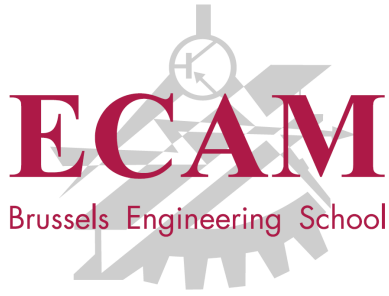


IC1T Programmation

Cours 3

Chaînes de caractères et fonctions

Sébastien Combéfis, Quentin Lurkin

Rappels

- Scripts Python
  - exécuter un script
  - fonction `print()`
  - paramètres nommés d'une fonction
  - fonction `input()`
- Contrôle de flux
  - opérateurs de comparaison, booléens et logiques
  - instuctions blocs
  - `if elif else`
  - boucle `while`

Objectifs

- Manipulations de chaînes de caractères
  - opérations sur les chaînes de caractères
  - accès aux caractères et *slicing*
  - les `string` sont non-modifiables
- Définition de fonctions
  - définition et appel de fonctions
  - paramètres et valeurs de retour
  - variables locales et globales

String

Chaînes de caractères

- Les chaînes de caractères sont un type de valeur.

  On les appelle `string` en programmation.

- Un `string` est composé d'une suite ordonnée de caractères.

  Chaque caractère a une place précise dans la suite.

- Parmi tous les caractères, on trouve les chiffres.

  Un `string` composé de chiffres n'est pas un entier.

Opérations sur les `string`

- L'opérateur `+` entre deux `string` permet de créer une nouvelle chaîne contenant les deux opérandes bout-à-bout.

  On appelle cette opération la concaténation.

- L'opérateur `*` entre un entier `n` et un `string` permet de créer une nouvelle chaîne contenant `n` fois la chaîne de caractères.

```
1  msg = "Hello" + " " + "World!!"   # msg contient 'Hello World!!'
2  s = 3*"bla"                       # s contient 'blablabla'
```

Accès aux caractères

- Il est possible d'accéder aux caractères individuels.

  Pour cela, on utilise les accolades [].

- Chaque caractère est repéré par son numéro d'ordre.

  Le numéro d'ordre est appelé indice.

- Les indices commencent à zéro.

  Les indices négatifs permettent d'indiquer un caractère en partant de la fin de la chaîne.

```
1  s = "j'aime les pâtes"
2  print( s[12] )                # affiche 'â'
3  print( s[0] )                 # affiche 'j'
4  print( s[-1] )                # affiche 's'
5  print( s[-4] )                # affiche 'â'
```

Le *slicing*

- Il est possible de référencer une sous-chaîne c-à-d une "tranche" de la chaîne.

  On sépare les deux indices délimitant la tranche par un ":".

- La tranche `[m:n]` référence la sous-chaîne qui va du m^e caractère *(compris)* au n^e caractère *(non-compris)*.

- Si on omet le premier indice, la tranche démarre au début de la chaîne.

  Idem avec le deuxième indice et la fin de la chaîne.

```
1  s = "j'aime les pâtes"
2  print( s[7:10] )              # affiche 'les'
3  print( s[11:-1] )             # affiche 'pâte'
4  print( s[11:] )               # affiche 'pâtes'
5  print( s[:6] )                # affiche 'j'aime'
```

Un `string` est non-modifiable

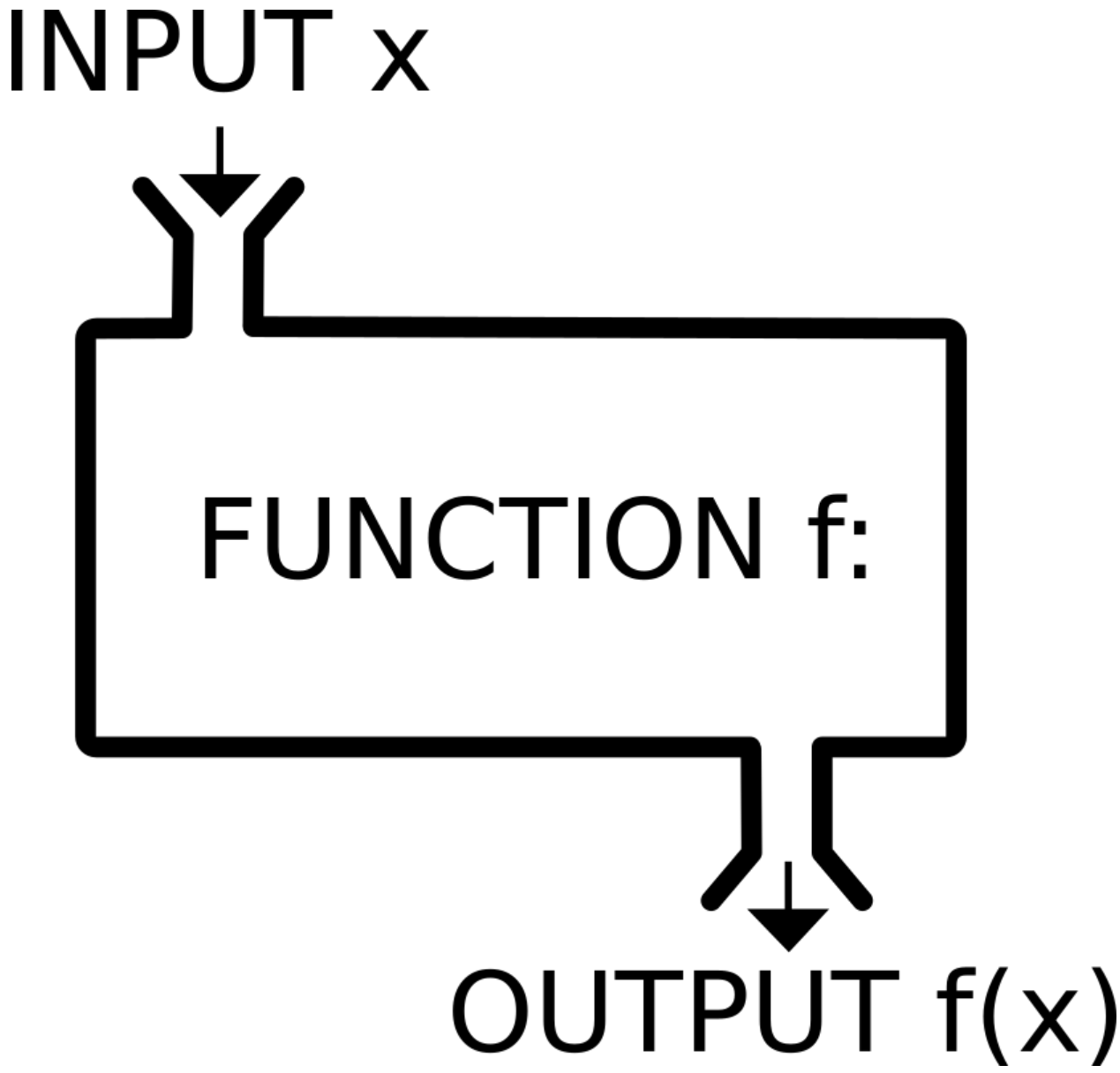
- Une fois créé, un `string` ne peut plus être modifié.
- On peut toujours créer un nouveau `string` avec une valeur adaptée.

```
1  s = "j'aime les pâtes"
2  s[12] = "u"                    # provoque une erreur !
3  s = s[:12] + "u" + s[13:]      # crée un nouveau string adapté
```

Longueur d'un `string`

- Il est possible de connaitre le nombre de caractères contenus dans un `string` grâce à la fonction `len()`.

```
1  s = "j'aime les pâtes"
2  print(len(s))                  # affiche 16
```


INPUT x
FUNCTION f:
OUTPUT f(x)

Définitions de fonctions

- Nous avons déjà vu plusieurs fonctions prédéfinies :
  `print()`, `sqrt()`, `len()`, `input()`, `int()`, ...
- Une fonction est une "boite noire" qui peut recevoir une ou plusieurs valeurs en arguments et renvoie une valeur de sortie.
- Définir ses propres fonctions sert à deux choses :
  - Eviter de répéter plusieurs fois le même code.
  - Structurer les longs programmes en les découpant en petites parties simples.

Fonctions sans argument ni valeur de retour

- Voici la définition d'une fonction qui affiche la table de 7 :

```
1  def table7():
2      n = 1
3      while n <= 10:
4          print(n, "x 7 =", n * 7)
5          n += 1
```

- Une fois la fonction définie, on peut l'appeler autant de fois qu'on veut :

```
1  # affichons 2 fois la table de 7
2  table7()
3  table7()
```

- Une fonction n'est exécutée que lorsqu'on l'appelle

Fonctions sans argument ni valeur de retour

- La fontion `table7()` ne retourne rien :

```
1  x = table7()      # affiche la table de 7
2  print(x)          # affiche None
```

- Comme elle ne prend pas de paramètre, elle affiche toujours la même chose.

Fonctions avec arguments

- La ou les valeurs reçues en paramètres sont utilisées lors de l'exécution de la fonction :

```
1  # définition de la fonction table().
2  def table(base):
3      n = 1
4      while n <= 10:
5          print(n, "x", base, "=", n * base)
6          n += 1
7
8  # appels à la fonction table()
9  table(3)          # affiche la table de 3
10 table(base = 8)   # affiche la table de 8
11 a = 42
12 table(a)          # ???
```

Fonctions avec arguments

- On peut aussi avoir plusieurs arguments :

```
1  # définition de la fonction table().
2  def table(base, start, length):
3      n = start
4      while n < start + length:
5          print(n, "x", base, "=", n * base)
6          n += 1
7
8  # appels à la fonction table()
9  table(3, 1, 10)
10 table(base=8, start=5, length=2)
```

Fonctions avec arguments

- On peut aussi avoir des valeurs par défaut pour les arguments :

```
1  # définition de la fonction table().
2  def table(base, start=1, length=10):
3      n = start
4      while n < start + length:
5          print(n, "x", base, "=", n * base)
6          n += 1
7
8  # appels à la fonction table()
9  table(8, 5, 2)
10 table(8)
11 table(8, 5)
12 table(8, length=2)
```

Fonctions avec valeur de retour

- Le mot-clé `return` permet de définir la valeur de retour :

```
1  # définition de la fonction multiply()
2  def multiply(a, b):
3      return a * b
4
5
6  res = multiply(7, 9)     # appel à la fonction. N'affiche rien.
7  print(res)               # affiche 63
8  print(multiply(6, 7))    # affiche 42
```

Fonctions avec valeur de retour

- On peut utiliser une fonction dans le corps d'une autre :

```
1  # définition de multiply()
2  def multiply(a, b):
3      return a * b
4
5  # définition de table()
6  def table(base, start=1, length=10):
7      n = start
8      while n < start + length:
9          # utilisation de multiply()
10         print(n, "x", base, "=", multiply(n, base))
11         n += 1
12
13 # appel à la fonction table()
14 table(5, length=3)
```

Fonctions avec valeur de retour

- Lorsque le mot-clé `return` est rencontré, la fonction s'arrête et le programme se poursuit après l'appel.

  Cela permet quelques simplifications :

```
1  def abs(x):
2      if x < 0:
3          return -x
4      else:
5          return x
```

```
1  # version simplifiée
2  def abs(x):
3      if x < 0:
4          return -x
5      return x
```

Variables locales et globales

- Dès que l'on a des fonctions, il faut comprendre la différence entre variables locales et globales.
- Une variable globale est définie en dehors de toute fonction. Elle est accessible dans l'ensemble du programme.
- Une variable locale est définie dans une fonction. Elle est accessible uniquement dans cette fonction.

```
1  def tvac(amount):
2      n = 1 + interest / 100    # ce n ci est local
3      return amount * n         # utilise le n local
4
5  interest = 21                 # interest est globale
6  n = 25                        # ce n ci est global
7  print(tvac(n))
```

Variables locales et globales

- Une variable locale est inaccessible hors de la fonction où elle est définie :

```
1  def fun():
2      a = 12
3
4  fun()
5  print(a)          # error
```

- Une variable globale est accessible partout :

```
1  def fun():
2      print(a)      # affiche 12
3
4  a = 12
5  fun()
```

Variables locales et globales

- Lorsqu'une variable locale porte le même nom qu'une globale, priorité à la locale :

```
1  def fun():
2      a = 42
3      print("dans fun :", a)          # affiche 42
4
5  a = 12
6  fun()
7  print("en dehors de fun :", a)     # affiche 12
```

- Même avant sa définition dans la fonction :

```
1  def fun():
2      print("globale :", a)          # error
3      a = 42
4      print("locale :", a)
5
6  a = 12
7  fun()
```

Variables locales et globales

- Si on souhaite modifier une variable globale de l'intérieur d'une fonction :

```
1  def fun():
2      global a
3      a = 42
4
5  a = 12
6  fun()
7  print(a)        # affiche 42
```

- Ce n'est pas une bonne pratique.

Une fonction est une valeur

- Une fonction peut être mise dans une variable :

```
1  def add(a, b):
2      return a + b
3
4  somme = add
5  print(somme(1, 2))      # affiche 3
```

Une fonction est une valeur

- Une fonction peut être passée en paramètre à une fonction :

```
1  def add(a, b):
2      return a + b
3
4  def multiply(a, b):
5      return a * b
6
7  def table(base, start=1, length=10, symbol="*", op=multiply):
8      n = start
9      while n < start + length:
10         print(n, symbol, base, "=", op(n, base))
11         n += 1
12
13 table(4, length=2)
14 table(4, length=2, symbol="+", op=add)
```

Crédits

- Borat : Cultural Learnings of America for Make Benefit Glorious Nation of Kazakhstan, Twentieth Century Fox
- https ://en.wikipedia.org/wiki/Function_(mathematics)#/media/File :Function_machine2.svg