qui est fonction du nombre d'octets utilisées pour son codage.
- L'opération `/` est aussi disponible sur les entiers en python, mais elle renvoie systématiquement un réel (même si le résultat est entier).



Exercice

Algorithme	Type de la variable ?	Valeur ?
<code>tva = 19.6</code>		



Exercice

Algorithme	Type de la variable ?	Valeur ?
<code>tva = 19.6</code>	Réel	



Exercice

Algorithme	Type de la variable ?	Valeur ?
$tva = 19.6$	R��el	$tva = 19.6$
$x = 4/3$		



Exercice

Algorithme	Type de la variable ?	Valeur ?
$tva = 19.6$	Réel	$tva = 19.6$
$x = 4/3$	Réel	



Exercice

Algorithme	Type de la variable ?	Valeur ?
$\text{tva} = 19.6$	Réel	$\text{tva} = 19.6$
$x = 4/3$	Réel	$x \approx 1,3333$
$n = 17\%5$		



Exercice

Algorithme	Type de la variable ?	Valeur ?
$\text{tva} = 19.6$	Réel	$\text{tva} = 19.6$
$x = 4/3$	Réel	$x \approx 1,3333$
$n = 17\%5$	Entier	



Exercice

Algorithme	Type de la variable ?	Valeur ?
$tva = 19.6$	Réel	$tva = 19.6$
$x = 4/3$	Réel	$x \approx 1,3333$
$n = 17\%5$	Entier	$n = 2$
$m = 28 // 3$		



Exercice

Algorithme	Type de la variable ?	Valeur ?
$tva = 19.6$	Réel	$tva = 19.6$
$x = 4/3$	Réel	$x \approx 1,3333$
$n = 17\%5$	Entier	$n = 2$
$m = 28 // 3$	Entier	



Exercice

Algorithme	Type de la variable ?	Valeur ?
$tva = 19.6$	Réel	$tva = 19.6$
$x = 4/3$	Réel	$x \approx 1,3333$
$n = 17\%5$	Entier	$n = 2$
$m = 28 // 3$	Entier	$m = 9$



Exercice

Algorithme	Type de la variable ?	Valeur ?
<pre>b = " True"</pre>		



Exercice

Algorithme	Type de la variable ?	Valeur ?
<pre>b = "True"</pre>	Chaîne	



Exercice

Algorithme	Type de la variable ?	Valeur ?
<pre>b = "True"</pre>	Chaîne	c = "True"
<pre>c = True and (False or True)</pre>		



Exercice

Algorithme	Type de la variable ?	Valeur ?
<pre>b = "True"</pre>	Chaîne	c = "True"
<pre>c = True and (False or True)</pre>	Booléen	



Exercice

Algorithme	Type de la variable ?	Valeur ?
<pre>b = "True"</pre>	Chaîne	c = "True"
<pre>c = True and (False or True)</pre>	Booléen	c = True
<pre>mot = "Le" + " chat"</pre>		



Exercice

Algorithme	Type de la variable ?	Valeur ?
<pre>b = "True"</pre>	Chaîne	c = "True"
<pre>c = True and (False or True)</pre>	Booléen	c = True
<pre>mot = "Le" + " chat"</pre>	Chaîne	



Exercice

Algorithme	Type de la variable ?	Valeur ?
<pre>b = "True"</pre>	Chaîne	c = "True"
<pre>c = True and (False or True)</pre>	Booléen	c = True
<pre>mot = "Le"+" chat"</pre>	Chaîne	mot = "Lechat"
<pre>mot2 = "3+3=" +6</pre>		



Exercice

Algorithme	Type de la variable ?	Valeur ?
<pre>b = "True"</pre>	Chaîne	c = "True"
<pre>c = True and (False or True)</pre>	Booléen	c = True
<pre>mot = "Le"+" chat"</pre>	Chaîne	mot = "Lechat"
<pre>mot2 = "3+3="+6</pre>	Erreur !	

- 1 Concepts de bases
 - Types de données
 - **Entrées-sorties**
 - Structures de contrôle

2 Structuration d'un programme

3 Données structurées

- Un programme a généralement besoin de communiquer avec l'extérieur :

- Un programme a généralement besoin de communiquer avec l'extérieur :
 - affichage de résultats sur un écran,

- Un programme a généralement besoin de communiquer avec l'extérieur :
 - affichage de résultats sur un écran,
 - demande à l'utilisateur de fournir une donnée,
 - ...

- Un programme a généralement besoin de communiquer avec l'extérieur :
 - affichage de résultats sur un écran,
 - demande à l'utilisateur de fournir une donnée,
 - ...
- Dans la réalité les instructions d'écriture et de lecture se font sur des périphériques :

- Un programme a généralement besoin de communiquer avec l'extérieur :
 - affichage de résultats sur un écran,
 - demande à l'utilisateur de fournir une donnée,
 - ...
- Dans la réalité les instructions d'écriture et de lecture se font sur des périphériques :
 - la lecture spécifie qu'une nouvelle valeur d'une variable doit être lue sur un périphérique (typiquement à l'aide du clavier)

- Un programme a généralement besoin de communiquer avec l'extérieur :
 - affichage de résultats sur un écran,
 - demande à l'utilisateur de fournir une donnée,
 - ...
- Dans la réalité les instructions d'écriture et de lecture se font sur des périphériques :
 - la lecture spécifie qu'une nouvelle valeur d'une variable doit être lue sur un périphérique (typiquement à l'aide du clavier)
 - l'écriture permet d'écrire sur un périphérique (typiquement l'écran) la valeur d'une variable ou d'une expression.

Définition 1.6

*La **lecture** est une instruction d'affectation à une variable d'une valeur saisie (au clavier) lorsque l'algorithme est exécuté.*

Définition 1.6

La **lecture** est une instruction d'affectation à une variable d'une valeur saisie (au clavier) lorsque l'algorithme est exécuté.

En python:

```
x = input()
```

Définition 1.6

La **lecture** est une instruction d'affectation à une variable d'une valeur saisie (au clavier) lorsque l'algorithme est exécuté.

En python:

```
x = input()
```

Attention !

- En python, le type par défaut d'une lecture sera une chaîne de caractère.

Définition 1.6

La **lecture** est une instruction d'affectation à une variable d'une valeur saisie (au clavier) lorsque l'algorithme est exécuté.

En python:

```
x = input()
```

Attention !

- En python, le type par défaut d'une lecture sera une chaîne de caractère.
- Si l'on souhaite, par exemple, un entier, il faut écrire :

Définition 1.6

La **lecture** est une instruction d'affectation à une variable d'une valeur saisie (au clavier) lorsque l'algorithme est exécuté.

En python:

```
x = input()
```

Attention !

- En python, le type par défaut d'une lecture sera une chaîne de caractère.
- Si l'on souhaite, par exemple, un entier, il faut écrire :

```
x = int(input())
```


Définition 1.7

*L'**écriture** est une instruction permettant d'écrire (c'est-à-dire d'afficher à l'écran) les valeurs des résultats ainsi que d'éventuels commentaires appropriés.*

Définition 1.7

*L'**écriture** est une instruction permettant d'écrire (c'est-à-dire d'afficher à l'écran) les valeurs des résultats ainsi que d'éventuels commentaires appropriés.*

En python:

```
x = 3  
print(x)
```

Définition 1.7

*L'**écriture** est une instruction permettant d'écrire (c'est-à-dire d'afficher à l'écran) les valeurs des résultats ainsi que d'éventuels commentaires appropriés.*

En python:

```
x = 3  
print(x)
```

- Pour afficher plusieurs chaînes `c1`, `c2`, `c3`, on peut utiliser la commande `print(c1+c2+c3)`. **Mais cela ne fonctionne qu'avec des chaînes de caractères.**
- Pour afficher plusieurs arguments de type différents `a1`, `a2`, `a3`, on peut utiliser la commande `print(a1,a2,a3)`.



Exercice

Qu'affiche l'algorithme suivant à l'écran ?

```
print("Entrez une valeur : ")
x = input()
y = 2*x
x = 2*x+y
print(x)
```



Exercice

Qu'affiche l'algorithme suivant à l'écran ?

```
print("Entrez une valeur : ")
x = input()
y = 2*x
x = 2*x+y
print(x)
```

Cet algorithme :

- Affiche le message "Entrez une valeur :"



Qu'affiche l'algorithme suivant à l'écran ?

```
print("Entrez une valeur : ")
x = input()
y = 2*x
x = 2*x+y
print(x)
```

Cet algorithme :

- Affiche le message "Entrez une valeur :"
- Attends que l'utilisateur entre un nombre.



Qu'affiche l'algorithme suivant à l'écran ?

```
print("Entrez une valeur : ")
x = input()
y = 2*x
x = 2*x+y
print(x)
```

Cet algorithme :

- Affiche le message "Entrez une valeur :"
- Attends que l'utilisateur entre un nombre.
- Affiche le résultat de 4 fois cet entier.

- 1 Concepts de bases
 - Types de données
 - Entrées-sorties
 - Structures de contrôle

- 2 Structuration d'un programme

- 3 Données structurées



Exemple

Écrivez un algorithme demandant à l'utilisateur de rentrer un entier et affichant la valeur absolue de cet entier.



Exemple

Écrivez un algorithme demandant à l'utilisateur de rentrer un entier et affichant la valeur absolue de cet entier.

Ici il nous manque un type d'instruction permettant de faire un **choix**.



Exemple

Écrivez un algorithme demandant à l'utilisateur de rentrer un entier et affichant la valeur absolue de cet entier.

Ici il nous manque un type d'instruction permettant de faire un **choix**.

Ce type d'instruction s'appelle **branchement conditionnel**.



Exemple

Écrivez un algorithme demandant à l'utilisateur de rentrer un entier et affichant la valeur absolue de cet entier.

Ici il nous manque un type d'instruction permettant de faire un **choix**.

Ce type d'instruction s'appelle **branchement conditionnel**.

Remarque : il est possible de faire cet exercice en n'utilisant pas de conditionnelle en retournant $\sqrt{x^2}$

Structures de contrôle

- Jusqu'à maintenant, les instructions d'un programme s'exécutaient dans l'ordre de leur apparition.

- Jusqu'à maintenant, les instructions d'un programme s'exécutaient dans l'ordre de leur apparition.
- Nous avons aussi besoin, dans un algorithme, de pouvoir :

- Jusqu'à maintenant, les instructions d'un programme s'exécutaient dans l'ordre de leur apparition.
- Nous avons aussi besoin, dans un algorithme, de pouvoir :
 - faire un choix (qui dépend par exemple d'une saisie de l'utilisateur),

- Jusqu'à maintenant, les instructions d'un programme s'exécutaient dans l'ordre de leur apparition.
- Nous avons aussi besoin, dans un algorithme, de pouvoir :
 - faire un choix (qui dépend par exemple d'une saisie de l'utilisateur),
 - répéter plusieurs fois une même série d'instructions.

- Jusqu'à maintenant, les instructions d'un programme s'exécutaient dans l'ordre de leur apparition.
- Nous avons aussi besoin, dans un algorithme, de pouvoir :
 - faire un choix (qui dépend par exemple d'une saisie de l'utilisateur),
 - répéter plusieurs fois une même série d'instructions.

Définition 1.8

*Une **structure de contrôle** est une instruction destinée à commander le déroulement du programme.*

Structure conditionnelle (1/2)

Définition 1.9

Une **structure conditionnelle** permet de réaliser une série d'instructions plutôt qu'une autre (donc de faire un choix) en fonction du résultat d'un test (donc d'une condition).

Structure conditionnelle (1/2)

Définition 1.9

Une **structure conditionnelle** permet de réaliser une série d'instructions plutôt qu'une autre (donc de faire un choix) en fonction du résultat d'un test (donc d'une condition).

```
if condition :  
    instructions 1
```

Si le résultat de "condition" est True alors l'algorithme réalise "instructions 1".

Structure conditionnelle (1/2)

Définition 1.9

Une **structure conditionnelle** permet de réaliser une série d'instructions plutôt qu'une autre (donc de faire un choix) en fonction du résultat d'un test (donc d'une condition).

```
if condition :  
•   instructions 1
```

```
if condition :  
    instructions 1  
• else :  
    instructions 2
```

Si le résultat de "condition" est True alors l'algorithme réalise "instructions 1".

Si le résultat de "condition" est True alors l'algorithme réalise "instructions 1", sinon "instructions 2".

Structure conditionnelle (2/2)

Structure conditionnelle (2/2)

`condition` est une expression dont l'évaluation retourne une valeur booléenne (donc `True` ou `False`)

Structure conditionnelle (2/2)

`condition` est une expression dont l'évaluation retourne une valeur booléenne (donc `True` ou `False`)

Exemple :

```
if age < 18 :  
    print("personne mineure")  
else :  
    print("personne majeure")
```

Précisions syntaxiques en python

Précisions syntaxiques en python

En python, on peut utiliser :

- une conditionnelle simple :

```
if condition :  
    instructions
```

Précisions syntaxiques en python

En python, on peut utiliser :

- une conditionnelle simple :

```
if condition :  
    instructions
```

Remarque : il faut une tabulation avant chacune des "instructions"

Précisions syntaxiques en python

En python, on peut utiliser :

- une conditionnelle simple :

```
if condition :  
    instructions
```

- une conditionnelle avec sinon :

```
if condition :  
    instructions1  
else :  
    instructions2
```

Remarque : il faut une tabulation avant chacune des "instructions"

Précisions syntaxiques en python

En python, on peut utiliser :

- une conditionnelle simple :

```
if condition :  
    instructions
```

- une conditionnelle avec sinon :

```
if condition :  
    instructions1  
else :  
    instructions2
```

Remarque : il faut une tabulation avant chacune des "instructions"

Remarque : le else est au même niveau que le if

Précisions syntaxiques en python

En python, on peut utiliser :

- une conditionnelle simple :

```
if condition :  
    instructions
```

Remarque : il faut une tabulation avant chacune des "instructions"

- une conditionnelle avec sinon :

```
if condition :  
    instructions1  
else :  
    instructions2
```

Remarque : le else est au même niveau que le if

- une conditionnelle avec succession de conditions :

```
if condition1 :  
    instructions1  
elif condition2 :  
    instructions2  
...
```

Exemple

Écrivez un algorithme qui affiche
3 fois la chaîne "coucou".



Exemple

Écrivez un algorithme qui affiche 3 fois la chaîne "coucou".

```
print("coucou")  
print("coucou")  
print("coucou")
```

Écrivez un algorithme qui demande à l'utilisateur d'entrer un entier positif et affiche autant de fois la chaîne "coucou".



Exemple

Écrivez un algorithme qui affiche 3 fois la chaîne "coucou".

```
print("coucou")  
print("coucou")  
print("coucou")
```

Écrivez un algorithme qui demande à l'utilisateur d'entrer un entier positif et affiche autant de fois la chaîne "coucou".

Ici on ne peut pas savoir avant d'exécuter l'algorithme combien de fois il faudra afficher la chaîne.



Exemple

Écrivez un algorithme qui affiche 3 fois la chaîne "coucou".

```
print("coucou")  
print("coucou")  
print("coucou")
```

Écrivez un algorithme qui demande à l'utilisateur d'entrer un entier positif et affiche autant de fois la chaîne "coucou".

Ici on ne peut pas savoir avant d'exécuter l'algorithme combien de fois il faudra afficher la chaîne.

*On devra donc utiliser une **structure répétitive** qui se servira de l'entier entré par l'utilisateur.*

Structure répétitive

Définition 1.10

Une **structure répétitive** (ou boucle) permet de répéter plusieurs fois une série d'instructions en fonction d'une condition.

Définition 1.10

Une **structure répétitive** (ou boucle) permet de répéter plusieurs fois une série d'instructions en fonction d'une condition.

Il existe deux familles de structures répétitives, selon que l'on connaisse ou non à l'avance le nombre de répétitions à réaliser.

Définition 1.10

Une **structure répétitive** (ou *boucle*) permet de répéter plusieurs fois une série d'instructions en fonction d'une condition.

Il existe deux familles de structures répétitives, selon que l'on connaisse ou non à l'avance le nombre de répétitions à réaliser.

Exemples :

- si je dois faire la somme de dix entiers saisies au clavier alors je connais à l'avance le nombre de répétitions à réaliser.

Définition 1.10

Une **structure répétitive** (ou *boucle*) permet de répéter plusieurs fois une série d'instructions en fonction d'une condition.

Il existe deux familles de structures répétitives, selon que l'on connaisse ou non à l'avance le nombre de répétitions à réaliser.

Exemples :

- si je dois faire la somme de dix entiers saisies au clavier alors je connais à l'avance le nombre de répétitions à réaliser.
- si je dois sortir du programme lorsque l'utilisateur saisie le caractère q (et uniquement dans ce cas) alors je ne connais pas à l'avance le nombre de répétitions.

Définition 1.11

*Une structure répétitive de type **while** permet de répéter une série d'instructions tant qu'une condition est vraie.*

Définition 1.11

*Une structure répétitive de type **while** permet de répéter une série d'instructions tant qu'une condition est vraie.*

```
while condition :  
    instructions
```

Définition 1.11

*Une structure répétitive de type **while** permet de répéter une série d'instructions tant qu'une condition est vraie.*

- *condition s'appelle **la condition d'arrêt**.*

```
while condition :  
    instructions
```

Définition 1.11

*Une structure répétitive de type **while** permet de répéter une série d'instructions tant qu'une condition est vraie.*

```
while condition :  
    instructions
```

- *condition* s'appelle **la condition d'arrêt**.
- tant que la condition d'arrêt vaut True, le programme exécute "instructions"

Définition 1.11

*Une structure répétitive de type **while** permet de répéter une série d'instructions tant qu'une condition est vraie.*

```
while condition :  
    instructions
```

- *condition* s'appelle **la condition d'arrêt**.
- tant que la condition d'arrêt vaut True, le programme exécute "instructions"
- normalement, "instructions" finira par modifier la valeur de *condition*, sinon l'algorithme tournera sans fin.

Définition 1.11

*Une structure répétitive de type **while** permet de répéter une série d'instructions tant qu'une condition est vraie.*

```
while condition :  
    instructions
```

- *condition* s'appelle **la condition d'arrêt**.
- tant que la condition d'arrêt vaut True, le programme exécute "instructions"
- normalement, "instructions" finira par modifier la valeur de *condition*, sinon l'algorithme tournera sans fin.

Dans ce type de structure :

- le nombre d'itérations dépend de la valeur d'une condition

Définition 1.11

*Une structure répétitive de type **while** permet de répéter une série d'instructions tant qu'une condition est vraie.*

```
while condition :  
    instructions
```

- *condition* s'appelle **la condition d'arrêt**.
- tant que la condition d'arrêt vaut True, le programme exécute "instructions"
- normalement, "instructions" finira par modifier la valeur de *condition*, sinon l'algorithme tournera sans fin.

Dans ce type de structure :

- le nombre d'itérations dépend de la valeur d'une condition
- ⇒ le nombre de répétitions n'est donc pas nécessairement connu à l'avance

Précisions syntaxiques en python:

Précisions syntaxiques en python:

En python, la boucle Tant que s'écrit `while` :

```
while condition :  
    instructions
```

Précisions syntaxiques en python:

En python, la boucle Tant que s'écrit `while` :

```
while condition :  
    instructions
```

Notez :

- les " :" après la condition

Précisions syntaxiques en python:

En python, la boucle Tant que s'écrit `while` :

```
while condition :  
    instructions
```

Notez :

- les " :" après la condition
- le décalage (avec tabulation) des instructions.



Exemple while

Que fait cet algorithme ?

```
somme = 0
nombre = input()
while nombre != -1 :
    somme = somme + nombre
    nombre = input()
print(somme)
```



Exemple while

Que fait cet algorithme ?

```
somme = 0
nombre = input()
while nombre != -1 :
    somme = somme + nombre
    nombre = input()
print(somme)
```

Cet algorithme demande à l'utilisateur d'entrer des nombres entiers au clavier et en fait la somme.



Exemple while

Que fait cet algorithme ?

```
somme = 0
nombre = input()
while nombre != -1 :
    somme = somme + nombre
    nombre = input()
print(somme)
```

Cet algorithme demande à l'utilisateur d'entrer des nombres entiers au clavier et en fait la somme.

Cette opération continue jusqu'à ce que l'utilisateur entre le nombre -1, l'algorithme affiche alors la somme obtenue.

Structure Pour (1/2)

Définition 1.12

*Une structure répétitive de type **pour** permet de répéter un nombre de fois connu à l'avance une série d'instructions.*

Définition 1.12

*Une structure répétitive de type **pour** permet de répéter un nombre de fois connu à l'avance une série d'instructions.*

```
for variable in ensemble_valeur :  
    instruction
```

Structure Pour (1/2)

Définition 1.12

*Une structure répétitive de type **pour** permet de répéter un nombre de fois connu à l'avance une série d'instructions.*

```
for variable in ensemble_valeur :  
    instruction
```

Remarques :

Définition 1.12

*Une structure répétitive de type **pour** permet de répéter un nombre de fois connu à l'avance une série d'instructions.*

```
for variable in ensemble_valeur :  
    instruction
```

Remarques :

- `variable` est une variable locale à la boucle, elle ne doit pas être modifiée par la série d'instructions.

Structure Pour (1/2)

Définition 1.12

*Une structure répétitive de type **pour** permet de répéter un nombre de fois connu à l'avance une série d'instructions.*

```
for variable in ensemble_valeur :  
    instruction
```

Remarques :

- `variable` est une variable locale à la boucle, elle ne doit pas être modifiée par la série d'instructions.
- à chaque tour de boucle, `variable` va varier en prenant dans l'ordre les valeurs de `ensemble_valeurs`.

Structure Pour (2/2)

Structure Pour (2/2)

Exemple :

```
for i in [3, 5, 10] :  
    print(i)
```

Structure Pour (2/2)

Exemple :

```
for i in [3, 5, 10] :  
    print(i)
```

- i va prendre successivement les valeurs 3, 5 et 10 :

Structure Pour (2/2)

Exemple :

```
for i in [3, 5, 10] :  
    print(i)
```

- i va prendre successivement les valeurs 3, 5 et 10 : il y a aura trois itérations de cette boucle.

Structure Pour (2/2)

Exemple :

```
for i in [3, 5, 10] :  
    print(i)
```

- i va prendre successivement les valeurs 3, 5 et 10 : il y a aura trois itérations de cette boucle.

Il est possible de spécifier qu'une variable prenne toutes les valeurs dans une plage d'entiers donnée :

Structure Pour (2/2)

Exemple :

```
for i in [3, 5, 10] :  
    print(i)
```

- i va prendre successivement les valeurs 3, 5 et 10 : il y a aura trois itérations de cette boucle.

Il est possible de spécifier qu'une variable prenne toutes les valeurs dans une plage d'entiers donnée :

```
for i in range(deb, fin) :  
    instructions
```

Structure Pour (2/2)

Exemple :

```
for i in [3, 5, 10] :  
    print(i)
```

- i va prendre successivement les valeurs 3, 5 et 10 : il y a aura trois itérations de cette boucle.

Il est possible de spécifier qu'une variable prenne toutes les valeurs dans une plage d'entiers donnée :

```
for i in range(deb, fin) :  
    instructions
```

Attention !

En python, la borne de fin est **exclue** !

Structure Pour (2/2)

Exemple :

```
for i in [3, 5, 10] :  
    print(i)
```

- i va prendre successivement les valeurs 3, 5 et 10 : il y a aura trois itérations de cette boucle.

Il est possible de spécifier qu'une variable prenne toutes les valeurs dans une plage d'entiers donnée :

```
for i in range(deb, fin) :  
    instructions
```

Attention !

En python, la borne de fin est **exclude** ! i prendra donc les valeurs deb, deb+1, ..., fin-1.

Structure Pour (2/2)

Exemple :

```
for i in [3, 5, 10] :  
    print(i)
```

- i va prendre successivement les valeurs 3, 5 et 10 : il y a aura trois itérations de cette boucle.

Il est possible de spécifier qu'une variable prenne toutes les valeurs dans une plage d'entiers donnée :

```
for i in range(deb, fin) :  
    instructions
```

Attention !

En python, la borne de fin est **exclude** ! i prendra donc les valeurs deb, deb+1, ..., fin-1.



Exemple Structure Pour

Qu'affiche l'algorithme suivant ?

```
somme = 0
```

```
for i in range (1,11) :
```

```
    somme = somme+i
```

```
print (somme)
```



Exemple Structure Pour

Qu'affiche l'algorithme suivant ?

```
somme = 0
```

```
for i in range (1,11) :
```

```
    somme = somme+i
```

```
print (somme)
```

Cet algorithme affiche la somme des entiers de 1 à 10 inclus : $1 + 2 + \dots + 10 =$



Exemple Structure Pour

Qu'affiche l'algorithme suivant ?

```
somme = 0
```

```
for i in range (1,11) :
```

```
    somme = somme+i
```

```
print (somme)
```

Cet algorithme affiche la somme des entiers de 1 à 10 inclus : $1 + 2 + \dots + 10 = 55$.

Lien entre structure Tant que et Pour

Lien entre structure Tant que et Pour

Il est facile de transformer une boucle pour en une boucle tant que :

Lien entre structure Tant que et Pour

Il est facile de transformer une boucle pour en une boucle tant que :

```
for i in range(1,10) :  
    instructions
```

Lien entre structure Tant que et Pour

Il est facile de transformer une boucle pour en une boucle tant que :

```
for i in range(1,10) :  
    instructions
```

→

Lien entre structure Tant que et Pour

Il est facile de transformer une boucle pour en une boucle tant que :

```
for i in range(1,10) :  
    instructions
```

→

```
i = 1  
while i < 10 :  
    instructions  
    i = i+1
```

Lien entre structure Tant que et Pour

Il est facile de transformer une boucle pour en une boucle tant que :

```
for i in range(1,10) :  
    instructions
```

→

```
i = 1  
while i < 10 :  
    instructions  
    i = i+1
```

L'inverse est loin d'être aussi simple et surtout « propre » :

Lien entre structure Tant que et Pour

Il est facile de transformer une boucle pour en une boucle tant que :

```
for i in range(1,10) :  
    instructions
```

→

```
i = 1  
while i < 10 :  
    instructions  
    i = i+1
```

L'inverse est loin d'être aussi simple et surtout « propre » :

```
i = 1  
j = 0  
while i <= 10 and j < i*2 :  
    instructions  
    i = i+1  
    j = i+j
```

Lien entre structure Tant que et Pour

Il est facile de transformer une boucle pour en une boucle tant que :

```
for i in range(1,10) :  
    instructions
```

→

```
i = 1  
while i < 10 :  
    instructions  
    i = i+1
```

L'inverse est loin d'être aussi simple et surtout « propre » :

```
i = 1  
j = 0  
while i <= 10 and j < i*2 :  
    instructions  
    i = i+1  
    j = i+j
```

Lien entre structure Tant que et Pour

Il est facile de transformer une boucle pour en une boucle tant que :

```
for i in range(1,10) :  
    instructions
```

→

```
i = 1  
while i < 10 :  
    instructions  
    i = i+1
```

L'inverse est loin d'être aussi simple et surtout « propre » :

```
i = 1  
j = 0  
while i <= 10 and j < i*2 :  
    instructions  
    i = i+1  
    j = i+j
```

→

```
j = 0  
for i in range(1,11) :  
    instructions  
    j = i+j  
    if j >= i*2 :  
        i = 11
```



Exercice

Écrire deux programmes permettant d'afficher les tables de multiplications de 1 à 10, l'un utilisant des boucles `pour`, l'autre utilisant des boucles `tant que`.



Exercice

Écrire deux programmes permettant d'afficher les tables de multiplications de 1 à 10, l'un utilisant des boucles `pour`, l'autre utilisant des boucles `tant que`.

```
for i in range(1,11) :  
    for j in range(1,11) :  
        print(i, "x", j, "=", i*j)
```



Exercice

Écrire deux programmes permettant d'afficher les tables de multiplications de 1 à 10, l'un utilisant des boucles `pour`, l'autre utilisant des boucles `tant que`.

```
for i in range(1,11) :  
    for j in range(1,11) :  
        print(i, "x", j, "=", i*j)
```

```
i = 1  
while i < 11 :  
    j = 1  
    while j < 11 :  
        print(i, "x", j, "=",  
              i*j)  
        j = j+1  
    i = i+1
```

- 1 Concepts de bases
- 2 Structuration d'un programme**
- 3 Données structurées

Structuration d'un programme

Structuration d'un programme

Motivation

Motivation

- Lorsque la taille d'un programme augmente, sa **lisibilité** devient de plus en plus difficile.

Structuration d'un programme

Motivation

- Lorsque la taille d'un programme augmente, sa **lisibilité** devient de plus en plus difficile.
- En outre, si certaines parties du programme sont répétées cela rend plus complexe la **maintenance** et cela nuit à la **réutilisabilité** de votre code.

Structuration d'un programme

Motivation

- Lorsque la taille d'un programme augmente, sa **lisibilité** devient de plus en plus difficile.
- En outre, si certaines parties du programme sont répétées cela rend plus complexe la **maintenance** et cela nuit à la **réutilisabilité** de votre code.

Objectif

Rendre vos programmes concis, clairs, compréhensifs, maintenables et réutilisables

Structuration d'un programme

Motivation

- Lorsque la taille d'un programme augmente, sa **lisibilité** devient de plus en plus difficile.
- En outre, si certaines parties du programme sont répétées cela rend plus complexe la **maintenance** et cela nuit à la **réutilisabilité** de votre code.

Objectif

Rendre vos programmes concis, clairs, compréhensifs, maintenables et réutilisables

Solution

Décomposer un programme en **sous-programmes**

Exemple

Exemple

```
# Volume d'un cylindre  
r = 3  
h = 2.3  
v = 3.14 * r * r * h  
print("Le volume du  
      cylindre vaut", v)
```

Exemple

```
# Volume d'un cylindre  
r = 3  
h = 2.3  
v = 3.14 * r * r * h  
print("Le volume du  
      cylindre vaut", v)
```

→

Exemple

```
# Volume d'un cylindre  
r = 3  
h = 2.3  
v = 3.14 * r * r * h  
print("Le volume du  
      cylindre vaut", v)
```

→

```
def aire_disque(r) :  
    return 3.14*r*r
```

Exemple

```
# Volume d'un cylindre
r = 3
h = 2.3
v = 3.14 * r * r * h
print("Le volume du
      cylindre vaut", v)
```

```
def aire_disque(r) :
    return 3.14*r*r
```

```
# Volume d'un cylindre
→ r = 3
h = 2.3
a = aire_disque(r)
v = a * h
print("Le volume du
      cylindre vaut", v)
```

Sous-programme

Définition 2.1

Un sous programme est une série d'instructions réalisant des traitements en fonction de données :

- *les données sont appelées arguments ou paramètres du sous-programme,*
- *le résultat du traitement est appelé valeur de retour du sous-programme.*

Définition 2.1

Un sous programme est une série d'instructions réalisant des traitements en fonction de données :

- *les données sont appelées arguments ou paramètres du sous-programme,*
- *le résultat du traitement est appelé valeur de retour du sous-programme.*

Avant d'être utilisé (c'est-à-dire appelé) un sous programme doit être déclaré.

Déclaration d'un sous-programme (1/2)

Déclaration d'un sous-programme (1/2)

Définition 2.2

La déclaration d'un sous-programme comporte :

Déclaration d'un sous-programme (1/2)

Définition 2.2

La déclaration d'un sous-programme comporte :

- une **signature** qui indique :

Définition 2.2

La déclaration d'un sous-programme comporte :

- une **signature** qui indique :
 - 1 le nom du sous-programme

Définition 2.2

La déclaration d'un sous-programme comporte :

- une **signature** qui indique :
 - 1 le nom du sous-programme
 - 2 le nom (local) des paramètres à fournir

Définition 2.2

La déclaration d'un sous-programme comporte :

- une **signature** qui indique :
 - ① *le nom du sous-programme*
 - ② *le nom (local) des paramètres à fournir*
Elle peut éventuellement indiquer (en commentaire) :
 - ③ *le rôle du sous-programme*

Définition 2.2

La déclaration d'un sous-programme comporte :

- une **signature** qui indique :

- ① le nom du sous-programme

- ② le nom (local) des paramètres à fournir

Elle peut éventuellement indiquer (en commentaire) :

- ③ le rôle du sous-programme

- ④ les pré-conditions : conditions que doivent remplir les données avant le début de l'algorithme

Définition 2.2

La déclaration d'un sous-programme comporte :

- une **signature** qui indique :

- ① le nom du sous-programme

- ② le nom (local) des paramètres à fournir

Elle peut éventuellement indiquer (en commentaire) :

- ③ le rôle du sous-programme

- ④ les pré-conditions : conditions que doivent remplir les données avant le début de l'algorithme

- ⑤ les post-conditions : conditions que doivent remplir les résultats après réalisation de l'algorithme

Déclaration d'un sous-programme (1/2)

Définition 2.2

La déclaration d'un sous-programme comporte :

- une **signature** qui indique :
 - ① le nom du sous-programme
 - ② le nom (local) des paramètres à fournir
Elle peut éventuellement indiquer (en commentaire) :
 - ③ le rôle du sous-programme
 - ④ les pré-conditions : conditions que doivent remplir les données avant le début de l'algorithme
 - ⑤ les post-conditions : conditions que doivent remplir les résultats après réalisation de l'algorithme
- un **corps** qui indique la description des instructions à réaliser

Déclaration d'un sous-programme (1/2)

Définition 2.2

La déclaration d'un sous-programme comporte :

- une **signature** qui indique :
 - ① le nom du sous-programme
 - ② le nom (local) des paramètres à fournir
Elle peut éventuellement indiquer (en commentaire) :
 - ③ le rôle du sous-programme
 - ④ les pré-conditions : conditions que doivent remplir les données avant le début de l'algorithme
 - ⑤ les post-conditions : conditions que doivent remplir les résultats après réalisation de l'algorithme
- un **corps** qui indique la description des instructions à réaliser

Remarque

Les paramètres d'un sous-programmes sont représentés par des noms quelconques appelés **paramètres formels**.

Exemple déclaration sous-programme (python)

```
def perimetre(rayon) :  
    """  
        Fonction calculant le périmètre  
        d'un cercle à partir de son rayon  
  
        Parameters :  
            rayon (float) : doit être positif  
  
        Returns :  
            float : le périmètre du cercle  
    """  
  
    p = 2 * 3.14 * rayon  
    return p
```

Cas particuliers

- Si un sous-programme n'a pas besoin de retourner une valeur alors:

- Si un sous-programme n'a pas besoin de retourner une valeur alors:
 - nous omettons l'instruction `return`

Cas particuliers

- Si un sous-programme n'a pas besoin de retourner une valeur alors:
 - nous omettons l'instruction `return`
- Si un sous-programme n'a pas de paramètre alors nous laisserons les parenthèses vides

Exemple

```
def afficher_bonjour() :  
    print(" Bonjour !")
```

Appel d'un sous-programme

Définition 2.3

*L'appel d'un sous-programme est **l'invocation** du sous-programme par son nom en fournissant des données en nombre requis.*

Définition 2.3

*L'appel d'un sous-programme est **l'invocation** du sous-programme par son nom en fournissant des données en nombre requis.*

- Les données fournies sont appelées **paramètres effectifs**.

Définition 2.3

*L'appel d'un sous-programme est **l'invocation** du sous-programme par son nom en fournissant des données en nombre requis.*

- Les données fournies sont appelées **paramètres effectifs**.
- Les paramètres effectifs peuvent être : des constantes, des valeurs littérales, des expressions, des valeurs de variables, ...

Définition 2.3

*L'appel d'un sous-programme est **l'invocation** du sous-programme par son nom en fournissant des données en nombre requis.*

- Les données fournies sont appelées **paramètres effectifs**.
- Les paramètres effectifs peuvent être : des constantes, des valeurs littérales, des expressions, des valeurs de variables, ...
- Dans le cas où un paramètre effectif est une variable :

Définition 2.3

*L'appel d'un sous-programme est l'**invocation** du sous-programme par son nom en fournissant des données en nombre requis.*

- Les données fournies sont appelées **paramètres effectifs**.
- Les paramètres effectifs peuvent être : des constantes, des valeurs littérales, des expressions, des valeurs de variables, ...
- Dans le cas où un paramètre effectif est une variable :
 - la variable doit être initialisée

Définition 2.3

*L'appel d'un sous-programme est l'**invocation** du sous-programme par son nom en fournissant des données en nombre requis.*

- Les données fournies sont appelées **paramètres effectifs**.
- Les paramètres effectifs peuvent être : des constantes, des valeurs littérales, des expressions, des valeurs de variables, ...
- Dans le cas où un paramètre effectif est une variable :
 - la variable doit être initialisée
 - le sous-programme ne peut que lire la variable (donc il ne pourra pas modifier sa valeur)

Exemple

```
def p_cercle(rayon) :  
    p = 2*3.14*rayon  
  
print("Quel est le rayon ?")  
input(r)  
peri = p_cercle(r)  
print("Le périmètre est :",peri)
```

Exemple

```
def p_cercle(rayon) :  
    p = 2*3.14*rayon  
  
print("Quel est le rayon ?")  
input(r)  
peri = p_cercle(r)  
print("Le périmètre est :", peri)
```

Notez :

- appel du sous-programme `p_cercle` avec passage du paramètre effectif *r* et récupération du résultat dans *peri*

Exemple

```
def p_cercle(rayon) :  
    p = 2*3.14*rayon  
  
print("Quel est le rayon ?")  
input(r)  
peri = p_cercle(r)  
print("Le périmètre est :", peri)
```

Notez :

- appel du sous-programme `p_cercle` avec passage du paramètre effectif *r* et récupération du résultat dans *peri*
- la valeur de retour (*p*) est retournée à l'instruction qui a appelée le sous-programme. (ici l'affectation à *peri*)

Définition 2.4

La portée d'une donnée désigne son niveau de visibilité :

Définition 2.4

La portée d'une donnée désigne son niveau de visibilité :

- *Les données (variables, constantes et paramètres formels) déclarées dans un sous programme ont une **portée locale** à ce sous programme (c'est-à-dire que la donnée n'est pas visible en dehors du sous-programme).*

Définition 2.4

La portée d'une donnée désigne son niveau de visibilité :

- *Les données (variables, constantes et paramètres formels) déclarées dans un sous programme ont une **portée locale** à ce sous programme (c'est-à-dire que la donnée n'est pas visible en dehors du sous-programme).*
- *Les autres données auront une **portée globale** (c'est-à-dire que la donnée est visible à n'importe quel endroit du programme).*

Définition 2.4

La portée d'une donnée désigne son niveau de visibilité :

- *Les données (variables, constantes et paramètres formels) déclarées dans un sous programme ont une **portée locale** à ce sous programme (c'est-à-dire que la donnée n'est pas visible en dehors du sous-programme).*
- *Les autres données auront une **portée globale** (c'est-à-dire que la donnée est visible à n'importe quel endroit du programme).*

Un bon algorithme **ne doit jamais** utiliser de variables globales dans un sous-programme.

Définition 2.4

La portée d'une donnée désigne son niveau de visibilité :

- *Les données (variables, constantes et paramètres formels) déclarées dans un sous programme ont une **portée locale** à ce sous programme (c'est-à-dire que la donnée n'est pas visible en dehors du sous-programme).*
- *Les autres données auront une **portée globale** (c'est-à-dire que la donnée est visible à n'importe quel endroit du programme).*

Un bon algorithme **ne doit jamais** utiliser de variables globales dans un sous-programme. *Il peut en revanche utiliser des constantes.*



Exercices

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```



Exercices

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```

Qu'affiche l'algorithme ci-dessus ?



Exercices

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```

Qu'affiche l'algorithme ci-dessus ?

Erreur : a n'est pas définie

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(b)
```



Exercices

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```

Qu'affiche l'algorithme ci-dessus ?

Erreur : a n'est pas définie

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(b)
```

Qu'affiche l'algorithme ci-dessus ?



Exercices

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```

Qu'affiche l'algorithme ci-dessus ?

Erreur : a n'est pas définie

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(b)
```

Qu'affiche l'algorithme ci-dessus ?

Erreur : b n'est pas définie

```
a = 4  
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```



Exercices

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```

Qu'affiche l'algorithme ci-dessus ?

Erreur : a n'est pas définie

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(b)
```

Qu'affiche l'algorithme ci-dessus ?

Erreur : b n'est pas définie

```
a = 4  
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```

Qu'affiche l'algorithme ci-dessus ?



Exercices

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```

Qu'affiche l'algorithme ci-dessus ?

Erreur : a n'est pas définie

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(b)
```

Qu'affiche l'algorithme ci-dessus ?

Erreur : b n'est pas définie

```
a = 4  
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```

Qu'affiche l'algorithme ci-dessus ?

L'algorithme affichera 4.

```
b = 4  
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(b)
```



Exercices

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```

Qu'affiche l'algorithme ci-dessus ?

Erreur : a n'est pas définie

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(b)
```

Qu'affiche l'algorithme ci-dessus ?

Erreur : b n'est pas définie

```
a = 4  
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```

Qu'affiche l'algorithme ci-dessus ?

L'algorithme affichera 4.

```
b = 4  
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(b)
```

Qu'affiche l'algorithme ci-dessus ?



Exercices

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```

Qu'affiche l'algorithme ci-dessus ?

Erreur : a n'est pas définie

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(b)
```

Qu'affiche l'algorithme ci-dessus ?

Erreur : b n'est pas définie

```
a = 4  
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```

Qu'affiche l'algorithme ci-dessus ?

L'algorithme affichera 4.

```
b = 4  
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(b)
```

Qu'affiche l'algorithme ci-dessus ?

L'algorithme affichera 4.

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 Données structurées**
 - Tableaux
 - Dictionnaires
 - Autres types de données structurées

Données scalaire et structurée

Définition 3.1

Un type de données est dit :

- **scalaire** lorsqu'il permet de stocker une seule information (on parle aussi de type atomique) : entiers, réels, caractères et booléens.

Définition 3.1

Un type de données est dit :

- **scalaire** lorsqu'il permet de stocker une seule information (on parle aussi de type atomique) : entiers, réels, caractères et booléens.
- **structuré** lorsqu'il permet de stocker plusieurs informations (on parle alors de type composite) : chaînes de caractères, tableaux, dictionnaires.

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 Données structurées**
 - **Tableaux**
 - Dictionnaires
 - Autres types de données structurées

Définition 3.2

Un **tableau** est une structure de données permettant de représenter une collection d'éléments de même type :

Définition 3.2

Un **tableau** est une structure de données permettant de représenter une collection d'éléments de même type :

- les éléments sont rangés consécutivement ;

Définition 3.2

Un **tableau** est une structure de données permettant de représenter une collection d'éléments de même type :

- les éléments sont rangés consécutivement ;
- les éléments sont accessibles par un numéro d'ordre : leur **indice**.

En python

En python

- Affectation : `tableau = [1,3,5,2,-3]`

En python

- Affectation : `tableau = [1,3,5,2,-3]`
- Accès (lecture et écriture) : `tableau[i]` avec `i` entier compris entre 0 et `len(tableau)-1`

En python

- Affectation : `tableau = [1,3,5,2,-3]`
- Accès (lecture et écriture) : `tableau[i]` avec `i` entier compris entre 0 et `len(tableau)-1`

En python, la taille des tableaux est modifiable. C'est une particularité : dans la plupart des autres langages, la taille des tableaux est fixe.

Exemple

```
def somme_des_element(tab) :  
    s = 0  
    for i in [2,5,6] :  
        s = s + tab[i]  
    return s
```

Compléments de commande en python 1/3

- $t_1 + t_2$: concaténation de t_1 et t_2 , mis bout-à-bout.

- $t_1 + t_2$: concaténation de t_1 et t_2 , mis bout-à-bout.

Exemple :

```
>>> [3,5] + [2,7,5]  
[3,5,2,7,5]
```

Compléments de commande en python 1/3

- $t_1 + t_2$: concaténation de t_1 et t_2 , mis bout-à-bout.

Exemple :

```
>>> [3,5] + [2,7,5]  
[3,5,2,7,5]
```

- $n * t$: concaténation de n occurrences de t ($t + \dots + t$)

Compléments de commande en python 1/3

- $t_1 + t_2$: concaténation de t_1 et t_2 , mis bout-à-bout.

Exemple :

```
>>> [3,5] + [2,7,5]  
[3,5,2,7,5]
```

- $n * t$: concaténation de n occurrences de t ($t + \dots + t$)

Exemple :

```
>>> 3*[1,4]  
[1,4,1,4,1,4]
```

```
>>> 7*[0]  
[0,0,0,0,0,0,0]
```

Compléments de commande en python 2/3

Compléments de commande en python 2/3

- `t[a:b]` : tableau des valeurs de `t` indicées de `a` à `b-1`

Compléments de commande en python 2/3

- `t[a:b]` : tableau des valeurs de `t` indicées de `a` à `b-1`
- `t[a:]` : tableau des valeurs de `t` indicées de `a` à la fin

Compléments de commande en python 2/3

- `t[a:b]` : tableau des valeurs de `t` indicées de `a` à `b-1`
- `t[a:]` : tableau des valeurs de `t` indicées de `a` à la fin
- `t[:b]` : tableau des valeurs de `t` indicées de 0 à `b-1`

Compléments de commande en python 2/3

- `t[a:b]` : tableau des valeurs de `t` indicées de `a` à `b-1`
- `t[a:]` : tableau des valeurs de `t` indicées de `a` à la fin
- `t[:b]` : tableau des valeurs de `t` indicées de 0 à `b-1`

Exemple :

```
>>> x = [-3,-2,-1,0,1,2,3,4,5,6,7]
>>> y = x[2:5] # clone le segment de l'indice 2 à 4
>>> y
[-1,0,1]

>>> y[1] = 18
>>> y
[-1,18,1]

>>> x
[-3,-2,-1,0,1,2,3,4,5,6,7] # x n'est pas modifié
```

Compléments de commande en python 3/3

- `in` : teste l'appartenance à un tableau

Compléments de commande en python 3/3

- `in` : teste l'appartenance à un tableau

```
>>> 2 in [4,2,8]
```

```
True
```

Compléments de commande en python 3/3

- `in` : teste l'appartenance à un tableau

```
>>> 2 in [4,2,8]
```

```
True
```

- `del t[a]` : supprime l'élément d'indice `a` du tableau `t`

Compléments de commande en python 3/3

- `in` : teste l'appartenance à un tableau

```
>>> 2 in [4,2,8]
```

```
True
```

- `del t[a]` : supprime l'élément d'indice `a` du tableau `t` `>>> x =`

```
[18,7,3,5,1,0,-2,9]
```

```
>>> del x[2]          # suppression de l'élément d'indice 2
```

```
>>> x
```

```
[18,7,5,1,0,-2,9]
```

Compléments de commande en python 3/3

- `in` : teste l'appartenance à un tableau

```
>>> 2 in [4,2,8]
```

```
True
```

- `del t[a]` : supprime l'élément d'indice `a` du tableau `t` `>>> x =`

```
[18,7,3,5,1,0,-2,9]
```

```
>>> del x[2]          # suppression de l'élément d'indice 2
```

```
>>> x
```

```
[18,7,5,1,0,-2,9]
```

- `del t[a:b]` : supprime les éléments d'indices `a` à `b-1` du tableau `t`

Compléments de commande en python 3/3

- `in` : teste l'appartenance à un tableau

```
>>> 2 in [4,2,8]
True
```

- `del t[a]` : supprime l'élément d'indice `a` du tableau `t` >>> `x =`

```
[18,7,3,5,1,0,-2,9]
>>> del x[2]          # suppression de l'élément d'indice 2
>>> x
[18,7,5,1,0,-2,9]
```

- `del t[a:b]` : supprime les éléments d'indices `a` à `b-1` du tableau `t`

```
>>> del x[4:]          # suppression des 3 derniers éléments
>>> x
[18,7,5,1]
```

Méthodes sur les tableaux

Définition 3.3

Une **méthode** (en *python*) est une fonction qui s'applique à un objet.

Définition 3.3

Une **méthode** (en *python*) est une fonction qui s'applique à un objet.

Syntaxe d'un appel : *nom_objet.nom_méthode(paramètres)*

Définition 3.3

Une **méthode** (en *python*) est une fonction qui s'applique à un objet.

Syntaxe d'un appel : `nom_objet.nom_méthode(paramètres)`

- `append` : ajoute un élément à la fin

Définition 3.3

Une **méthode** (en *python*) est une fonction qui s'applique à un objet.

Syntaxe d'un appel : *nom_objet.nom_méthode(paramètres)*

- **append** : ajoute un élément à la fin

```
>>> a = [2,5,7,3]
```

```
>>> a.append(12)
```

```
>>> a
```

```
[2,5,7,3,12]
```

Définition 3.3

Une **méthode** (en *python*) est une fonction qui s'applique à un objet.

Syntaxe d'un appel : *nom_objet.nom_méthode(paramètres)*

- **append** : ajoute un élément à la fin

```
>>> a = [2,5,7,3]
>>> a.append(12)
>>> a
[2,5,7,3,12]
```

- **pop** : supprime le dernier élément et le renvoie

Définition 3.3

Une **méthode** (en *python*) est une fonction qui s'applique à un objet.

Syntaxe d'un appel : *nom_objet.nom_méthode(paramètres)*

- **append** : ajoute un élément à la fin

```
>>> a = [2,5,7,3]
>>> a.append (12)
>>> a
[2,5,7,3,12]
```

- **pop** : supprime le dernier élément et le renvoie

```
>>> a.pop ()
12
>>> a
[2,5,7,3]
```

Méthodes sur les tableaux : compléments

- `reverse` : renverse l'ordre des éléments
- `extend(t)` : rajoute les éléments du tableau `t` à la fin du tableau courant
- `count(e)` : compte le nombre d'occurrences de `e` dans le tableau
- `index(e)` : renvoie l'indice de la première occurrence de `e` dans le tableau
- `insert(i,e)` : insert `e` à l'indice `i`
- `sort` : trie par ordre croissant

Parcours de tableau : for

Parcours de tableau : for

`for x in t` : crée une boucle dans laquelle `x` prend chaque valeur du tableau `t`

Parcours de tableau : for

for x in t : crée une boucle dans laquelle x prend chaque valeur du tableau t

```
t = [1,5,2]  
for x in t :  
    print (x)
```


Parcours de tableau : for

for x in t : crée une boucle dans laquelle x prend chaque valeur du tableau t

```
t = [1,5,2]
for x in t :
    print (x)
```

Affiche

1
5
2

Parcours de tableau : for

`for x in t` : crée une boucle dans laquelle `x` prend chaque valeur du tableau `t`

```
t = [1,5,2]
```

```
for x in t :
```

```
    print (x)
```

Affiche

1

5

2

Ex : Compter le nombre de valeurs paires dans la liste 1

Parcours de tableau : for

for x in t : crée une boucle dans laquelle x prend chaque valeur du tableau t

t = [1,5,2]	Affiche
for x in t :	1
print (x)	5
	2

Ex : Compter le nombre de valeurs paires dans la liste 1

```
def compte_pair(t) :  
    compteur = 0  
    for k in t :  
        if k%2 == 0 :  
            compteur+=1  
    return compteur
```

Parcours de tableau : for

for x in t : crée une boucle dans laquelle x prend chaque valeur du tableau t

```
t = [1,5,2]
for x in t :
    print (x)
```

Affiche

1
5
2

Ex : Compter le nombre de valeurs paires dans la liste 1

```
def compte_pair(t) :
    compteur = 0
    for k in t :
        if k%2 == 0 :
            compteur+=1
    return compteur
```

```
def compte_pair(t) :
    compteur = 0
    n = len(t)
    for i in range(n) :
        if t[i]%2 == 0 :
            compteur+=1
    return compteur
```

Parcours de tableau : for

for x in t : crée une boucle dans laquelle x prend chaque valeur du tableau t

```
t = [1,5,2]
for x in t :
    print (x)
```

Affiche

1
5
2

Ex : Compter le nombre de valeurs paires dans la liste 1

```
def compte_pair(t) :
    compteur = 0
    for k in t :
        if k%2 == 0 :
            compteur+=1
    return compteur
```

```
def compte_pair(t) :
    compteur = 0
    n = len(t)
    for i in range(n) :
        if t[i]%2 == 0 :
            compteur+=1
    return compteur
```

Remarque : list(range (n)) permet de créer un tableau contenant tous les éléments du range (attention, range ne crée pas un tableau)

Parcours de tableaux : compléments

`for (i,e) in enumerate (t) :` parcourt le tableau `t` en mémorisant dans `i` les indices successifs et dans `e` les éléments successifs.

```
t = [1,5,2]
```

```
for (i,e) in enumerate (t) :
```

```
    print ("L'element d'indice ", i, " du tableau est ", e)
```

Affiche :

L'element d'indice 0 du tableau est 1

L'element d'indice 1 du tableau est 5

L'element d'indice 2 du tableau est 2

Création de tableau

Création de tableau

On va illustrer les méthodes en créant le tableau $[1, 4, 9, 16, \dots, n^2]$.

Création de tableau

On va illustrer les méthodes en créant le tableau $[1, 4, 9, 16, \dots, n^2]$.

- Créer un tableau de longueur n et modifier un à un les éléments

```
n = ...  
t = n*[0]  
for i in range(n) :  
    t[i] = (i+1)**2
```

Création de tableau

On va illustrer les méthodes en créant le tableau $[1, 4, 9, 16, \dots, n^2]$.

- Créer un tableau de longueur n et modifier un à un les éléments

```
n = ...  
t = n*[0]  
for i in range(n) :  
    t[i] = (i+1)**2
```

- Créer un tableau vide et insérer un à un les éléments

```
n = ...  
t = []  
for i in range(1, n+1) :  
    t.append(i**2)
```

Création de tableau

On va illustrer les méthodes en créant le tableau $[1, 4, 9, 16, \dots, n^2]$.

- Créer un tableau de longueur n et modifier un à un les éléments

```
n = ...  
t = n*[0]  
for i in range(n) :  
    t[i] = (i+1)**2
```

- Créer un tableau vide et insérer un à un les éléments

```
n = ...  
t = []  
for i in range(1, n+1) :  
    t.append(i**2)
```

- Utiliser une *list comprehension* : $[expr \text{ for } i \text{ in range}(\dots)]$ crée le tableau des valeurs successives de $expr$ lorsque i parcourt le range

```
n = ...  
t = [i**2 for i in range(1, n+1)]
```

Le pb de la duplication

Le pb de la duplication

```
>>> x = [4,2,5]
>>> y = x
>>> y += [3,9]
>>> x
>>> [4,2,5,3,9]
```

Le pb de la duplication

```
>>> x = [4,2,5]
>>> y = x
>>> y += [3,9]
>>> x
>>> [4,2,5,3,9]
```

Explications :

- si `x` a pour valeur un tableau, `x` contient seulement l'adresse de l'emplacement mémoire du tableau (référence).

Le pb de la duplication

```
>>> x = [4,2,5]
>>> y = x
>>> y += [3,9]
>>> x
>>> [4,2,5,3,9]
```

Explications :

- si x a pour valeur un tableau, x contient seulement l'adresse de l'emplacement mémoire du tableau (référence).
- $y = x$ implique la copie de l'adresse uniquement, donc les deux variables référencent le même tableau

Le pb de la duplication

```
>>> x = [4,2,5]
>>> y = x
>>> y += [3,9]
>>> x
>>> [4,2,5,3,9]
```

Explications :

- si x a pour valeur un tableau, x contient seulement l'adresse de l'emplacement mémoire du tableau (référence).
- $y = x$ implique la copie de l'adresse uniquement, donc les deux variables référencent le même tableau
- conséquence : une modification de x ou de y modifie les deux variables

Alternative : dupliquer ou cloner le tableau

2 types d'égalités

- égalité structurelle : les éléments sont égaux 2 à 2, testable avec "=="
- égalité physique : référencé au même endroit, une modification de l'un modifie l'autre, testable avec "is"

Rq : égalité physique \Rightarrow égalité structurelle

Solutions pour éviter l'égalité physique

- `y = list(x)`
- `y = x[:]`

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 Données structurées**
 - Tableaux
 - Dictionnaires**
 - Autres types de données structurées

Limite des tableaux

Problème : on souhaite enregistrer des informations (par exemple l'âge) sur des personnes (représentées par leur nom).

Problème : on souhaite enregistrer des informations (par exemple l'âge) sur des personnes (représentées par leur nom).

→ il faut un type de données structuré → on peut par exemple utiliser 2 tableaux : un pour les noms et un pour les âges avec une correspondance entre indice.

Problème : on souhaite enregistrer des informations (par exemple l'âge) sur des personnes (représentées par leur nom).

→ il faut un type de données structuré → on peut par exemple utiliser 2 tableaux : un pour les noms et un pour les âges avec une correspondance entre indice.

Mais : l'accès et la modification des informations nécessite une recherche coûteuse en temps.

Solution : utiliser des dictionnaires !

Définition 3.4

Un **dictionnaire** (*python*) est une table d'association clé-valeur.
Les clés sont toutes d'un même type et doivent être comparables.

Définition 3.4

Un **dictionnaire** (*python*) est une table d'association clé-valeur.
Les clés sont toutes d'un même type et doivent être comparables.

Exemple

```
>>> d = {"Julien" : 18, "Virginie" : 19, "Matéo" : 18}
>>> d["Virginie"]
19
>>> d["Chloé"] = 21
>>> d
{"Julien" : 18, "Virginie" : 19, "Matéo" : 18, "Chloé" : 21}
```

Fonctions et méthodes sur les dictionnaires

taille d'un dictionnaire : `len(d)`

taille d'un dictionnaire : `len(d)`

accès à la valeur d'une clé : `d[c]`

Fonctions et méthodes sur les dictionnaires

taille d'un dictionnaire : `len(d)`

accès à la valeur d'une clé : `d[c]`

présence d'une clé : `c in d`

Fonctions et méthodes sur les dictionnaires

taille d'un dictionnaire : `len(d)`

accès à la valeur d'une clé : `d[c]`

présence d'une clé : `c in d`

ajout d'une association : `d[c] = v`

Fonctions et méthodes sur les dictionnaires

taille d'un dictionnaire : `len(d)`

accès à la valeur d'une clé : `d[c]`

présence d'une clé : `c in d`

ajout d'une association : `d[c] = v`

parcourir un dictionnaire

```
for c in d :  
    print(d[c])
```

faire une copie de d1 : **Attention !**

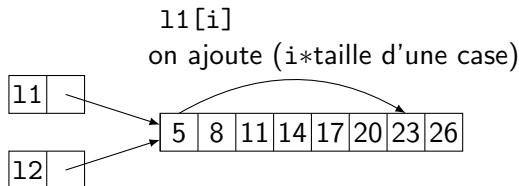
~~d2 = d1~~ (sinon les dictionnaires sont dépendants comme les tableaux)

solution : `d2 = copy(d1)`

Fonctionnement des tableaux et dictionnaires

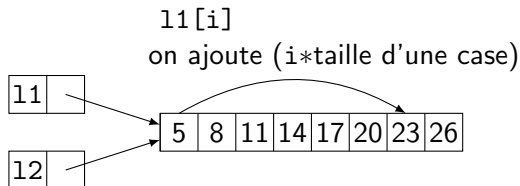
Fonctionnement des tableaux et dictionnaires

- Modèle mémoire des tableaux : cases mémoires consécutives



Fonctionnement des tableaux et dictionnaires

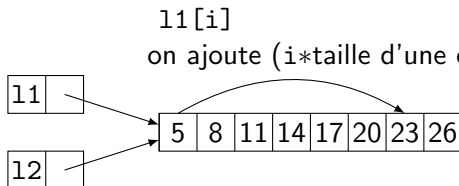
- Modèle mémoire des tableaux : cases mémoires consécutives



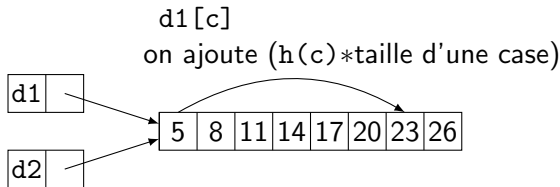
- Modèle mémoire des dictionnaires : **table de hachage**

Fonctionnement des tableaux et dictionnaires

- Modèle mémoire des tableaux : cases mémoires consécutives



- Modèle mémoire des dictionnaires : **table de hachage**
→ un dictionnaire est associé à une **fonction de hachage** h .



Quand utiliser un dictionnaire ?

Quand utiliser un dictionnaire ?

Un tableau peut être vu comme un dictionnaire dont les clés sont $\llbracket 0, n \rrbracket$

Quand utiliser un dictionnaire ?

Un tableau peut être vu comme un dictionnaire dont les clés sont $\llbracket 0, n \rrbracket$

- si besoin de clés de types non entier
 - `string`
 - `float`
 - tous les types immuables (Contre-exemple : pas les listes)

Quand utiliser un dictionnaire ?

Un tableau peut être vu comme un dictionnaire dont les clés sont $\llbracket 0, n \rrbracket$

- si besoin de clés de types non entier
 - `string`
 - `float`
 - tous les types immuables (Contre-exemple : pas les listes)
- si besoin d'un ensemble de clés entières :
 - négatives
 - non-consécutives

Quand utiliser un dictionnaire ?

Un tableau peut être vu comme un dictionnaire dont les clés sont $\llbracket 0, n \rrbracket$

- si besoin de clés de types non entier
 - string
 - float
 - tous les types immuables (Contre-exemple : pas les listes)
- si besoin d'un ensemble de clés entières :
 - négatives
 - non-consécutives

Remarque : on peut implémenter un dictionnaire en utilisant un tableau de couples (clé,valeur), mais les opérations seront moins efficaces.

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 Données structurées**
 - Tableaux
 - Dictionnaires
 - **Autres types de données structurées**

Tuples

Tuples

Différences tuples - tableau

Différences tuples - tableau

- on ne peut pas modifier, supprimer ou insérer un élément d'un tuple

Différences tuples - tableau

- on ne peut pas modifier, supprimer ou insérer un élément d'un tuple
- délimité par des parenthèses, pas par des crochets

Différences tuples - tableau

- on ne peut pas modifier, supprimer ou insérer un élément d'un tuple
- délimité par des parenthèses, pas par des crochets

```
>>> t = (7,2,9)
>>> len(t)
3
>>> class(t)
<class 'tuple'>
>>> t[1]
2
>>> u = (3,)          # tuple de longueur 1
```

Tuples

Différences tuples - tableau

- on ne peut pas modifier, supprimer ou insérer un élément d'un tuple
- délimité par des parenthèses, pas par des crochets

```
>>> t = (7,2,9)
>>> len(t)
3
>>> class(t)
<class 'tuple'>
>>> t[1]
2
>>> u = (3,)      # tuple de longueur 1
```

Remarques :

Tuples

Différences tuples - tableau

- on ne peut pas modifier, supprimer ou insérer un élément d'un tuple
- délimité par des parenthèses, pas par des crochets

```
>>> t = (7,2,9)
>>> len(t)
3
>>> class(t)
<class 'tuple'>
>>> t[1]
2
>>> u = (3,)          # tuple de longueur 1
```

Remarques :

- on peut concaténer les tuples

Tuples

Différences tuples - tableau

- on ne peut pas modifier, supprimer ou insérer un élément d'un tuple
- délimité par des parenthèses, pas par des crochets

```
>>> t = (7,2,9)
>>> len(t)
3
>>> class(t)
<class 'tuple'>
>>> t[1]
2
>>> u = (3,)          # tuple de longueur 1
```

Remarques :

- on peut concaténer les tuples
- couple : tuple de longueur 2

Tuples

Différences tuples - tableau

- on ne peut pas modifier, supprimer ou insérer un élément d'un tuple
- délimité par des parenthèses, pas par des crochets

```
>>> t = (7,2,9)
>>> len(t)
3
>>> class(t)
<class 'tuple'>
>>> t[1]
2
>>> u = (3,)          # tuple de longueur 1
```

Remarques :

- on peut concaténer les tuples
- couple : tuple de longueur 2
- souvent utiles pour renvoyer plusieurs valeurs.

Chaînes de caractères

Chaînes de caractères

Chaînes : similaires aux tuples - MAIS - chaque élément est une chaîne de longueur 1

Chaînes de caractères

Chaînes : similaires aux tuples - MAIS - chaque élément est une chaîne de longueur 1

- **Conséquences** : pas de modification, suppression ou insertion de lettre

Chaînes de caractères

Chaînes : similaires aux tuples - MAIS - chaque élément est une chaîne de longueur 1

- **Conséquences** : pas de modification, suppression ou insertion de lettre
- " " ou ' ' : pour pouvoir considérer l'un de ces symboles comme un symbole normal. ("l'espoir", mais pas 'l'espoir')

Chaînes de caractères

Chaînes : similaires aux tuples - MAIS - chaque élément est une chaîne de longueur 1

- **Conséquences** : pas de modification, suppression ou insertion de lettre
- " " ou ' ' : pour pouvoir considérer l'un de ces symboles comme un symbole normal. ("l'espoir", mais pas 'l'espoir')

```
>>> s = 'le lac'
```

```
>>> s[1]
```

```
'e'
```

```
>>> for c in s :
```

```
...     print(c)
```

```
l
```

```
e
```

```
l
```

```
a
```

```
c
```

```
>>> len(s)
```

```
6
```

Symboles particuliers pour les chaînes en python

Symboles particuliers pour les chaînes en python

Codage	Interprétation
\\	\
\'	'
\"	"
\n	saut de ligne
\t	tabulation horizontale
\r	retour chariot

Symboles particuliers pour les chaînes en python

Codage	Interprétation
\\	\
\/	'
\"	"
\n	saut de ligne
\t	tabulation horizontale
\r	retour chariot

Exemple :

```
>>> s = 'Je demandais au Python :\ n\ t- Le lion ou l\
'antilope ?'
```

```
>>> print(s)
```

Je demandais au Python :

- Le lion ou l'antilope ?

Fonctions et méthodes sur les chaînes

Fonctions et méthodes sur les chaînes

- `join` : concatène les chaînes contenues dans une liste en intercalant un séparateur donné

```
>>> ch = ','.join(['Nifnif','Nafnaf','Noufnouf'])  
>>> ch  
'Nifnif,Nafnaf,Noufnouf'
```

Fonctions et méthodes sur les chaînes

- `join` : concatène les chaînes contenues dans une liste en intercalant un séparateur donné

```
>>> ch = ','.join(['Nifnif','Nafnaf','Noufnouf'])
>>> ch
'Nifnif,Nafnaf,Noufnouf'
```

- `split` : sépare une chaîne en sous-chaînes séparées par un séparateur donné

```
>>> ch.split(',')
['Nifnif','Nafnaf','Noufnouf']
```

Fonctions et méthodes sur les chaînes

- `join` : concatène les chaînes contenues dans une liste en intercalant un séparateur donné

```
>>> ch = ','.join(['Nifnif','Nafnaf','Noufnouf'])
>>> ch
'Nifnif,Nafnaf,Noufnouf'
```

- `split` : sépare une chaîne en sous-chaînes séparées par un séparateur donné

```
>>> ch.split(',')
['Nifnif','Nafnaf','Noufnouf']
```

- `list` : transforme la chaîne en liste de chaînes de longueur 1

```
>>> s = 'gorille'
>>> l = list(s)
>>> l
['g','o','r','i','l','l','e']
>>> l[0] = 'G'
>>> ''.join(l)
'Gorille'
```