Informatique fondamentale

Variables, types et données

R.Gosswiller

Bases de la syntaxe Python

Le Python

Langage de programmation

Instructions
Programmation et exécution séquentielle

Langage interprêté

Pas de compilation
Interprêteur
Pas de binaire

La console Python

Principe

Console d'interprêtation dynamique du Python
Interprêtation syntaxique séquentielle

Utilisation

```
1          python3
2          >>> a = 3
3          >>> print(a)
4          3
```

Fichiers .py

Syntaxe

Tout le code Python est regroupé dans un ou plusieurs fichiers .py

Interprêteur

Interprêtation du code par l'interprêteur Python

```
1 python3 file.py
```

Mots-clés

Liste des mots réservés

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

Variables et typage

Les variables

Définition

Une variable est la représentation, dans un code, d'une donnée stockée en mémoire.
Elle permet la manipulation de cette donnée au sein d'un programme

Principe

Elément basique de la représentation de l'information
Typage dynamique (déterminé par le compilateur/interprète)

Les variables

Opérations

Une variable correspond à un emplacement mémoire. L'on peut donc opérer deux opérations de base : lire et écrire.

Ecriture

Ecrire correspond à donner une valeur à une variable.

```
1 var = 15
```

Les variables

Créer une variable

Créer une variable se fait en deux étapes : Définir et assigner.

Définition

L'étape de définition prépare l'espace nécessaire à la variable en mémoire, dépendant du type de la variable.
Cette étape est transparente en Python.

Assignation

L'étape d'assignation correspond à donner une valeur par défaut à une variable.

Les variables

```
1 var = 15
2
3 var = 'Toto'
4
5 var = False
```

Les variables

Assignation multiple

```
1 x = y = 7
2
3 a, b, c = 12, 3.4, 'Toto'
```

Types primitifs

int	Entier
long	Entier dynamique
float	Flottant
complex	Nombre complexe
byte (str)	Chaîne ASCII
unicode	Chaîne UNICODE
file	Fichier
bool	Booléen

Règles de nommage

- Minuscules
- camelCase
- Alphabet standard
- Pas de caractères spéciaux

Exemples

```
1 myWonderfulInt, container, variable
```

La fonction type

Principe

La fonction type renvoie le type d'une variable

Exemple

```
1 var = 15;
2 # <class 'int'>
3 var = True;
4 # <class 'bool'>
```

La fonction print

Principe

La fonction print affiche le contenu d'une variable

Syntaxe

```
1 print(value)
```

Exemple

```
1 >>> var = 15;
2 >>> print(var)
3 15
4 >>> var = True;
5 >>> print(var)
6 'True'
```

Structures conditionnelles

Structure conditionnelle

If

L'instruction If permet de conditionner l'exécution d'une instruction
Si la condition est vraie, l'instruction (ou l'ensemble d'instructions) est exécutée

Syntaxe

```
1 if( a > 100):
2     doStuff()
```

Structure conditionnelle

Else

Réalisation d'une instruction uniquement si le If correspondant n'est pas vrai

Syntaxe

```
1 if(condition):
2     consequence
3 else:
4     consequenceB
```

Structure conditionnelle

Elif

Réalisation d'une instruction si le If correspondant n'est pas vrai et qu'une autre condition est vraie

Syntaxe

```
1 if(condition):
2     consequence
3 elif (conditionB):
4     consequenceB
```

Exemple

```
1 if( a > 100):
2     doStuff()
3 elif (a > 50):
4     doOtherStuff()
```

Opérateurs de comparaison

Opérateur	Syntaxe	Rôle
>,	a > b	Supérieur
>=	a >= b	Supérieur ou égal
<	a < b	Inférieur
<=	a <= b	Inférieur ou égal
==	a == b	Test d'égalité ou égal
!=	a != b	Différence

Composition de conditions

and

Opérateur **and**

ET logique

```
1 if(a and b):
2     doStuff() // Si a ET b sont vraies
3
4 if(a > 100 and b < 50):
5     doStuff() # Si a > 100 ET b < 50
```

Composition de conditions

or

Opérateur **or**

OU logique

```
1 if(a or b):
2     doStuff() // Si a OU b sont vraies
3
4 if(a > 100 or b < 50):
5     doStuff() # Si a > 100 OU b < 50
```

Composition de conditions

not

Opérateur **not**

NON logique

```
1 if(not(a)):
2     doStuff()  // Si a n'est␣pas␣vraie
3
4 if(not(a␣>␣100)):
5 ␣␣␣␣doStuff()␣#␣Si␣a␣<=␣100
```

Fonctions

Les fonctions

Définition

Une fonction est une portion de code dédiée à la réalisation d'une tâche.
Elle est définie selon 5 éléments :

- Nom
- Type et valeur de retour
- Arguments d'entrée
- Instructions

Terminologie

Procédure

Nom, arguments, instructions

Sous-programme

Nom, instructions, type de retour

Prototype

Nom, type de retour, arguments

Les fonctions

Il existe deux étapes dans l'utilisation d'une fonction

Définition

On définit la fonction, ses entrées/sorties et les tâches qu'elle doit réaliser

Appel

On fait appel au code de la fonction pour réaliser une tâche à un moment précis de l'exécution

Le type de retour

Utilisation

Chaque fonction est une portion de code isolée dédiée à la réalisation d'une tâche.
Cette tâche pouvant être la réalisation d'un calcul ou d'une opération, le type de retour permet de renvoyer la valeur calculée

Les arguments

Arguments de fonction

Les arguments ou paramètres de fonctions sont les valeurs demandées en entrée
Ce sont des variables locales à la fonction
Chaque variable doit avoir une valeur définie lors de l'appel de la fonction

Prototypes

Définition

La version primitive d'une fonction s'appelle son prototype
Il s'agit d'une pré-définition de la fonction

Utilisation

Un prototype renseigne sur les entrées et sorties de la fonction, sans en donner les instructions
C'est une méthode de représentation par boîte noire

Portée des variables

La portée d'une variable désigne le périmètre dans lequel elle est définie
Plus la portée d'une variable est étendue, plus son accès est possible depuis n'importe où dans un programme

Mémoire

En mémoire, la portée se manifeste par le droit d'accès ou non à un emplacement donné
Chaque instruction peut ou non accéder à certains emplacements mémoire, selon la portée de la variable associée

La programmation impérative

Définition

La programmation impérative est un modèle de programmation représentant un programme comme un ensemble d'opérations consécuvitement exécutées par la machine
Le modèle consiste à morceler la structure d'un programme en tâches et sous-tâches décrivant le découpage fonctionnel d'un problème

Attention

Ne pas confondre programmation impérative avec programmation fonctionnelle

Implémenter des fonctions en Python

Fonctions en Python

Principe

Une fonction est définie par le mot-clé **def** en Python

Syntaxe

```
1 def doSomething():
2         instruction1
3         instruction2
4
5 doSomething()
```

Le mot-clé return

return

return permet de renvoyer la valeur d'une variable donnée en sortie d'une fonction
Cette valeur pourra être stockée dans une variable pour utilisation ultérieure

Syntaxe

```
1 def doStuff():
2         a = 20
3         return a
4
5 var = doStuff()
6 print(var) // Affiche 20
```

Règles de nommage

Fonctions

- Minuscules
- camelCase
- Alphabet standard
- Verbes simples
- Pas de caractères spéciaux

Exemples

```
1 ajouteDix, verifieParite, racineCarree
```

Arguments en Python

Détails

Les arguments en Python sont définis à la volée dans le prototype de fonction
Il n'est pas nécessaire de les typer

```
1 def doMult(a, b):
2         return a * b
```

Arguments en Python

Chaque paramètre est obligatoire lors de l'appel

```
1 def doMult(a, b):
2         return a * b
3
4 var = doMult(2) // Erreur
```

Portée

Arguments

Chaque argument de fonction a une portée locale à la fonction

Syntaxe

```
1 def doStuff(a, b):
2         a = a + 1  // Variable locale
3         return a * b // Variable locale
4
5 var = a + 1 // Hors scope
```

Variables globales

Définition

Une variable globale a une portée étendue à tout le programme

Avantages

Accessibilité, définition unique

Inconvénients

Gestion mémoire, maintenabilité, clarté du code, dépendances

Récursivité

La récursivité

"Pour comprendre la récursivité, il vous faut d'abord comprendre la récursivité"

La récursivité

Le modèle de la récursivité consiste en la résolution d'une opération par la résolution de ses composantes

Définition

La récursivité est un processus itératif basé sur la référence d'un objet à lui-même

Fonctions récursives

Principe

Une fonction récursive contient au moins un appel à elle-même

Syntaxe

```
1 def doAdd(a):
2         if (a==1):
3                 return a
4         else:
5                 return 1+doAdd(a-1)
```

Conditions de récursivité

Condition d'arrêt

Il existe au moins un cas dans lequel l'appel récursif n'est pas effectué

Pas

Il existe au moins un cas dans lequel l'appel récursif est effectué

Récursivité terminale

Définition

Dans le cas où l'appel récursif est la dernière instruction d'une fonction et que cette instruction est isolée, on parle alors de récursivité terminale

Syntaxe

```
1 def doTermRec(a, acc):
2         if(a==1):
3                 return acc
4         else:
5                 return doTermRec(a-1, acc+1)
```

Intérêt

Pas de cumul des appels

Pas d'augmentation de la complexité en espace

Convergence

Une suite d'appels récursifs doit converger tôt ou tard vers la condition d'arrêt.
La vérification de cette convergence permet d'éviter les **boucles infinies**

Récursivité en Python

Implémentation

Implémenter de la récursivité en Python implique de définir une fonction qui va s'appeler elle-même
On introduit la condition d'arrêt et la condition de récursion par une structure conditionnelle

Syntaxe

```
1 def fact(value):
2         if(value == 1):
3                 return value
4         else:
5                 return fact(value-1)*value
```

Récursivité par l'exemple

Exemple

Problématique

On souhaite réaliser une fonction récursive pour calculer les termes de la suite de Fibonacci

Rappel

La suite de Fibonacci est une suite d'entiers où chaque valeur est égale à la somme des deux précédentes

Etape 1

Déterminer les points d'entrée/sortie

Arguments

La fonction prendra en entrée 1 entier n, égal à l'indice de la suite que l'on veut calculer

Retour

On souhaite calculer le n-ième indice de la suite

Etape 2

Définir la fonction

Syntaxe

```
1 def fibo(n):
```

Etape 3

Déterminer les conditions d'arrêt

Conditions

On s'arrête lorsque l'indice donné en entrée atteint 0 ou 1

Syntaxe

```
1  def fibo(n):
2          if(n==1):
3                  return 1
4          if(n==0):
5                  return 0
```

Etape 4

On définit les conditions de l'appel récursif

Appel

A chaque indice, on calcule les valeurs des deux indices précédents

Syntaxe

```
1  def fibo(n):
2          if(n==1):
3                  return 1
4          if(n==0):
5                  return 1
6
7          return fibo(n-1)+fibo(n-2)
```

Etape 5

On teste le résultat et la convergence

Résultats

```
1  fibo(5)
2  return fibo(4) + fibo(3)
3  return fibo(3)+ fibo(2) + fibo(2) + fibo(1)
4  return fibo(2) + fibo(1) +
5  fibo(1) + fibo(0) + fibo(1)+fibo(0) + 1
6  return fibo(1) + fibo(0) + 1 + 1 + 0 + 1 + 0 + 1
7  return 1 + 0 + 1 + 1 +0 +1 + 0 + 1
8  return 5
```