Initiation au langage Python

Mathias Ettinger

LP ESSIG

2013 – 2014

Introduction

Pourquoi Python ?

- Langage de programmation (comme beaucoup d'autres).

Pourquoi Python ?

- Langage de programmation (comme beaucoup d'autres).
- Portable et libre (non lié à un environnement particulier).

Pourquoi Python ?

- Langage de programmation (comme beaucoup d'autres).
- Portable et libre (non lié à un environnement particulier).
- Dynamique (interprété) ou compilé.

Pourquoi Python ?

- Langage de programmation (comme beaucoup d'autres).
- Portable et libre (non lié à un environnement particulier).
- Dynamique (interprété) ou compilé.
- Extensible : il s'interface avec d'autres langages ou logiciels.

Pourquoi Python ?

- Langage de programmation (comme beaucoup d'autres).
- Portable et libre (non lié à un environnement particulier).
- Dynamique (interprété) ou compilé.
- Extensible : il s'interface avec d'autres langages ou logiciels.
- Orthogonal : un petit nombre de concepts engendrent des constructions très riches.

Pourquoi Python ?

- Langage de programmation (comme beaucoup d'autres).
- Portable et libre (non lié à un environnement particulier).
- Dynamique (interprété) ou compilé.
- Extensible : il s'interface avec d'autres langages ou logiciels.
- Orthogonal : un petit nombre de concepts engendrent des constructions très riches.
- Dynamiquement (mais fortement) typé.

Pourquoi Python ?

- Langage de programmation (comme beaucoup d'autres).
- Portable et libre (non lié à un environnement particulier).
- Dynamique (interprété) ou compilé.
- Extensible : il s'interface avec d'autres langages ou logiciels.
- Orthogonal : un petit nombre de concepts engendrent des constructions très riches.
- Dynamiquement (mais fortement) typé.
- Gestion implicite des ressources (peut être vu comme un inconvénient) qui facilite la prise en main.

Pourquoi Python ?

- Langage de programmation (comme beaucoup d'autres).
- Portable et libre (non lié à un environnement particulier).
- Dynamique (interprété) ou compilé.
- Extensible : il s'interface avec d'autres langages ou logiciels.
- Orthogonal : un petit nombre de concepts engendrent des constructions très riches.
- Dynamiquement (mais fortement) typé.
- Gestion implicite des ressources (peut être vu comme un inconvénient) qui facilite la prise en main.
- Orienté objet (supporte l'héritage multiple et la surcharge des opérateurs).

Pourquoi Python ?

- Langage de programmation (comme beaucoup d'autres).
- Portable et libre (non lié à un environnement particulier).
- Dynamique (interprété) ou compilé.
- Extensible : il s'interface avec d'autres langages ou logiciels.
- Orthogonal : un petit nombre de concepts engendrent des constructions très riches.
- Dynamiquement (mais fortement) typé.
- Gestion implicite des ressources (peut être vu comme un inconvénient) qui facilite la prise en main.
- Orienté objet (supporte l'héritage multiple et la surcharge des opérateurs).
- Réflectif et introspectif.

Pourquoi Python ?

- Langage de programmation (comme beaucoup d'autres).
- Portable et libre (non lié à un environnement particulier).
- Dynamique (interprété) ou compilé.
- Extensible : il s'interface avec d'autres langages ou logiciels.
- Orthogonal : un petit nombre de concepts engendrent des constructions très riches.
- Dynamiquement (mais fortement) typé.
- Gestion implicite des ressources (peut être vu comme un inconvénient) qui facilite la prise en main.
- Orienté objet (supporte l'héritage multiple et la surcharge des opérateurs).
- Réflectif et introspectif.
- Largement contribué : les bibliothèques standard permettent d'accéder à un grand nombre de services (regexp, fichiers, signaux, web, géolocalisation, ihm, sgbd...) et les bibliothèques utilisateur sont facilement distribuables et installables.

Pourquoi Python ?

- Langage de programmation (comme beaucoup d'autres).
- Portable et libre (non lié à un environnement particulier).
- Dynamique (interprété) ou compilé.
- Extensible : il s'interface avec d'autres langages ou logiciels.
- Orthogonal : un petit nombre de concepts engendrent des constructions très riches.
- Dynamiquement (mais fortement) typé.
- Gestion implicite des ressources (peut être vu comme un inconvénient) qui facilite la prise en main.
- Orienté objet (supporte l'héritage multiple et la surcharge des opérateurs).
- Réflectif et introspectif.
- Largement contribué : les bibliothèques standard permettent d'accéder à un grand nombre de services (regexp, fichiers, signaux, web, géolocalisation, ihm, sgbd...) et les bibliothèques utilisateur sont facilement distribuables et installables.
- Optionnellement multithreadé.

Un python, des Pythons

Python peut être utilisé de deux façons différentes :

Un python, des Pythons

Python peut être utilisé de deux façons différentes :

- L'interpréteur en « ligne de commande » :

Un python, des Pythons

Python peut être utilisé de deux façons différentes :

- L'interpréteur en « ligne de commande » :
  - IDLE sous Windows ;

Un python, des Pythons

Python peut être utilisé de deux façons différentes :

- L'interpréteur en « ligne de commande » :
  - IDLE sous Windows ;
  - une fenêtre de terminal sous linux ou MacOS.

Un python, des Pythons

Python peut être utilisé de deux façons différentes :

- L'interpréteur en « ligne de commande » :
  - IDLE sous Windows ;
  - une fenêtre de terminal sous linux ou MacOS.
- L'exécution d'un fichier source :

Un python, des Pythons

Python peut être utilisé de deux façons différentes :

- L'interpréteur en « ligne de commande » :
  - IDLE sous Windows ;
  - une fenêtre de terminal sous linux ou MacOS.
- L'exécution d'un fichier source :
  - double clic sur le fichier ;

Un python, des Pythons

Python peut être utilisé de deux façons différentes :

- L'interpréteur en « ligne de commande » :
  - IDLE sous Windows ;
  - une fenêtre de terminal sous linux ou MacOS.
- L'exécution d'un fichier source :
  - double clic sur le fichier ;
  - exécution du fichier depuis la ligne de commande (`python nom_du_fichier.py`).

Un python, des Pythons

Python peut être utilisé de deux façons différentes :

- L'interpréteur en « ligne de commande » :
  - IDLE sous Windows ;
  - une fenêtre de terminal sous linux ou MacOS.
- L'exécution d'un fichier source :
  - double clic sur le fichier ;
  - exécution du fichier depuis la ligne de commande (`python nom_du_fichier.py`).

Python est rétrocompatible jusqu'à la version 3.

Dive into Python

Quelques ressources qui peuvent s'avérer utiles tout au long de l'apprentissage de Python :

- L'auto documentation accessible dans l'interpréteur via la commande `help()`.
- Le site officiel (en anglais) : `http://www.python.org/` regorge de documentation technique sur les bibliothèques standard et possède un tutoriel.
- Apprendre à programmer avec Python 3 : `http://inforef.be/swi/python.htm` est un cours sur l'utilisation de Python 3 allant des simples mathématiques aux concepts avancés de jeux en réseau, création de PDF ou encore gestion de base de donnée.
- Dive into Python 3 (en anglais) : `http://getpython3.com/diveintopython3/` plonge rapidement vers des concept avancés du langage.

Premiers pas

Anatomie du serpent

Un interpréteur à l'écoute

`>>>` Signifie que Python est prêt et vous invite à taper une commande.
Ce signe peut être remplacé par `...` lorsque Python comprend que votre commande attend une suite.

Calculons

```
>>> 1+2
3
>>> 1 + \
...     4
5
>>> exit()
```

Écrivons

```
>>> "toto"
'toto'
>>> 'bon\
... jour'
'bonjour'
>>> """
... Longue
... phrase
... """
'\nLongue\nphrase\n'
```

Anatomie du serpent

Python est une calculette (évoluée)

- Opérations mathématiques standard (`+`, `-`, `*`, `/`) avec respect des priorités et utilisation des parenthèses.
- Division entière (`//`), modulo (`%`) et puissance (`**`).
- Opérations bits à bits (`<<`, `>>`, `&`, `|`, `~`, `^`).

Anatomie du serpent

Python est une calculette (évoluée)

- Opérations mathématiques standard (`+`, `-`, `*`, `/`) avec respect des priorités et utilisation des parenthèses.
- Division entière (`//`), modulo (`%`) et puissance (`**`).
- Opérations bits à bits (`<<`, `>>`, `&`, `|`, `~`, `^`).

Python est une imprimerie

- Définition (`"`, `'`, `"""`) et concaténation (`+`) de chaînes de caractères.
- Formatage (`'%s, %s!' % ('Hello', 'world')`) avancé.
- Affichage paramétrable (`print(1, '+', 3, '=', 4, sep='', end=' ')`).
- Lecture d'informations au clavier(`input()`).
- Lecture et écriture de fichiers (`with open('toto.txt') as f: pass`).

Python comme langage de programmation

Brique de base : la variable

Python comme langage de programmation

Brique de base : la variable

- permet de stocker une information (valeur, structure, fichier, morceau de code...) ;

Python comme langage de programmation

Brique de base : la variable

- permet de stocker une information (valeur, structure, fichier, morceau de code. . .) ;
- doit être nommée et possède un type ;

Python comme langage de programmation

Brique de base : la variable

- permet de stocker une information (valeur, structure, fichier, morceau de code. . .) ;
- doit être nommée et possède un type ;
- le type n'est pas déclaré, il est connu à l'exécution et peut varier au cours du programme ;

Python comme langage de programmation

Brique de base : la variable

- permet de stocker une information (valeur, structure, fichier, morceau de code. . .) ;
- doit être nommée et possède un type ;
- le type n'est pas déclaré, il est connu à l'exécution et peut varier au cours du programme ;
- le nom de variable contient uniquement des chiffres, des lettres et des underscores et doit commencer par une lettre ;

Python comme langage de programmation

Brique de base : la variable

- permet de stocker une information (valeur, structure, fichier, morceau de code. . .) ;
- doit être nommée et possède un type ;
- le type n'est pas déclaré, il est connu à l'exécution et peut varier au cours du programme ;
- le nom de variable contient uniquement des chiffres, des lettres et des underscores et doit commencer par une lettre ;
- la casse est importante (python≠Python≠PYTHON).

Python comme langage de programmation

Brique de base : la variable

- permet de stocker une information (valeur, structure, fichier, morceau de code. . .) ;
- doit être nommée et possède un type ;
- le type n'est pas déclaré, il est connu à l'exécution et peut varier au cours du programme ;
- le nom de variable contient uniquement des chiffres, des lettres et des underscores et doit commencer par une lettre ;
- la casse est importante (python≠Python≠PYTHON).

Mots réservés

Les 33 mots suivant font partie du langage et ne peuvent pas être utilisés comme nom de variable :

and	as	assert	break	class	continue	def	del	elif
else	except	False	finally	for	from	global	if	import
in	is	lambda	None	nonlocal	not	or	pass	raise
return	True	try	while	with	yield			

Le reste (comme les noms de fonctions) est autorisé, soyez attentifs.

Python comme langage de programmation

Types de base

- `bool` : `True`, `False`.
- `int` : 0, 1, -4, 12344932321341233 1. . .
- `float` : 3., -2.423912, 123432.3204, 1.24E-6. . .
- `str` : 'toto', ' ', '\nSalut !\n'. . .
- `NoneType` : `None`.
- `bytes` : pour des données encodées (chaînes, images. . .).

Python comme langage de programmation

Types de base

- `bool` : `True`, `False`.
- `int` : 0, 1, -4, 12344932321341233 1. . .
- `float` : 3., -2.423912, 123432.3204, 1.24E-6. . .
- `str` : 'toto', ' ', '\nSalut !\n'. . .
- `NoneType` : `None`.
- `bytes` : pour des données encodées (chaînes, images. . .).

Type d'une variable ?

```
>>> type(3)
<class 'int'>
>>> type('coucou')
<class 'str'>
>>> type((4 + 5)/2 <= 2)
<class 'bool'>
```

Conversions de types

```
>>> int(3.14)
3
>>> str((4 + 5)/2 <= 2)
'False'
>>> float("3")
3.0
```

Python comme langage de programmation

Types évolués : les conteneurs

Python comme langage de programmation

Types évolués : les conteneurs

- Les n-uplets (`tuple`) : tableaux d'éléments de taille fixe. Chaque élément est accessible par un indice (entier). `()`, `("blanc", 8, 3.14)`...

Python comme langage de programmation

Types évolués : les conteneurs

- Les n-uplets (`tuple`) : tableaux d'éléments de taille fixe. Chaque élément est accessible par un indice (entier). `()`, `("blanc", 8, 3.14)`...
- Les listes (`list`) : n-uplet de taille variable. `[]`, `[3, 4, 'toto']`...

Python comme langage de programmation

Types évolués : les conteneurs

- Les n-uplets (`tuple`) : tableaux d'éléments de taille fixe. Chaque élément est accessible par un indice (entier). `()`, `("blanc", 8, 3.14)`...
- Les listes (`list`) : n-uplet de taille variable. `[]`, `[3, 4, 'toto']`...
- Les dictionnaires (`dict`) : liste où les éléments sont accessibles par une clef. `{}`, `{"nom": "Jean", "age": 28}`...

Python comme langage de programmation

Types évolués : les conteneurs

- Les n-uplets (`tuple`) : tableaux d'éléments de taille fixe. Chaque élément est accessible par un indice (entier). `()`, `("blanc", 8, 3.14)`...
- Les listes (`list`) : n-uplet de taille variable. `[]`, `[3, 4, 'toto']`...
- Les dictionnaires (`dict`) : liste où les éléments sont accessibles par une clef. `{}`, `{"nom": "Jean", "age": 28}`...
- Les ensembles (`set`) : liste ne contenant pas de doublons. Les éléments ne sont pas directement accessibles. `set()`, `{1, 3, 8}`...

Python comme langage de programmation

Types évolués : les conteneurs

- Les n-uplets (`tuple`) : tableaux d'éléments de taille fixe. Chaque élément est accessible par un indice (entier). `()`, `("blanc", 8, 3.14)`...
- Les listes (`list`) : n-uplet de taille variable. `[]`, `[3, 4, 'toto']`...
- Les dictionnaires (`dict`) : liste où les éléments sont accessibles par une clef. `{}`, `{"nom": "Jean", "age": 28}`...
- Les ensembles (`set`) : liste ne contenant pas de doublons. Les éléments ne sont pas directement accessibles. `set()`, `{1, 3, 8}`...
- Les `str` peuvent être vus comme un `tuple`.

Python comme langage de programmation

Types évolués : les conteneurs

- Les n-uplets (`tuple`) : tableaux d'éléments de taille fixe. Chaque élément est accessible par un indice (entier). `()`, `("blanc", 8, 3.14)`...
- Les listes (`list`) : n-uplet de taille variable. `[]`, `[3, 4, 'toto']`...
- Les dictionnaires (`dict`) : liste où les éléments sont accessibles par une clef. `{}`, `{"nom": "Jean", "age": 28}`...
- Les ensembles (`set`) : liste ne contenant pas de doublons. Les éléments ne sont pas directement accessibles. `set()`, `{1, 3, 8}`...
- Les `str` peuvent être vus comme un `tuple`.

Construire ses types : la programmation orientée objets

- Définition de classes à l'aide du mot réservé `class`.
- Constructions pouvant contenir tout type de données.
- En python, tout est objet.

Python comme langage de programmation

Manipuler les conteneurs

Python comme langage de programmation

Manipuler les conteneurs

- Accès à un élément avec `[i]` où `i` est un entier (ou une clef).

Python comme langage de programmation

Manipuler les conteneurs

- Accès à un élément avec `[i]` où `i` est un entier (ou une clef).
- Le dernier élément peut être accédé à l'indice -1.

Python comme langage de programmation

Manipuler les conteneurs

- Accès à un élément avec `[i]` où `i` est un entier (ou une clef).
- Le dernier élément peut être accédé à l'indice -1.
- Extraction des éléments de l'indice `b` (inclus) à l'indice `e` (exclus) avec `[b:e]` (ne fonctionne pas pour les `dict`).

Python comme langage de programmation

Manipuler les conteneurs

- Accès à un élément avec `[i]` où `i` est un entier (ou une clef).
- Le dernier élément peut être accédé à l'indice -1.
- Extraction des éléments de l'indice `b` (inclus) à l'indice `e` (exclus) avec `[b:e]` (ne fonctionne pas pour les `dict`).
- Si une borne est omise, le sous ensemble commence au premier élément et/ou s'arrête au dernier.

Python comme langage de programmation

Manipuler les conteneurs

- Accès à un élément avec `[i]` où `i` est un entier (ou une clef).
- Le dernier élément peut être accédé à l'indice -1.
- Extraction des éléments de l'indice `b` (inclus) à l'indice `e` (exclus) avec `[b:e]` (ne fonctionne pas pour les `dict`).
- Si une borne est omise, le sous ensemble commence au premier élément et/ou s'arrête au dernier.
- Ajouter un élément à un `set` : `s.add()`.

Python comme langage de programmation

Manipuler les conteneurs

- Accès à un élément avec `[i]` où `i` est un entier (ou une clef).
- Le dernier élément peut être accédé à l'indice -1.
- Extraction des éléments de l'indice `b` (inclus) à l'indice `e` (exclus) avec `[b:e]` (ne fonctionne pas pour les `dict`).
- Si une borne est omise, le sous ensemble commence au premier élément et/ou s'arrête au dernier.
- Ajouter un élément à un `set` : `s.add()`.
- Ajouter un élément en fin de `list` : `l.append()`

Python comme langage de programmation

Manipuler les conteneurs

- Accès à un élément avec `[i]` où `i` est un entier (ou une clef).
- Le dernier élément peut être accédé à l'indice -1.
- Extraction des éléments de l'indice `b` (inclus) à l'indice `e` (exclus) avec `[b:e]` (ne fonctionne pas pour les `dict`).
- Si une borne est omise, le sous ensemble commence au premier élément et/ou s'arrête au dernier.
- Ajouter un élément à un `set` : `s.add()`.
- Ajouter un élément en fin de `list` : `l.append()`
- Ajouter un élément à un `dict` : affectation directe.

Python comme langage de programmation

Manipuler les conteneurs

- Accès à un élément avec `[i]` où `i` est un entier (ou une clef).
- Le dernier élément peut être accédé à l'indice -1.
- Extraction des éléments de l'indice `b` (inclus) à l'indice `e` (exclus) avec `[b:e]` (ne fonctionne pas pour les `dict`).
- Si une borne est omise, le sous ensemble commence au premier élément et/ou s'arrête au dernier.
- Ajouter un élément à un `set` : `s.add()`.
- Ajouter un élément en fin de `list` : `l.append()`
- Ajouter un élément à un `dict` : affectation directe.
- `del` supprime un élément.

Python comme langage de programmation

Manipuler les `str`

Python comme langage de programmation

Manipuler les `str`

- `str` peut être vu comme un tuple pour l'accès aux éléments.

Python comme langage de programmation

Manipuler les `str`

- `str` peut être vu comme un tuple pour l'accès aux éléments.
- Deux chaînes de caractères peuvent être concaténées à l'aide de `+`.

Python comme langage de programmation

Manipuler les `str`

- `str` peut être vu comme un tuple pour l'accès aux éléments.
- Deux chaînes de caractères peuvent être concaténées à l'aide de `+`.
- Une chaîne de caractère peut être concaténée à elle même plusieurs fois à l'aide de `*`.

Python comme langage de programmation

Manipuler les `str`

- `str` peut être vu comme un tuple pour l'accès aux éléments.
- Deux chaînes de caractères peuvent être concaténées à l'aide de `+`.
- Une chaîne de caractère peut être concaténée à elle même plusieurs fois à l'aide de `*`.
- On peut contruire une nouvelle chaîne à partir de `s` en changeant la casse de certaines lettres (`s.upper()`, `s.lower()`, `s.capitalize()`, `s.title()`).

Python comme langage de programmation

Manipuler les `str`

- `str` peut être vu comme un tuple pour l'accès aux éléments.
- Deux chaînes de caractères peuvent être concaténées à l'aide de `+`.
- Une chaîne de caractère peut être concaténée à elle même plusieurs fois à l'aide de `*`.
- On peut contruire une nouvelle chaîne à partir de `s` en changeant la casse de certaines lettres (`s.upper()`, `s.lower()`, `s.capitalize()`, `s.title()`).
- Différents tests sont possibles (`s.startwith()`, `s.endswith()`, `s.isalpha()`, `s.isdecimal()`).

Python comme langage de programmation

Manipuler les `str`

- `str` peut être vu comme un tuple pour l'accès aux éléments.
- Deux chaînes de caractères peuvent être concaténées à l'aide de `+`.
- Une chaîne de caractère peut être concaténée à elle même plusieurs fois à l'aide de `*`.
- On peut contruire une nouvelle chaîne à partir de `s` en changeant la casse de certaines lettres (`s.upper()`, `s.lower()`, `s.capitalize()`, `s.title()`).
- Différents tests sont possibles (`s.startwith()`, `s.endswith()`, `s.isalpha()`, `s.isdecimal()`).
- La séparation de mots est possible (`s.partition()`, `s.split()`, `s.splitlines()`).

Programmer en Python

L'art de charmer les serpents

Une instruction, ~~des~~ un bloc d'instructions

L'art de charmer les serpents

Une instruction, ~~des~~ un bloc d'instructions

- Le symbole #, s'il n'est pas dans une chaîne de caractère, entame un commentaire jusqu'à la fin de la ligne courante.

L'art de charmer les serpents

Une instruction, ~~des~~ un bloc d'instructions

- Le symbole #, s'il n'est pas dans une chaîne de caractère, entame un commentaire jusqu'à la fin de la ligne courante.
- Le symbole pour délimiter la fin d'une instruction est le symbole de fin de ligne.

L'art de charmer les serpents

Une instruction, ~~des~~ un bloc d'instructions

- Le symbole #, s'il n'est pas dans une chaîne de caractère, entame un commentaire jusqu'à la fin de la ligne courante.
- Le symbole pour délimiter la fin d'une instruction est le symbole de fin de ligne.
- Une instruction peut tenir sur plusieurs lignes à l'aide du symbole \ immédiatement suivi d'une fin de ligne.

L'art de charmer les serpents

Une instruction, ~~des~~ un bloc d'instructions

- Le symbole `#`, s'il n'est pas dans une chaîne de caractère, entame un commentaire jusqu'à la fin de la ligne courante.
- Le symbole pour délimiter la fin d'une instruction est le symbole de fin de ligne.
- Une instruction peut tenir sur plusieurs lignes à l'aide du symbole `\` immédiatement suivi d'une fin de ligne.
- Le symbole pour démarrer un bloc d'instructions est le double point `:`.

L'art de charmer les serpents

Une instruction, ~~des~~ un bloc d'instructions

- Le symbole `#`, s'il n'est pas dans une chaîne de caractère, entame un commentaire jusqu'à la fin de la ligne courante.
- Le symbole pour délimiter la fin d'une instruction est le symbole de fin de ligne.
- Une instruction peut tenir sur plusieurs lignes à l'aide du symbole `\` immédiatement suivi d'une fin de ligne.
- Le symbole pour démarrer un bloc d'instructions est le double point `:`.
- Un bloc d'instructions DOIT se démarquer par son indentation (espaces supplémentaires en début de ligne).

L'art de charmer les serpents

Une instruction, ~~des~~ un bloc d'instructions

- Le symbole `#`, s'il n'est pas dans une chaîne de caractère, entame un commentaire jusqu'à la fin de la ligne courante.
- Le symbole pour délimiter la fin d'une instruction est le symbole de fin de ligne.
- Une instruction peut tenir sur plusieurs lignes à l'aide du symbole `\` immédiatement suivi d'une fin de ligne.
- Le symbole pour démarrer un bloc d'instructions est le double point `:`.
- Un bloc d'instructions DOIT se démarquer par son indentation (espaces supplémentaires en début de ligne).
- La fin d'un bloc d'instruction est visible par le retour à une indentation antérieure.

L'art de charmer les serpents

Approfondissement des blocs d'instructions

```
# -*- coding: utf-8 -*-

class Demo():
  abs = 3
  def __init__(self, y):
    self.ord = y
    print('objet initialisé')
  def sayCoords(self):
    print(self.abs, self.ord)
  def isOrig(self):
    return not (self.abs or self.ord)

def main():
  d = Demo(0)
  print(d.isOrig())
  d.sayCoords()
  d.abs = 0
  print(d.isOrig())

if __name__ == '__main__':
  main()
```

L'art de charmer les serpents

Approfondissement des blocs d'instructions

```
# -*- coding: utf-8 -*-

class Demo():
  abs = 3
  def __init__(self, y):
    self.ord = y
    print('objet initialisé')
  def sayCoords(self):
    print(self.abs, self.ord)
  def isOrig(self):
    return not (self.abs or self.ord)

def main():
  d = Demo(0)
  print(d.isOrig())
  d.sayCoords()
  d.abs = 0
  print(d.isOrig())

if __name__ == '__main__':
  main()
```

Ce code est « complexe »

Ne cherchez pas à comprendre en détail ce que fait chaque ligne de ce code. Il s'agit juste d'une illustration sur l'utilisation de l'indentation.

L'art de charmer les serpents

Fichiers source

- Texte pur, un simple éditeur de texte suffit (notepad, vim, geany...).
- Extension .py.
- Pour exécuter le fichier comme un programme depuis la ligne de commande, rajouter `#!/usr/bin/env python` en première ligne.
- Pour l'utilisation de caractères accentués, indiquer à python l'encodage du fichier en plaçant `# -*- coding: latin-1 -*-` (ou approprié) en seconde ligne. Si rien n'est indiqué, python suppose de l'UTF-8.

Exécution d'un programme

- Double clic sur le fichier (ouvrir avec python).
- `python nom_du_fichier.py` en ligne de commande.
- `./nom_du_fichier.py` en ligne de commande si le fichier est exécutable et que la première ligne est bien configurée.
- Dans l'interpréteur : `import nom_du_fichier`.

Branchements

Instructions conditionnelles : `if`

Branchements

Instructions conditionnelles : `if`

- Un bloc d'instruction est exécuté uniquement si une condition est remplie.

Branchements

Instructions conditionnelles : `if`

- Un bloc d'instruction est exécuté uniquement si une condition est remplie.
- L'instruction `else`, juste après un bloc `if`, détermine un bloc d'instruction exécutée uniquement si la condition n'est pas remplie.

Branchements

Instructions conditionnelles : `if`

- Un bloc d'instruction est exécuté uniquement si une condition est remplie.
- L'instruction `else`, juste après un bloc `if`, détermine un bloc d'instruction exécutée uniquement si la condition n'est pas remplie.
- Des instructions `elif`, avec de nouvelles conditions, peuvent être intercalées entre le « bloc if » et le « bloc else ».

Branchements

Instructions conditionnelles : `if`

- Un bloc d'instruction est exécuté uniquement si une condition est remplie.
- L'instruction `else`, juste après un bloc `if`, détermine un bloc d'instruction exécutée uniquement si la condition n'est pas remplie.
- Des instructions `elif`, avec de nouvelles conditions, peuvent être intercalées entre le « bloc if » et le « bloc else ».

Exemple

```
>>> if n > 99:
...     print("Dépasse la centaine")
... elif n == 42:
...     print("42 FTW!")
... else:
...     print("En dessous de cent")
...
```

Exercice

Écrire un programme qui demande un chiffre à l'utilisateur et affiche s'il est pair ou impair.

Conditions

Évaluer une expression conditionnelle

Conditions

Évaluer une expression conditionnelle

- Une condition est une expression qui, évaluée, vaut soit `True` soit `False`.

Conditions

Évaluer une expression conditionnelle

- Une condition est une expression qui, évaluée, vaut soit `True` soit `False`.
- Une expression est une condition.

Conditions

Évaluer une expression conditionnelle

- Une condition est une expression qui, évaluée, vaut soit `True` soit `False`.
- Une expression est une condition.
  - ou plutôt, elle se réduit à une condition par application implicite de `bool()`.

Conditions

Évaluer une expression conditionnelle

- Une condition est une expression qui, évaluée, vaut soit `True` soit `False`.
- Une expression est une condition.
  - ou plutôt, elle se réduit à une condition par application implicite de `bool()`.
- Comparer deux expressions (`==`, `!=`) forme une condition.

Conditions

Évaluer une expression conditionnelle

- Une condition est une expression qui, évaluée, vaut soit `True` soit `False`.
- Une expression est une condition.
  - ou plutôt, elle se réduit à une condition par application implicite de `bool()`.
- Comparer deux expressions (`==`, `!=`) forme une condition.
- Deux expressions de types comparables peuvent être comparées selon leur ordre (`<`, `<=`, `>`, `>=`).

Conditions

Évaluer une expression conditionnelle

- Une condition est une expression qui, évaluée, vaut soit `True` soit `False`.
- Une expression est une condition.
  - ou plutôt, elle se réduit à une condition par application implicite de `bool()`.
- Comparer deux expressions (`==`, `!=`) forme une condition.
- Deux expressions de types comparables peuvent être comparées selon leur ordre (`<`, `<=`, `>`, `>=`).
- Les conditions peuvent êtres agencées de façon logique (`not`, `and`, `or`).

Conditions

Évaluer une expression conditionnelle

- Une condition est une expression qui, évaluée, vaut soit `True` soit `False`.
- Une expression est une condition.
  - ou plutôt, elle se réduit à une condition par application implicite de `bool()`.
- Comparer deux expressions (`==`, `!=`) forme une condition.
- Deux expressions de types comparables peuvent être comparées selon leur ordre (`<`, `<=`, `>`, `>=`).
- Les conditions peuvent êtres agencées de façon logique (`not`, `and`, `or`).

Astuce and or

Les opérateurs logiques `and` et `or` ne réduisent pas automatiquement les expressions en valeur booléennes.

Équivalence

```
>>> if condition:
...     resultat = v1
... else:
...     resultat = v2
>>> resultat = condition and v1 or v2
```

Boucles

Instructions répétitives : `while`

Boucles

Instructions répétitives : `while`

- Un bloc d'instructions est exécuté, en boucle, tant qu'une condition est remplie.

Boucles

Instructions répétitives : `while`

- Un bloc d'instructions est exécuté, en boucle, tant qu'une condition est remplie.
- Le bloc d'instructions doit modifier le contenu d'au moins une variable de la condition pour espérer que la boucle s'arrête un jour.

Boucles

Instructions répétitives : `while`

- Un bloc d'instructions est exécuté, en boucle, tant qu'une condition est remplie.
- Le bloc d'instructions doit modifier le contenu d'au moins une variable de la condition pour espérer que la boucle s'arrête un jour.
- On peut arrêter la boucle de façon explicite (même si la condition est toujours vraie) à l'aide de l'instruction `break`.

Boucles

Instructions répétitives : `while`

- Un bloc d'instructions est exécuté, en boucle, tant qu'une condition est remplie.
- Le bloc d'instructions doit modifier le contenu d'au moins une variable de la condition pour espérer que la boucle s'arrête un jour.
- On peut arrêter la boucle de façon explicite (même si la condition est toujours vraie) à l'aide de l'instruction `break`.

Exercice

Écrire un script qui demande des nombres positifs à l'utilisateur. Un nombre négatif met fin au programme et lui fait afficher le nombre de chiffres, les extrêmes et la moyenne.

Exemple

```
>>> a, b, c = 1, 1, 10
>>> while c:
...     a, b, c = b, a+b, c-1
...     print(a, end=' ')
...
1 2 3 5 8 13 21 34 55 89
```

Affectations

Assigner une valeur à une variable : =

Affectations

Assigner une valeur à une variable : =

- Un nom de variable reçoit comme contenu le résultat de l'évaluation d'une expression.

Affectations

Assigner une valeur à une variable : `=`

- Un nom de variable reçoit comme contenu le résultat de l'évaluation d'une expression.
- L'expression est toujours évaluée avant d'être affectée (`a = a + 1`).

Affectations

Assigner une valeur à une variable : =

- Un nom de variable reçoit comme contenu le résultat de l'évaluation d'une expression.
- L'expression est toujours évaluée avant d'être affectée (`a = a + 1`).
- L'expression peut être aussi complexe que souhaitée (et donc être un conteneur d'expressions).

Affectations

Assigner une valeur à une variable : =

- Un nom de variable reçoit comme contenu le résultat de l'évaluation d'une expression.
- L'expression est toujours évaluée avant d'être affectée (`a = a + 1`).
- L'expression peut être aussi complexe que souhaitée (et donc être un conteneur d'expressions).

Unpacking

- Un conteneur peut être dépaquetté, chaque élément étant affecté simultanément à différentes variables.
- Le nombre de variables doit être égal au nombre d'éléments contenus.
- On peut se passer de certaines variables à l'aide de la variable spéciale `_`.
- Un conteneur de taille inconnue peut dépaquetter les éléments « restant » sous forme de liste (`a, *reste = (1, 2, 3, 4)`).

Affectations

Exemple d'affectation

```
>>> a = 3
>>> b = print(a)
3
>>> print(b)
None
```

list unpacking

```
>>> l = [1, (2, 3, 4), 5, 6, 7]
>>> a, *r, b = l
>>> c, _, _ = r
>>> d, e, f = c
>>> print(a, b, c, d, e, f, r)
1 7 (2, 3, 4) 2 3 4 [(2, 3, 4), 5, 6]
```

set unpacking

```
>>> s = {1, 2, 3}
>>> s.add(8)
>>> s.add(5)
>>> a, b, c, d, e = s
>>> print(a, b, c, d, e)
8 1 2 3 5
```

tuple unpacking

```
>>> a, b = 1, (2, 3, 4)
>>> c, *r = b
>>> print(a, b, c, r)
1 (2, 3, 4) 2 [3, 4]
```

str unpacking

```
>>> a, b, c = 'thé'
>>> *d, = 'boue'
>>> *_, e = 'testons'
>>> print(a, b, c, d, e)
t h é ['b', 'o', 'u', 'e'] s
```

dict unpacking

```
>>> d = {'ville': "Toulouse"}
>>> d['age'] = 32
>>> d['sexe'] = 'M'
>>> a, b, c = d
>>> print(a, b, c)
age sexe ville
```

Boucles (2)

Parcourir les éléments d'un conteneur : `for...in`

Boucles (2)

Parcourir les éléments d'un conteneur : `for...in`

- Un bloc d'instructions est répété pour chaque valeur contenue dans un conteneur.

Boucles (2)

Parcourir les éléments d'un conteneur : `for...in`

- Un bloc d'instructions est répété pour chaque valeur contenue dans un conteneur.
- L'instruction `break` arrête la répétition avant la fin des valeurs.

Boucles (2)

Parcourir les éléments d'un conteneur : `for...in`

- Un bloc d'instructions est répété pour chaque valeur contenue dans un conteneur.
- L'instruction `break` arrête la répétition avant la fin des valeurs.
- L'unpacking peut être utilisé pour un conteneur contenant des conteneurs de taille fixe et connue.

Boucles (2)

Parcourir les éléments d'un conteneur : `for...in`

- Un bloc d'instructions est répété pour chaque valeur contenue dans un conteneur.
- L'instruction `break` arrête la répétition avant la fin des valeurs.
- L'unpacking peut être utilisé pour un conteneur contenant des conteneurs de taille fixe et connue.
- `range()` produit une liste de nombre croissants.

Boucles (2)

Parcourir les éléments d'un conteneur : `for...in`

- Un bloc d'instructions est répété pour chaque valeur contenue dans un conteneur.
- L'instruction `break` arrête la répétition avant la fin des valeurs.
- L'unpacking peut être utilisé pour un conteneur contenant des conteneurs de taille fixe et connue.
- `range()` produit une liste de nombre croissants.

Exemple

```
>>> a = 0
>>> for v,m in [(12, 'un'), (8, 'non'), (0, 'dit')]:
...   a = a + 1
...   print("le mot", a, "est", m, "et vaut", v)
...   if a > 1:
...     break
...
le mot 1 est un et vaut 12
le mot 2 est non et vaut 8
```

Mais aussi

Le mot-clé `in` est également disponible pour construire des expressions conditionnelles qui testent la présence d'un élément dans un conteneur.

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `print()` pour afficher la représentation d'un ou plusieurs éléments :

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `print()` pour afficher la représentation d'un ou plusieurs éléments :
  - `sep` pour définir l'expression à utiliser entre chaque terme (une espace par défaut) ;

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `print()` pour afficher la représentation d'un ou plusieurs éléments :
  - `sep` pour définir l'expression à utiliser entre chaque terme (une espace par défaut) ;
  - `end` pour définir l'expression à utiliser à la fin de l'affichage (une nouvelle ligne par défaut) ;

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `print()` pour afficher la représentation d'un ou plusieurs éléments :
  - `sep` pour définir l'expression à utiliser entre chaque terme (une espace par défaut) ;
  - `end` pour définir l'expression à utiliser à la fin de l'affichage (une nouvelle ligne par défaut) ;
  - `file` pour spécifier le fichier dans lequel écrire (l'écran par défaut).

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `print()` pour afficher la représentation d'un ou plusieurs éléments :
  - `sep` pour définir l'expression à utiliser entre chaque terme (une espace par défaut) ;
  - `end` pour définir l'expression à utiliser à la fin de l'affichage (une nouvelle ligne par défaut) ;
  - `file` pour spécifier le fichier dans lequel écrire (l'écran par défaut).
- `input()` pour attendre une entrée utilisateur :

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `print()` pour afficher la représentation d'un ou plusieurs éléments :
  - `sep` pour définir l'expression à utiliser entre chaque terme (une espace par défaut) ;
  - `end` pour définir l'expression à utiliser à la fin de l'affichage (une nouvelle ligne par défaut) ;
  - `file` pour spécifier le fichier dans lequel écrire (l'écran par défaut).
- `input()` pour attendre une entrée utilisateur :
  - si un paramètre est fournit, il est affiché sur la ligne avant le curseur.

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `print()` pour afficher la représentation d'un ou plusieurs éléments :
  - `sep` pour définir l'expression à utiliser entre chaque terme (une espace par défaut) ;
  - `end` pour définir l'expression à utiliser à la fin de l'affichage (une nouvelle ligne par défaut) ;
  - `file` pour spécifier le fichier dans lequel écrire (l'écran par défaut).
- `input()` pour attendre une entrée utilisateur :
  - si un paramètre est fournit, il est affiché sur la ligne avant le curseur.
- `len()` pour connaître la taille d'un conteneur.

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `print()` pour afficher la représentation d'un ou plusieurs éléments :
  - `sep` pour définir l'expression à utiliser entre chaque terme (une espace par défaut) ;
  - `end` pour définir l'expression à utiliser à la fin de l'affichage (une nouvelle ligne par défaut) ;
  - `file` pour spécifier le fichier dans lequel écrire (l'écran par défaut).
- `input()` pour attendre une entrée utilisateur :
  - si un paramètre est fournit, il est affiché sur la ligne avant le curseur.
- `len()` pour connaître la taille d'un conteneur.
- `sum()` pour additionner les nombres dans un conteneur :

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `print()` pour afficher la représentation d'un ou plusieurs éléments :
  - `sep` pour définir l'expression à utiliser entre chaque terme (une espace par défaut) ;
  - `end` pour définir l'expression à utiliser à la fin de l'affichage (une nouvelle ligne par défaut) ;
  - `file` pour spécifier le fichier dans lequel écrire (l'écran par défaut).
- `input()` pour attendre une entrée utilisateur :
  - si un paramètre est fournit, il est affiché sur la ligne avant le curseur.
- `len()` pour connaître la taille d'un conteneur.
- `sum()` pour additionner les nombres dans un conteneur :
  - le conteneur doit contenir uniquement des nombres ;

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `print()` pour afficher la représentation d'un ou plusieurs éléments :
  - `sep` pour définir l'expression à utiliser entre chaque terme (une espace par défaut) ;
  - `end` pour définir l'expression à utiliser à la fin de l'affichage (une nouvelle ligne par défaut) ;
  - `file` pour spécifier le fichier dans lequel écrire (l'écran par défaut).
- `input()` pour attendre une entrée utilisateur :
  - si un paramètre est fournit, il est affiché sur la ligne avant le curseur.
- `len()` pour connaître la taille d'un conteneur.
- `sum()` pour additionner les nombres dans un conteneur :
  - le conteneur doit contenir uniquement des nombres ;
  - `start` valeur de départ pour l'addition (0 par défaut).

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `print()` pour afficher la représentation d'un ou plusieurs éléments :
  - `sep` pour définir l'expression à utiliser entre chaque terme (une espace par défaut) ;
  - `end` pour définir l'expression à utiliser à la fin de l'affichage (une nouvelle ligne par défaut) ;
  - `file` pour spécifier le fichier dans lequel écrire (l'écran par défaut).
- `input()` pour attendre une entrée utilisateur :
  - si un paramètre est fournit, il est affiché sur la ligne avant le curseur.
- `len()` pour connaître la taille d'un conteneur.
- `sum()` pour additionner les nombres dans un conteneur :
  - le conteneur doit contenir uniquement des nombres ;
  - `start` valeur de départ pour l'addition (0 par défaut).
- `exit()` pour arrêter le programme à tout moment :

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `print()` pour afficher la représentation d'un ou plusieurs éléments :
  - `sep` pour définir l'expression à utiliser entre chaque terme (une espace par défaut) ;
  - `end` pour définir l'expression à utiliser à la fin de l'affichage (une nouvelle ligne par défaut) ;
  - `file` pour spécifier le fichier dans lequel écrire (l'écran par défaut).
- `input()` pour attendre une entrée utilisateur :
  - si un paramètre est fournit, il est affiché sur la ligne avant le curseur.
- `len()` pour connaître la taille d'un conteneur.
- `sum()` pour additionner les nombres dans un conteneur :
  - le conteneur doit contenir uniquement des nombres ;
  - `start` valeur de départ pour l'addition (0 par défaut).
- `exit()` pour arrêter le programme à tout moment :
  - si un paramètre est fournit, il est affiché à l'écran avant de quitter.

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `range()` pour construire une liste croissante d'entiers ne dépassant pas une certaine valeur :

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `range()` pour construire une liste croissante d'entiers ne dépassant pas une certaine valeur :
  - deux valeurs déterminent la borne de départ en plus de la borne d'arrivée (0 par défaut) ;

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `range()` pour construire une liste croissante d'entiers ne dépassant pas une certaine valeur :
  - deux valeurs déterminent la borne de départ en plus de la borne d'arrivée (0 par défaut) ;
  - une troisième valeur indique le pas d'incrémentation (1 par défaut).

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `range()` pour construire une liste croissante d'entiers ne dépassant pas une certaine valeur :
  - deux valeurs déterminent la borne de départ en plus de la borne d'arrivée (0 par défaut) ;
  - une troisième valeur indique le pas d'incrémentation (1 par défaut).
- `help()` pour avoir de l'aide sur un type de donnée ou une fonction.

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `range()` pour construire une liste croissante d'entiers ne dépassant pas une certaine valeur :
  - deux valeurs déterminent la borne de départ en plus de la borne d'arrivée (0 par défaut) ;
  - une troisième valeur indique le pas d'incrémentation (1 par défaut).
- `help()` pour avoir de l'aide sur un type de donnée ou une fonction.
- `eval()` pour interpréter une chaîne de caractères comme une instruction :

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `range()` pour construire une liste croissante d'entiers ne dépassant pas une certaine valeur :
  - deux valeurs déterminent la borne de départ en plus de la borne d'arrivée (0 par défaut) ;
  - une troisième valeur indique le pas d'incrémentation (1 par défaut).
- `help()` pour avoir de l'aide sur un type de donnée ou une fonction.
- `eval()` pour interpréter une chaîne de caractères comme une instruction :
  - très puissant, attention donc ;

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `range()` pour construire une liste croissante d'entiers ne dépassant pas une certaine valeur :
  - deux valeurs déterminent la borne de départ en plus de la borne d'arrivée (0 par défaut) ;
  - une troisième valeur indique le pas d'incrémentation (1 par défaut).
- `help()` pour avoir de l'aide sur un type de donnée ou une fonction.
- `eval()` pour interpréter une chaîne de caractères comme une instruction :
  - très puissant, attention donc ;
  - souvent utilisé pour interpréter une entrée utilisateur.

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `range()` pour construire une liste croissante d'entiers ne dépassant pas une certaine valeur :
  - deux valeurs déterminent la borne de départ en plus de la borne d'arrivée (0 par défaut) ;
  - une troisième valeur indique le pas d'incrémentation (1 par défaut).
- `help()` pour avoir de l'aide sur un type de donnée ou une fonction.
- `eval()` pour interpréter une chaîne de caractères comme une instruction :
  - très puissant, attention donc ;
  - souvent utilisé pour interpréter une entrée utilisateur.
- `dir()` pour lister les éléments (variables, fonctions, modules, classes) connues :

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `range()` pour construire une liste croissante d'entiers ne dépassant pas une certaine valeur :
  - deux valeurs déterminent la borne de départ en plus de la borne d'arrivée (0 par défaut) ;
  - une troisième valeur indique le pas d'incrémentation (1 par défaut).
- `help()` pour avoir de l'aide sur un type de donnée ou une fonction.
- `eval()` pour interpréter une chaîne de caractères comme une instruction :
  - très puissant, attention donc ;
  - souvent utilisé pour interpréter une entrée utilisateur.
- `dir()` pour lister les éléments (variables, fonctions, modules, classes) connues :
  - si un paramètre est fournit, liste ce qu'il contient.

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `range()` pour construire une liste croissante d'entiers ne dépassant pas une certaine valeur :
  - deux valeurs déterminent la borne de départ en plus de la borne d'arrivée (0 par défaut) ;
  - une troisième valeur indique le pas d'incrémentation (1 par défaut).
- `help()` pour avoir de l'aide sur un type de donnée ou une fonction.
- `eval()` pour interpréter une chaîne de caractères comme une instruction :
  - très puissant, attention donc ;
  - souvent utilisé pour interpréter une entrée utilisateur.
- `dir()` pour lister les éléments (variables, fonctions, modules, classes) connues :
  - si un paramètre est fournit, liste ce qu'il contient.
- `open()` pour ouvrir un fichier :

Utiliser des fonctions

Quelques fonctions prédéfinies à utiliser sans modération

- `range()` pour construire une liste croissante d'entiers ne dépassant pas une certaine valeur :
  - deux valeurs déterminent la borne de départ en plus de la borne d'arrivée (0 par défaut) ;
  - une troisième valeur indique le pas d'incrémentation (1 par défaut).
- `help()` pour avoir de l'aide sur un type de donnée ou une fonction.
- `eval()` pour interpréter une chaîne de caractères comme une instruction :
  - très puissant, attention donc ;
  - souvent utilisé pour interpréter une entrée utilisateur.
- `dir()` pour lister les éléments (variables, fonctions, modules, classes) connues :
  - si un paramètre est fournit, liste ce qu'il contient.
- `open()` pour ouvrir un fichier :
  - `mode` permet de spécifier les restrictions d'utilisation ('r', 'w', '+', 'a'...) du fichier ('r' par défaut).

Utiliser des fonctions

Accéder à de nouvelles fonctionnalités : `import`

Utiliser des fonctions

Accéder à de nouvelles fonctionnalités : `import`

- Pour des raisons de performances, toutes les fonctionnalitées du langage ne sont pas chargées à chaque exécution.

Utiliser des fonctions

Accéder à de nouvelles fonctionnalités : `import`

- Pour des raisons de performances, toutes les fonctionnalitées du langage ne sont pas chargées à chaque exécution.
- Des modules regroupent des fonctionnalités autour d'un thème commun.

Utiliser des fonctions

Accéder à de nouvelles fonctionnalités : `import`

- Pour des raisons de performances, toutes les fonctionnalitées du langage ne sont pas chargées à chaque exécution.
- Des modules regroupent des fonctionnalités autour d'un thème commun.
- Un module est chargé à l'aide de `import`.

Utiliser des fonctions

Accéder à de nouvelles fonctionnalités : `import`

- Pour des raisons de performances, toutes les fonctionnalitées du langage ne sont pas chargées à chaque exécution.
- Des modules regroupent des fonctionnalités autour d'un thème commun.
- Un module est chargé à l'aide de `import`.
- Les fonctions d'un module sont accessibles par `nom_du_module.nomDeLaFonction()`.

Utiliser des fonctions

Accéder à de nouvelles fonctionnalités : `import`

- Pour des raisons de performances, toutes les fonctionnalitées du langage ne sont pas chargées à chaque exécution.
- Des modules regroupent des fonctionnalités autour d'un thème commun.
- Un module est chargé à l'aide de `import`.
- Les fonctions d'un module sont accessibles par `nom_du_module.nomDeLaFonction()`.
- On peut ne charger que certaines fonctions d'un module avec `from...import`. Ces fonctions sont directement accessibles par leur nom.

Utiliser des fonctions

Accéder à de nouvelles fonctionnalités : `import`

- Pour des raisons de performances, toutes les fonctionnalitées du langage ne sont pas chargées à chaque exécution.
- Des modules regroupent des fonctionnalités autour d'un thème commun.
- Un module est chargé à l'aide de `import`.
- Les fonctions d'un module sont accessibles par `nom_du_module.nomDeLaFonction()`.
- On peut ne charger que certaines fonctions d'un module avec `from...import`. Ces fonctions sont directement accessibles par leur nom.
- Le joker `*` désigne l'ensemble des fonctions d'un module.

Utiliser des fonctions

Accéder à de nouvelles fonctionnalités : `import`

- Pour des raisons de performances, toutes les fonctionnalitées du langage ne sont pas chargées à chaque exécution.
- Des modules regroupent des fonctionnalités autour d'un thème commun.
- Un module est chargé à l'aide de `import`.
- Les fonctions d'un module sont accessibles par `nom_du_module.nomDeLaFonction()`.
- On peut ne charger que certaines fonctions d'un module avec `from...import`. Ces fonctions sont directement accessibles par leur nom.
- Le joker `*` désigne l'ensemble des fonctions d'un module.
- Pour éviter les conflits de noms (ou pour écrire moins) on peut renommer les fonctions ou les modules avec `as`.

Utiliser des fonctions

Importer et utiliser

```
>>> import math
>>> math.sin(math.pi)
1.2246467991473532e-16
>>> from math import *
>>> cos(pi)
-1.0
>>> from time import time
>>> time()
1357058524.6716356
>>> from random import randint as random
>>> random(3,10)
8
>>> from random import choice
>>> choice('un mot')
'o'
```

Utiliser des fonctions

Importer et utiliser

```
>>> import math
>>> math.sin(math.pi)
1.2246467991473532e-16
>>> from math import *
>>> cos(pi)
-1.0
>>> from time import time
>>> time()
1357058524.6716356
>>> from random import randint as random
>>> random(3,10)
8
>>> from random import choice
>>> choice('un mot')
'o'
```

Turtle

Le module **turtle** définit un environnement dans lequel une tortue dessine dans une fenêtre. On contrôle son déplacement avec **forward** et **backward**. Elle tourne (en degré) avec **left** et **right**. Elle arrête de dessiner après un **up** et reprend après un **down**. **goto** la positionne à un endroit particulier et **reset** efface tout.
Écrire un programme qui demande des formes à l'utilisateur et les dessine. Une forme qui n'a pas été prévue termine le programme.

Écrire des fonctions

Fonctions utilisateur : `def`

Écrire des fonctions

Fonctions utilisateur : `def`

- Un bloc d'instructions est nommé et utilisable en tant que fonction.

Écrire des fonctions

Fonctions utilisateur : `def`

- Un bloc d'instructions est nommé et utilisable en tant que fonction.
- L'instruction `return` arrête la fonction et peut éventuellement permettre de renvoyer une valeur.

Écrire des fonctions

Fonctions utilisateur : `def`

- Un bloc d'instructions est nommé et utilisable en tant que fonction.
- L'instruction `return` arrête la fonction et peut éventuellement permettre de renvoyer une valeur.
- La fin du bloc d'instructions de la fonction est équivalent à un `return`.

Écrire des fonctions

Fonctions utilisateur : `def`

- Un bloc d'instructions est nommé et utilisable en tant que fonction.
- L'instruction `return` arrête la fonction et peut éventuellement permettre de renvoyer une valeur.
- La fin du bloc d'instructions de la fonction est équivalent à un `return`.
- Une fonction possède une séquence de paramètres représenté par un `tuple` (éventuellement vide) de noms de variable.

Écrire des fonctions

Fonctions utilisateur : `def`

- Un bloc d'instructions est nommé et utilisable en tant que fonction.
- L'instruction `return` arrête la fonction et peut éventuellement permettre de renvoyer une valeur.
- La fin du bloc d'instructions de la fonction est équivalent à un `return`.
- Une fonction possède une séquence de paramètres représenté par un `tuple` (éventuellement vide) de noms de variable.
- Les paramètres peuvent posséder une valeur par défaut (`nom = valeur`).

Écrire des fonctions

Fonctions utilisateur : `def`

- Un bloc d'instructions est nommé et utilisable en tant que fonction.
- L'instruction `return` arrête la fonction et peut éventuellement permettre de renvoyer une valeur.
- La fin du bloc d'instructions de la fonction est équivalent à un `return`.
- Une fonction possède une séquence de paramètres représenté par un `tuple` (éventuellement vide) de noms de variable.
- Les paramètres peuvent posséder une valeur par défaut (`nom = valeur`).
- À l'appel d'une fonction, au moins autant de valeurs que le nombre de paramètres qui n'ont pas de valeur par défaut doivent être fournies.

Écrire des fonctions

Fonctions utilisateur : `def`

- Un bloc d'instructions est nommé et utilisable en tant que fonction.
- L'instruction `return` arrête la fonction et peut éventuellement permettre de renvoyer une valeur.
- La fin du bloc d'instructions de la fonction est équivalent à un `return`.
- Une fonction possède une séquence de paramètres représenté par un `tuple` (éventuellement vide) de noms de variable.
- Les paramètres peuvent posséder une valeur par défaut (`nom = valeur`).
- À l'appel d'une fonction, au moins autant de valeurs que le nombre de paramètres qui n'ont pas de valeur par défaut doivent être fournies.
- Une chaîne de caractères (`docstring`) peut être ajoutée en première ligne du bloc d'instructions pour servir d'aide visible à l'aide de `help()`.

Écrire des fonctions

Définition et utilisation de fonctions

```
>>> def print10Tables():
...   for i in range(10):
...     printTable(i+1)
...
>>> def printTable(base, start=1, end=10):
...   for i in range(start,end+1):
...     print(base, 'x', i, '=', base * i)
...
>>> print10Tables()
1 x 1 = 1
1 x 2 = 2
# coupé par manque de place
10 x 10 = 100
>>> printTable(13,11,end=12)
13 x 11 = 143
13 x 12 = 156
```

Écrire des fonctions

Définition et utilisation de fonctions

```
>>> def print10Tables():
...   for i in range(10):
...     printTable(i+1)
...
>>> def printTable(base, start=1, end=10):
...   for i in range(start,end+1):
...     print(base, 'x', i, '=', base * i)
...
>>> print10Tables()
1 x 1 = 1
1 x 2 = 2
# coupé par manque de place
10 x 10 = 100
>>> printTable(13,11,end=12)
13 x 11 = 143
13 x 12 = 156
```

Écriture alternative

Pour les fonctions courtes, une syntaxe semblable aux fonctions mathématiques existe : `lambda x: x**2 - 3*x + 4`.

Des variables à portée de main

Notion de portée

Des variables à portée de main

Notion de portée

- À l'entrée dans une fonction, les noms de variables sont réinitialisés et sont restaurés à la sortie de la fonction.

Des variables à portée de main

Notion de portée

- À l'entrée dans une fonction, les noms de variables sont réinitialisés et sont restaurés à la sortie de la fonction.
- Les paramètres sont les seules variables directement connues de la fonction.

Des variables à portée de main

Notion de portée

- À l'entrée dans une fonction, les noms de variables sont réinitialisés et sont restaurés à la sortie de la fonction.
- Les paramètres sont les seules variables directement connues de la fonction.
- Les variables déclarées à l'extérieur de la fonction peuvent être accédées grâce au mot-clef `global`.

Des variables à portée de main

Notion de portée

- À l'entrée dans une fonction, les noms de variables sont réinitialisés et sont restaurés à la sortie de la fonction.
- Les paramètres sont les seules variables directement connues de la fonction.
- Les variables déclarées à l'extérieur de la fonction peuvent être accédées grâce au mot-clef `global`.

Variable non déclarée

```
>>> GLOBAL_VAR = 0
>>> def func():
...     if GLOBAL_VAR:
...         print("Valeur :", GLOBAL_VAR)
...     GLOBAL_VAR += 1
...
>>> func()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in func
UnboundLocalError: local variable
'GLOBAL_VAR' referenced before assignment
```

Utilisation de `global`

```
>>> A = 0
>>> def add_a():
...     global A
...     A += 1
...     print(A)
...
>>> add_a()
1
```

Us et coutûmes

Écrire un programme de façon standard

Us et coutûmes

Écrire un programme de façon standard

- Python définit un certain nombre de variables de façon automatique ;

Us et coutûmes

Écrire un programme de façon standard

- Python définit un certain nombre de variables de façon automatique ;
- la variale `__name__` contient le nom du module importé ;

Us et coutûmes

Écrire un programme de façon standard

- Python définit un certain nombre de variables de façon automatique ;
- la variale `__name__` contient le nom du module importé ;
- le nom du programme principal est `'__main__'` ;

Us et coutûmes

Écrire un programme de façon standard

- Python définit un certain nombre de variables de façon automatique ;
- la variale `__name__` contient le nom du module importé ;
- le nom du programme principal est `'__main__'` ;
- dans un fichier, il est courant de mettre le code à exécuter dans une fonction (`main()`) et de n'exécuter que le test `if __name__ == '__main__': main()`.

Us et coutûmes

Écrire un programme de façon standard

- Python définit un certain nombre de variables de façon automatique ;
- la variale `__name__` contient le nom du module importé ;
- le nom du programme principal est `'__main__'` ;
- dans un fichier, il est courant de mettre le code à exécuter dans une fonction (`main()`) et de n'exécuter que le test `if __name__ == '__main__': main()`.
- Pour se passer de `input()`, les arguments de la ligne de commande sont stockés dans la variable `argv` du module `sys`.

Us et coutûmes

Écrire un programme de façon standard

- Python définit un certain nombre de variables de façon automatique ;
- la variale `__name__` contient le nom du module importé ;
- le nom du programme principal est `'__main__'` ;
- dans un fichier, il est courant de mettre le code à exécuter dans une fonction (`main()`) et de n'exécuter que le test `if __name__ == '__main__': main()`.
- Pour se passer de `input()`, les arguments de la ligne de commande sont stockés dans la variable `argv` du module `sys`.

Pourquoi ?

Lors d'un `import`, le code du fichier est exécuté une (et une seule) fois. Cette technique permet de définir une fonction (qui sera donc exécutable plusieurs fois) que l'on appelle quand on veut, mais surtout avec les arguments qu'on veut, tout en gardant une exécution possible par la ligne de commande.

Une histoire de contruction

List comprehensions

Une histoire de contruction

List comprehensions

- Une opération courante est de construire une liste (ou un dictionnaire) en y ajoutant les éléments un par un ;

Une histoire de contruction

List comprehensions

- Une opération courante est de construire une liste (ou un dictionnaire) en y ajoutant les éléments un par un ;
- ces constructions (peu esthétiques) ne sont pas forcément comprises à la première lecture ;

Une histoire de contruction

List comprehensions

- Une opération courante est de construire une liste (ou un dictionnaire) en y ajoutant les éléments un par un ;
- ces constructions (peu esthétiques) ne sont pas forcément comprises à la première lecture ;
- python propose les « list comprehensions » pour faciliter cette création ;

Une histoire de contruction

List comprehensions

- Une opération courante est de construire une liste (ou un dictionnaire) en y ajoutant les éléments un par un ;
- ces constructions (peu esthétiques) ne sont pas forcément comprises à la première lecture ;
- python propose les « list comprehensions » pour faciliter cette création ;
- la syntaxe est `[expression for variable in iterable]` ;

Une histoire de contruction

List comprehensions

- Une opération courante est de construire une liste (ou un dictionnaire) en y ajoutant les éléments un par un ;
- ces constructions (peu esthétiques) ne sont pas forcément comprises à la première lecture ;
- python propose les « list comprehensions » pour faciliter cette création ;
- la syntaxe est `[expression for variable in iterable]` ;
- les « dict comprehensions » suivent le même principe : `{expression: expression for variable in iterable}`.

Une histoire de contruction

List comprehensions

- Une opération courante est de construire une liste (ou un dictionnaire) en y ajoutant les éléments un par un ;
- ces constructions (peu esthétiques) ne sont pas forcément comprises à la première lecture ;
- python propose les « list comprehensions » pour faciliter cette création ;
- la syntaxe est `[expression for variable in iterable]` ;
- les « dict comprehensions » suivent le même principe : `{expression: expression for variable in iterable}`.

Attention aux « tuple comprehensions »

Même s'il est possible d'utiliser la sémantique des list comprehensions pour les tuples, le résultat n'est pas forcément celui qu'on attend.

De l'art de s'exprimer convenablement

Formatage des chaînes de caractères

De l'art de s'exprimer convenablement

Formatage des chaînes de caractères

- Quelques fois il semblera plus intéressant de construire une chaîne de caractères plutôt que de l'imprimer ;

De l'art de s'exprimer convenablement

Formatage des chaînes de caractères

- Quelques fois il semblera plus intéressant de construire une chaîne de caractères plutôt que de l'imprimer ;
- mais les possibilités de formatage de `print()` ou de l'écriture dans un fichier pourraient sembler plus attirantes ;

De l'art de s'exprimer convenablement

Formatage des chaînes de caractères

- Quelques fois il semblera plus intéressant de construire une chaîne de caractères plutôt que de l'imprimer ;
- mais les possibilités de formatage de `print()` ou de l'écriture dans un fichier pourraient sembler plus attirantes ;
- c'est sans compter sur l'opération `format()` des chaînes de caractères ;

De l'art de s'exprimer convenablement

Formatage des chaînes de caractères

- Quelques fois il semblera plus intéressant de construire une chaîne de caractères plutôt que de l'imprimer ;
- mais les possibilités de formatage de `print()` ou de l'écriture dans un fichier pourraient sembler plus attirantes ;
- c'est sans compter sur l'opération `format()` des chaînes de caractères ;
- un couple d'accolades à l'intérieur d'une chaîne peut définir de la place pour une variable à insérer plus tard ;

De l'art de s'exprimer convenablement

Formatage des chaînes de caractères

- Quelques fois il semblera plus intéressant de construire une chaîne de caractères plutôt que de l'imprimer ;
- mais les possibilités de formatage de `print()` ou de l'écriture dans un fichier pourraient sembler plus attirantes ;
- c'est sans compter sur l'opération `format()` des chaînes de caractères ;
- un couple d'accolades à l'intérieur d'une chaîne peut définir de la place pour une variable à insérer plus tard ;
- `'{1} {0}!'.format('world','Hello')` donnera `'Hello world!'` ;

De l'art de s'exprimer convenablement

Formatage des chaînes de caractères

- Quelques fois il semblera plus intéressant de construire une chaîne de caractères plutôt que de l'imprimer ;
- mais les possibilités de formatage de `print()` ou de l'écriture dans un fichier pourraient sembler plus attirantes ;
- c'est sans compter sur l'opération `format()` des chaînes de caractères ;
- un couple d'accolades à l'intérieur d'une chaîne peut définir de la place pour une variable à insérer plus tard ;
- `'{1} {0}!'.format('world','Hello')` donnera `'Hello world!'` ;
- `'pi vaut {:.5f}. {stunt}'.format(math.pi, stunt='Étonnant, non ?')` donnera `'pi vaut 3.14159. Étonnant, non ?'`.

De l'art de s'exprimer convenablement

Formatage des chaînes de caractères

- Quelques fois il semblera plus intéressant de construire une chaîne de caractères plutôt que de l'imprimer ;
- mais les possibilités de formatage de `print()` ou de l'écriture dans un fichier pourraient sembler plus attirantes ;
- c'est sans compter sur l'opération `format()` des chaînes de caractères ;
- un couple d'accolades à l'intérieur d'une chaîne peut définir de la place pour une variable à insérer plus tard ;
- `'{1} {0}!'.format('world','Hello')` donnera `'Hello world!'` ;
- `'pi vaut {:.5f}. {stunt}'.format(math.pi, stunt='Étonnant, non ?')` donnera `'pi vaut 3.14159. Étonnant, non ?'`.
- D'autres exemples sur `http://docs.python.org/3.3/library/string.html`.

POO en Python

De l'intérêt de regrouper les informations

Une variable par donnée

```
>>> cb1_owner = "Foo"
>>> cb1_code = 1234
>>> cb1_max_amount = 1000.
>>> cb2_owner = "Bar"
>>> cb2_code = 5678
>>> cb2_max_amount = 500.
```

De l'intérêt de regrouper les informations

Une variable par donnée

```
>>> cb1_owner = "Foo"
>>> cb1_code = 1234
>>> cb1_max_amount = 1000.
>>> cb2_owner = "Bar"
>>> cb2_code = 5678
>>> cb2_max_amount = 500.
```

Groupement des données

```
>>> cb1 = {
...     "owner": "Foo"
...     "code": 1234
...     "max_amount": 1000.
... }
>>> cb2 = {
...     "owner": "Bar"
...     "code": 5678
...     "max_amount": 500.
... }
```

De l'intérêt de regrouper les informations

Une variable par donnée

```
>>> cb1_owner = "Foo"
>>> cb1_code = 1234
>>> cb1_max_amount = 1000.
>>> cb2_owner = "Bar"
>>> cb2_code = 5678
>>> cb2_max_amount = 500.
```

Groupement des données

```
>>> cb1 = {
...     "owner": "Foo"
...     "code": 1234
...     "max_amount": 1000.
... }
>>> cb2 = {
...     "owner": "Bar"
...     "code": 5678
...     "max_amount": 500.
... }
```

Avantages

De l'intérêt de regrouper les informations

Une variable par donnée

```
>>> cb1_owner = "Foo"
>>> cb1_code = 1234
>>> cb1_max_amount = 1000.
>>> cb2_owner = "Bar"
>>> cb2_code = 5678
>>> cb2_max_amount = 500.
```

Groupement des données

```
>>> cb1 = {
...     "owner": "Foo"
...     "code": 1234
...     "max_amount": 1000.
... }
>>> cb2 = {
...     "owner": "Bar"
...     "code": 5678
...     "max_amount": 500.
... }
```

Avantages

- On ne se perd pas dans les noms de variables.

De l'intérêt de regrouper les informations

Une variable par donnée

```
>>> cb1_owner = "Foo"
>>> cb1_code = 1234
>>> cb1_max_amount = 1000.
>>> cb2_owner = "Bar"
>>> cb2_code = 5678
>>> cb2_max_amount = 500.
```

Groupement des données

```
>>> cb1 = {
...     "owner": "Foo"
...     "code": 1234
...     "max_amount": 1000.
... }
>>> cb2 = {
...     "owner": "Bar"
...     "code": 5678
...     "max_amount": 500.
... }
```

Avantages

- On ne se perd pas dans les noms de variables.
- Regroupement sémantique.

De l'intérêt de regrouper les informations

Une variable par donnée

```
>>> cb1_owner = "Foo"
>>> cb1_code = 1234
>>> cb1_max_amount = 1000.
>>> cb2_owner = "Bar"
>>> cb2_code = 5678
>>> cb2_max_amount = 500.
```

Groupement des données

```
>>> cb1 = {
...     "owner": "Foo"
...     "code": 1234
...     "max_amount": 1000.
... }
>>> cb2 = {
...     "owner": "Bar"
...     "code": 5678
...     "max_amount": 500.
... }
```

Avantages

- On ne se perd pas dans les noms de variables.
- Regroupement sémantique.
- Écriture de fonctions simplifiée.

De l'intérêt d'automatiser les traitements

Une légère erreur

```
>>> cb1 = {
...   "owner": "Foo"
...   "code": 1234
...   "max_amount": 1000.
... }
>>> cb2 = {
...   "name": "Bar"
...   "code": 5678
...   "max_amount": 500.
... }
```

De l'intérêt d'automatiser les traitements

Une légère erreur

```
>>> cb1 = {
...   "owner": "Foo"
...   "code": 1234
...   "max_amount": 1000.
... }
>>> cb2 = {
...   "name": "Bar"
...   "code": 5678
...   "max_amount": 500.
... }
```

Une classe

```
>>> class CB:
...   def __init__(self, o, code, max):
...     self.owner = owner
...     self.code = code
...     self.max_amount = max
...
>>> cb1 = CB("Foo", 1234, 1000.)
>>> cb2 = CB("Bar", 5678, 500.)
```

De l'intérêt d'automatiser les traitements

Une légère erreur

```
>>> cb1 = {
...   "owner": "Foo"
...   "code": 1234
...   "max_amount": 1000.
... }
>>> cb2 = {
...   "name": "Bar"
...   "code": 5678
...   "max_amount": 500.
... }
```

Une classe

```
>>> class CB:
...   def __init__(self, o, code, max):
...     self.owner = owner
...     self.code = code
...     self.max_amount = max
...
>>> cb1 = CB("Foo", 1234, 1000.)
>>> cb2 = CB("Bar", 5678, 500.)
```

Avantages

De l'intérêt d'automatiser les traitements

Une légère erreur

```
>>> cb1 = {
...   "owner": "Foo"
...   "code": 1234
...   "max_amount": 1000.
... }
>>> cb2 = {
...   "name": "Bar"
...   "code": 5678
...   "max_amount": 500.
... }
```

Une classe

```
>>> class CB:
...   def __init__(self, o, code, max):
...     self.owner = owner
...     self.code = code
...     self.max_amount = max
...
>>> cb1 = CB("Foo", 1234, 1000.)
>>> cb2 = CB("Bar", 5678, 500.)
```

Avantages

- Moins de ré-écriture de codes, donc moins d'erreurs potentielles.

De l'intérêt d'automatiser les traitements

Une légère erreur

```
>>> cb1 = {
...   "owner": "Foo"
...   "code": 1234
...   "max_amount": 1000.
... }
>>> cb2 = {
...   "name": "Bar"
...   "code": 5678
...   "max_amount": 500.
... }
```

Une classe

```
>>> class CB:
...   def __init__(self, o, code, max):
...     self.owner = owner
...     self.code = code
...     self.max_amount = max
...
>>> cb1 = CB("Foo", 1234, 1000.)
>>> cb2 = CB("Bar", 5678, 500.)
```

Avantages

- Moins de ré-écriture de codes, donc moins d'erreurs potentielles.
- Sémantique forte vis-à-vis des concepts.

De l'intérêt d'automatiser les traitements

Une légère erreur

```
>>> cb1 = {
...   "owner": "Foo"
...   "code": 1234
...   "max_amount": 1000.
... }
>>> cb2 = {
...   "name": "Bar"
...   "code": 5678
...   "max_amount": 500.
... }
```

Une classe

```
>>> class CB:
...   def __init__(self, o, code, max):
...     self.owner = owner
...     self.code = code
...     self.max_amount = max
...
>>> cb1 = CB("Foo", 1234, 1000.)
>>> cb2 = CB("Bar", 5678, 500.)
```

Avantages

- Moins de ré-écriture de codes, donc moins d'erreurs potentielles.
- Sémantique forte vis-à-vis des concepts.
- Automatisation possible des traitements (initialisation, méthodes. . .).

Définir une classe

Regrouper des données et les opérations pour les manipuler : `class`

Définir une classe

Regrouper des données et les opérations pour les manipuler : `class`

- Une classe est un moule pour fabriquer des objets.

Définir une classe

Regrouper des données et les opérations pour les manipuler : `class`

- Une classe est un moule pour fabriquer des objets.
- Un objet est un ensemble de données manipulables via des opérations spécifiques.

Définir une classe

Regrouper des données et les opérations pour les manipuler : `class`

- Une classe est un moule pour fabriquer des objets.
- Un objet est un ensemble de données manipulables via des opérations spécifiques.
- Une classe définit des attributs (variables) et des méthodes (fonctions).

Définir une classe

Regrouper des données et les opérations pour les manipuler : `class`

- Une classe est un moule pour fabriquer des objets.
- Un objet est un ensemble de données manipulables via des opérations spécifiques.
- Une classe définit des attributs (variables) et des méthodes (fonctions).
- À l'appel d'une méthode, l'objet qui appelle cette méthode est passé comme premier argument de la fonction.

Définir une classe

Regrouper des données et les opérations pour les manipuler : `class`

- Une classe est un moule pour fabriquer des objets.
- Un objet est un ensemble de données manipulables via des opérations spécifiques.
- Une classe définit des attributs (variables) et des méthodes (fonctions).
- À l'appel d'une méthode, l'objet qui appelle cette méthode est passé comme premier argument de la fonction.
- Pour créer un objet, on utilise le nom de la classe comme une fonction.

Définir une classe

Regrouper des données et les opérations pour les manipuler : `class`

- Une classe est un moule pour fabriquer des objets.
- Un objet est un ensemble de données manipulables via des opérations spécifiques.
- Une classe définit des attributs (variables) et des méthodes (fonctions).
- À l'appel d'une méthode, l'objet qui appelle cette méthode est passé comme premier argument de la fonction.
- Pour créer un objet, on utilise le nom de la classe comme une fonction.
- Des paramètres peuvent être fournis lors de la création de l'objet et sont traités par la méthode spéciale `__init__`.

Définir une classe

Regrouper des données et les opérations pour les manipuler : `class`

- Une classe est un moule pour fabriquer des objets.
- Un objet est un ensemble de données manipulables via des opérations spécifiques.
- Une classe définit des attributs (variables) et des méthodes (fonctions).
- À l'appel d'une méthode, l'objet qui appelle cette méthode est passé comme premier argument de la fonction.
- Pour créer un objet, on utilise le nom de la classe comme une fonction.
- Des paramètres peuvent être fournis lors de la création de l'objet et sont traités par la méthode spéciale `__init__`.
- L'accès à un attribut ou une méthode d'un objet se fait à l'aide du point `.` .

Définir une classe

Rôle et utilisation de l'objet (`self`)

Définir une classe

Rôle et utilisation de l'objet (`self`)

- Un objet contient des données et des informations pour les manipuler.

Définir une classe

Rôle et utilisation de l'objet (`self`)

- Un objet contient des données et des informations pour les manipuler.
- Pour chaque méthode, l'objet qui exécute la méthode est appelé `self`.

Définir une classe

Rôle et utilisation de l'objet (`self`)

- Un objet contient des données et des informations pour les manipuler.
- Pour chaque méthode, l'objet qui exécute la méthode est appelé `self`.
- Cet objet est le premier paramètre passé à la méthode lors de son exécution et doit donc être déclaré en tant que tel.

Définir une classe

Rôle et utilisation de l'objet (`self`)

- Un objet contient des données et des informations pour les manipuler.
- Pour chaque méthode, l'objet qui exécute la méthode est appelé `self`.
- Cet objet est le premier paramètre passé à la méthode lors de son exécution et doit donc être déclaré en tant que tel.
- L'appel à une méthode est de la forme : `nom_objet.nom_méthode(paramètres_autres_que_self)`.

Définir une classe

Rôle et utilisation de l'objet (`self`)

- Un objet contient des données et des informations pour les manipuler.
- Pour chaque méthode, l'objet qui exécute la méthode est appelé **`self`**.
- Cet objet est le premier paramètre passé à la méthode lors de son exécution et doit donc être déclaré en tant que tel.
- L'appel à une méthode est de la forme : `nom_objet.nom_méthode(paramètres_autres_que_self)`.

Exemple 1

```
>>> class Ex1:
...     def __init__(self):
...         self.var = "Quelque chose"
...     def affiche(self, message = None):
...         print(message if message is not
None else self.var)
...
>>> test = Ex1()
>>> test.affiche()
Quelque chose
```

Exemple 2

```
>>> class Ex2:
...     message = "Bonjour"
...     def affiche(self, nom):
...         print(self.message, nom)
...
>>> test = Ex2()
>>> test.affiche("les gens !")
Bonjour les gens !
```

N'oubliez pas la documentation

Utilisation des docstrings et de `help`

N'oubliez pas la documentation

Utilisation des docstrings et de `help`

- La commande `help` documente la classe, ses méthodes et les attributs « de classe ».

N'oubliez pas la documentation

Utilisation des docstrings et de `help`

- La commande `help` documente la classe, ses méthodes et les attributs « de classe ».
- Comme pour une fonction classique, une docstring peut être ajoutée lors de la définition d'une méthode pour garnir cette documentation.

N'oubliez pas la documentation

Utilisation des docstrings et de `help`

- La commande `help` documente la classe, ses méthodes et les attributs « de classe ».
- Comme pour une fonction classique, une docstring peut être ajoutée lors de la définition d'une méthode pour garnir cette documentation.
- De la même manière, une docstring peut être ajoutée pour documenter la classe.

N'oubliez pas la documentation

Utilisation des docstrings et de `help`

- La commande `help` documente la classe, ses méthodes et les attributs « de classe ».
- Comme pour une fonction classique, une docstring peut être ajoutée lors de la définition d'une méthode pour garnir cette documentation.
- De la même manière, une docstring peut être ajoutée pour documenter la classe.
- La commande `dir` sur le nom de la classe liste les mêmes informations que `help`.

N'oubliez pas la documentation

Utilisation des docstrings et de `help`

- La commande `help` documente la classe, ses méthodes et les attributs « de classe ».
- Comme pour une fonction classique, une docstring peut être ajoutée lors de la définition d'une méthode pour garnir cette documentation.
- De la même manière, une docstring peut être ajoutée pour documenter la classe.
- La commande `dir` sur le nom de la classe liste les mêmes informations que `help`.
- La commande `dir` sur un objet rajoute les variables « internes » (initialisées dans une méthode).

Quelques mots sur le modèle mémoire

Alias, variables et paramètres

Quelques mots sur le modèle mémoire

Alias, variables et paramètres

- Une variable ne stocke pas directement les données d'un objet, mais une référence vers la zone mémoire qui contient ces données.

Quelques mots sur le modèle mémoire

Alias, variables et paramètres

- Une variable ne stocke pas directement les données d'un objet, mais une référence vers la zone mémoire qui contient ces données.
- L'affectation ne fait que recopier les valeurs de cette référence, sans dupliquer la zone mémoire.

Quelques mots sur le modèle mémoire

Alias, variables et paramètres

- Une variable ne stocke pas directement les données d'un objet, mais une référence vers la zone mémoire qui contient ces données.
- L'affectation ne fait que recopier les valeurs de cette référence, sans dupliquer la zone mémoire.
- L'utilisation d'une méthode qui modifie l'état interne d'un objet modifie la zone mémoire.

Quelques mots sur le modèle mémoire

Alias, variables et paramètres

- Une variable ne stocke pas directement les données d'un objet, mais une référence vers la zone mémoire qui contient ces données.
- L'affectation ne fait que recopier les valeurs de cette référence, sans dupliquer la zone mémoire.
- L'utilisation d'une méthode qui modifie l'état interne d'un objet modifie la zone mémoire.
- Donc les différentes variables qui référencent un même objet voient toutes la même modification.

Quelques mots sur le modèle mémoire

Alias, variables et paramètres

- Une variable ne stocke pas directement les données d'un objet, mais une référence vers la zone mémoire qui contient ces données.
- L'affectation ne fait que recopier les valeurs de cette référence, sans dupliquer la zone mémoire.
- L'utilisation d'une méthode qui modifie l'état interne d'un objet modifie la zone mémoire.
- Donc les différentes variables qui référencent un même objet voient toutes la même modification.
- Donc un même objet peut être connus sous différents noms (alias).

Quelques mots sur le modèle mémoire

Alias, variables et paramètres

- Une variable ne stocke pas directement les données d'un objet, mais une référence vers la zone mémoire qui contient ces données.
- L'affectation ne fait que recopier les valeurs de cette référence, sans dupliquer la zone mémoire.
- L'utilisation d'une méthode qui modifie l'état interne d'un objet modifie la zone mémoire.
- Donc les différentes variables qui référencent un même objet voient toutes la même modification.
- Donc un même objet peut être connus sous différents noms (alias).
- Le passage de paramètres dans une fonction est un cas particulier d'alias.

Spécialiser les comportements

Héritage

Spécialiser les comportements

Héritage

- Une classe représente un concept.

Spécialiser les comportements

Héritage

- Une classe représente un concept.
- Certains concepts sont liés par l'appartenance : attributs.

Spécialiser les comportements

Héritage

- Une classe représente un concept.
- Certains concepts sont liés par l'appartenance : attributs.
- Certains concepts sont liés par la spécialisation : héritage.

Spécialiser les comportements

Héritage

- Une classe représente un concept.
- Certains concepts sont liés par l'appartenance : attributs.
- Certains concepts sont liés par la spécialisation : héritage.
- L'héritage est une façon de définir des classes permettant de spécifier cette relation de spécialisation.

Spécialiser les comportements

Héritage

- Une classe représente un concept.
- Certains concepts sont liés par l'appartenance : attributs.
- Certains concepts sont liés par la spécialisation : héritage.
- L'héritage est une façon de définir des classes permettant de spécifier cette relation de spécialisation.
- En d'autres termes, l'héritage permet de « recopier » le comportement d'une classe en en remplaçant (ou ajoutant) certains morceaux.

Spécialiser les comportements

Héritage

- Une classe représente un concept.
- Certains concepts sont liés par l'appartenance : attributs.
- Certains concepts sont liés par la spécialisation : héritage.
- L'héritage est une façon de définir des classes permettant de spécifier cette relation de spécialisation.
- En d'autres termes, l'héritage permet de « recopier » le comportement d'une classe en en remplaçant (ou ajoutant) certains morceaux.

Exemple d'héritage

Une classe `FigureGéométrique` représente le concept général des figures géométriques. Elle peut être spécialisée en `Triangle`, `Carré` ou encore `Pentagone`. La notion d'héritage permet de « garder » l'information qu'un `Triangle` est une `FigureGéométrique`.

Spécialiser les comportements

Héritage

Spécialiser les comportements

Héritage

- L'héritage est mis en place par la syntaxe `class ClasseFille(ClasseMère):`.

Spécialiser les comportements

Héritage

- L'héritage est mis en place par la syntaxe `class ClasseFille(ClasseMère):`.
- La classe mère est celle qui va être « recopiée », elle a un comportement plus générique.

Spécialiser les comportements

Héritage

- L'héritage est mis en place par la syntaxe `class ClasseFille(ClasseMère):`.
- La classe mère est celle qui va être « recopiée », elle a un comportement plus générique.
- La classe fille va être spécialisée à partir de la classe mère : elle a un comportement plus spécifique.

Spécialiser les comportements

Héritage

- L'héritage est mis en place par la syntaxe `class ClasseFille(ClasseMère):`.
- La classe mère est celle qui va être « recopiée », elle a un comportement plus générique.
- La classe fille va être spécialisée à partir de la classe mère : elle a un comportement plus spécifique.
- Toutes les méthodes et attributs de la classe mère sont disponibles dans la classe fille implicitement.

Spécialiser les comportements

Héritage

- L'héritage est mis en place par la syntaxe `class ClasseFille(ClasseMère):`.
- La classe mère est celle qui va être « recopiée », elle a un comportement plus générique.
- La classe fille va être spécialisée à partir de la classe mère : elle a un comportement plus spécifique.
- Toutes les méthodes et attributs de la classe mère sont disponibles dans la classe fille implicitement.
- On peut redéfinir les comportement de la classe fille en ré-écrivant les méthodes ad-hoc.

Spécialiser les comportements

Héritage

- L'héritage est mis en place par la syntaxe `class ClasseFille(ClasseMère):`.
- La classe mère est celle qui va être « recopiée », elle a un comportement plus générique.
- La classe fille va être spécialisée à partir de la classe mère : elle a un comportement plus spécifique.
- Toutes les méthodes et attributs de la classe mère sont disponibles dans la classe fille implicitement.
- On peut redéfinir les comportement de la classe fille en ré-écrivant les méthodes ad-hoc.
- On peut accéder à une méthode de la classe mère qui a été redéfinie dans la classe fille par `ClasseMère.nomMéthode()`.

Spécialiser les comportements

Héritage

- L'héritage est mis en place par la syntaxe `class ClasseFille(ClasseMère):`.
- La classe mère est celle qui va être « recopiée », elle a un comportement plus générique.
- La classe fille va être spécialisée à partir de la classe mère : elle a un comportement plus spécifique.
- Toutes les méthodes et attributs de la classe mère sont disponibles dans la classe fille implicitement.
- On peut redéfinir les comportement de la classe fille en ré-écrivant les méthodes ad-hoc.
- On peut accéder à une méthode de la classe mère qui a été redéfinie dans la classe fille par `ClasseMère.nomMéthode()`.
- Toutes les classes héritent (implicitement) de la classe object.

Connaître les comportements

Héritage multiple

Connaître les comportements

Héritage multiple

- Il est possible de créer une classe possédant plusieurs classes mère.

Connaître les comportements

Héritage multiple

- Il est possible de créer une classe possédant plusieurs classes mère.
- Toutes les méthodes et attributs des classes mères sont hérités.

Connaître les comportements

Héritage multiple

- Il est possible de créer une classe possédant plusieurs classes mère.
- Toutes les méthodes et attributs des classes mères sont hérités.
- En cas d'ambiguïté, la classe mère déclarée en premier a priorité.

Connaître les comportements

Héritage multiple

- Il est possible de créer une classe possédant plusieurs classes mère.
- Toutes les méthodes et attributs des classes mères sont hérités.
- En cas d'ambiguïté, la classe mère déclarée en premier a priorité.

Tester une relation ou une appartenance

Connaître les comportements

Héritage multiple

- Il est possible de créer une classe possédant plusieurs classes mère.
- Toutes les méthodes et attributs des classes mères sont hérités.
- En cas d'ambiguïté, la classe mère déclarée en premier a priorité.

Tester une relation ou une appartenance

- La fonction `issubclass` prend deux noms de classe en paramètre. Elle permet de déterminer si la première est une classe fille de la seconde.

Connaître les comportements

Héritage multiple

- Il est possible de créer une classe possédant plusieurs classes mère.
- Toutes les méthodes et attributs des classes mères sont hérités.
- En cas d'ambiguïté, la classe mère déclarée en premier a priorité.

Tester une relation ou une appartenance

- La fonction `issubclass` prend deux noms de classe en paramètre. Elle permet de déterminer si la première est une classe fille de la seconde.
- La fonction `isinstance` prend un objet et un nom de classe en paramètre. Elle permet de déterminer si la classe de l'objet est celle passée en paramètre (ou une de ses classes filles) ou non.

Des fonctions et des méthodes

Classification des méthodes

Des fonctions et des méthodes

Classification des méthodes

- Il arrive d'écrire des fonctions (pour séparer logiquement du code ou pour le factoriser) qui ne sont utiles qu'à une classe en particulier.

Des fonctions et des méthodes

Classification des méthodes

- Il arrive d'écrire des fonctions (pour séparer logiquement du code ou pour le factoriser) qui ne sont utiles qu'à une classe en particulier.
- Pour renforcer la sémantique associée à de telle fonction, il serait logique de les incorporer au corps de la classe.

Des fonctions et des méthodes

Classification des méthodes

- Il arrive d'écrire des fonctions (pour séparer logiquement du code ou pour le factoriser) qui ne sont utiles qu'à une classe en particulier.
- Pour renforcer la sémantique associée à de telle fonction, il serait logique de les incorporer au corps de la classe.
- Le décorateur `@staticmethod` permet de se passer du premier argument `self`.

Des fonctions et des méthodes

Classification des méthodes

- Il arrive d'écrire des fonctions (pour séparer logiquement du code ou pour le factoriser) qui ne sont utiles qu'à une classe en particulier.
- Pour renforcer la sémantique associée à de telle fonction, il serait logique de les incorporer au corps de la classe.
- Le décorateur `@staticmethod` permet de se passer du premier argument `self`.
- De telles méthodes peuvent être appellées soit avec le nom de la classe, soit à partir d'un objet.

Des fonctions et des méthodes

Classification des méthodes

- Il arrive d'écrire des fonctions (pour séparer logiquement du code ou pour le factoriser) qui ne sont utiles qu'à une classe en particulier.
- Pour renforcer la sémantique associée à de telle fonction, il serait logique de les incorporer au corps de la classe.
- Le décorateur `@staticmethod` permet de se passer du premier argument `self`.
- De telles méthodes peuvent être appellées soit avec le nom de la classe, soit à partir d'un objet.
- Dans le même ordre d'idée, le décorateur `@classmethod` remplace le premier paramètre `self` par le paramètre `cls` qui est la classe (de l'objet) faisant appel à la méthode.

Exceptions

Présentation technique

Une classe presque comme les autres

Présentation technique

Une classe presque comme les autres

- Toute classe fille de `BaseException` est considéré comme une exception.

Présentation technique

Une classe presque comme les autres

- Toute classe fille de `BaseException` est considéré comme une exception.
- Dans la pratique, il est courant de voir une définition simple de la forme : `class NomDeMonException(Exception): pass`

Présentation technique

Une classe presque comme les autres

- Toute classe fille de `BaseException` est considéré comme une exception.
- Dans la pratique, il est courant de voir une définition simple de la forme : `class NomDeMonException(Exception): pass`
- Les exceptions sont des classes qui représentent des erreurs dans l'utilisation du programme.

Présentation technique

Une classe presque comme les autres

- Toute classe fille de `BaseException` est considéré comme une exception.
- Dans la pratique, il est courant de voir une définition simple de la forme : `class NomDeMonException(Exception): pass`
- Les exceptions sont des classes qui représentent des erreurs dans l'utilisation du programme.
- Une exception qui n'est pas traitée par le programme conduit à un message d'erreur.

Présentation pratique

Créer des exceptions, l'exception à la règle

Présentation pratique

Créer des exceptions, l'exception à la règle

- En tant que classe, on peut toujours créer une exception en faisant appel au constructeur de la classe.

Présentation pratique

Créer des exceptions, l'exception à la règle

- En tant que classe, on peut toujours créer une exception en faisant appel au constructeur de la classe.
- Pour déclencher l'erreur associée à une exception, on utilise le mot-clé `raise` associé au nom de la classe ou à une instance.

Présentation pratique

Créer des exceptions, l'exception à la règle

- En tant que classe, on peut toujours créer une exception en faisant appel au constructeur de la classe.
- Pour déclencher l'erreur associée à une exception, on utilise le mot-clé `raise` associé au nom de la classe ou à une instance.
- `raise` arrête l'exécution du bloc en cours, puis des blocs l'encadrant, jusqu'à ce qu'elle soit traitée ou qu'elle ne puisse plus remonter et que l'erreur arrête le programme.

Présentation pratique

Créer des exceptions, l'exception à la règle

- En tant que classe, on peut toujours créer une exception en faisant appel au constructeur de la classe.
- Pour déclencher l'erreur associée à une exception, on utilise le mot-clé `raise` associé au nom de la classe ou à une instance.
- `raise` arrête l'exécution du bloc en cours, puis des blocs l'encadrant, jusqu'à ce qu'elle soit traitée ou qu'elle ne puisse plus remonter et que l'erreur arrête le programme.
- Pendant le traitement d'une exception, `raise` seul permet de relancer la même exception pour qu'elle remonte à nouveau à travers les blocs d'instructions.

Essayer et se tromper

Traiter des exceptions : `try...except...else...finaly`

Essayer et se tromper

Traiter des exceptions : `try`...`except`...`else`...`finaly`

- Une exception qui est levée à l'intérieur d'un bloc `try` peut être traitée par un bloc `except` associé.

Essayer et se tromper

Traiter des exceptions : `try`...`except`...`else`...`finaly`

- Une exception qui est levée à l'intérieur d'un bloc `try` peut être traitée par un bloc `except` associé.
- Un bloc `except` prend un (ou plusieurs) nom de classe et n'est exécuté que si une exception de cette classe est levée dans le bloc `try`

Essayer et se tromper

Traiter des exceptions : `try`...`except`...`else`...`finaly`

- Une exception qui est levée à l'intérieur d'un bloc `try` peut être traitée par un bloc `except` associé.
- Un bloc `except` prend un (ou plusieurs) nom de classe et n'est exécuté que si une exception de cette classe est levée dans le bloc `try`
- Un seul bloc `except` est exécuté et ils sont testé dans l'ordre de leur écriture.

Essayer et se tromper

Traiter des exceptions : `try`...`except`...`else`...`finaly`

- Une exception qui est levée à l'intérieur d'un bloc `try` peut être traitée par un bloc `except` associé.
- Un bloc `except` prend un (ou plusieurs) nom de classe et n'est exécuté que si une exception de cette classe est levée dans le bloc `try`
- Un seul bloc `except` est exécuté et ils sont testé dans l'ordre de leur écriture.
- Un bloc `except` sans nom de classe est exécuté quelque soit l'exception.

Essayer et se tromper

Traiter des exceptions : `try`...`except`...`else`...`finaly`

- Une exception qui est levée à l'intérieur d'un bloc `try` peut être traitée par un bloc `except` associé.
- Un bloc `except` prend un (ou plusieurs) nom de classe et n'est exécuté que si une exception de cette classe est levée dans le bloc `try`
- Un seul bloc `except` est exécuté et ils sont testé dans l'ordre de leur écriture.
- Un bloc `except` sans nom de classe est exécuté quelque soit l'exception.
- Un bloc `else` optionel est exécuté uniquement s'il n'y a pas eu d'exceptions dans le bloc `try`.

Essayer et se tromper

Traiter des exceptions : `try`...`except`...`else`...`finaly`

- Une exception qui est levée à l'intérieur d'un bloc `try` peut être traitée par un bloc `except` associé.
- Un bloc `except` prend un (ou plusieurs) nom de classe et n'est exécuté que si une exception de cette classe est levée dans le bloc `try`
- Un seul bloc `except` est exécuté et ils sont testé dans l'ordre de leur écriture.
- Un bloc `except` sans nom de classe est exécuté quelque soit l'exception.
- Un bloc `else` optionel est exécuté uniquement s'il n'y a pas eu d'exceptions dans le bloc `try`.
- Un bloc `finaly` est exécuté dans tous les cas.

Exemple

Utilisation d'exceptions utilisateur et prédéfinie

```
import sys

class FileTooBig(Exception):
  pass

for nomFichier in sys.argv[1:]:
  try:
    fichier = open(nomFichier,'r')
  except IOError:
    print('Impossible d\'ouvrir le fichier', nomFichier)
  else:
    try:
      listeLignes = fichier.readlines()
      if len(listeLignes) > 100:
        raise FileTooBig(str(len(listeLignes)) + " lignes")
    except FileTooBig:
      print('Notre exception apparait, mais on ne veut pas la traiter')
      raise
    else:
      for ligne in listeLignes:
        print(ligne)
    finaly:
      fichier.close()
```