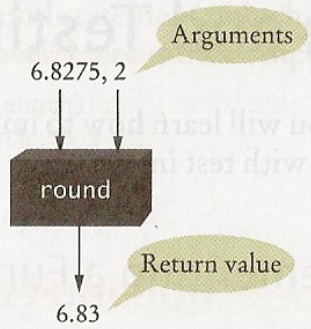
Les fonctions

The round Function as a Black Box



Function Definition

Syntax `def` *functionName*(*parameterName*$_1$, *parameterName*$_2$, . . .) :
statements

Name of function
Name of parameter variable
Function header
Function body, executed when function is called.

```
def cubeVolume(sideLength) :
   volume = sideLength ** 3
   return volume
```

return statement exits function and returns result.

Program with Functions

By convention, `main` is the starting point of the program.

The `cubeVolume` function is defined below.

```
def main() :
   result = cubeVolume(2)
   print("A cube with side length 2 has volume", result)

def cubeVolume(sideLength) :
   volume = sideLength ** 3
   return volume

main()
```

This statement is outside any function definitions.

Les fonctions

1. Introduction

L'un des concepts les plus importants en programmation est celui de « fonctions ». ***Les fonctions permettent de décomposer un programme complexe en une série de sous-programmes plus simples***, lesquels peuvent à leur tour êtres décomposés en fragments plus petits.
D'autre part, ***les fonctions sont réutilisables*** : si nous disposons d'une fonction capable de calculer une racine carrée par exemple, nous pouvons l'utiliser un peu partout dans nos programmes sans avoir à la ré-écrire à chaque fois.
Les fonctions sont donc les éléments structurant de base de tout langage procédural. Elles offrent différents avantages :

✓ ***Evite la répétition** : on peut « factoriser » une portion de code qui se répète lors de son exécution.*
✓ ***Met en relief les données et les résultats** : entrées et sorties de la fonction.*
✓ ***Permet la réutilisation** : mécanisme de l'import.*
✓ ***Décompose une tâche** complexe en tâches plus simples.*

2. Les fonctions prédéfinies

2.1. La fonction `print()`

Nous avons déjà rencontrés cette fonction. Précisons simplement ici qu'elle permet d'afficher n'importe quel nombre de valeurs fournies en arguments (dans les parenthèses). On peut afficher les valeurs en ligne avec l'instruction : `end`

```
>>> n =0
>>> while n<6:
...     print("zut", end ="")
...     n = n+1
...
zutzutzutzutzut
```

2.2. La fonction `input()`

La plupart des scripts élaborés nécessitent à un moment une ***intervention de l'utilisateur*** (entrée d'un paramètre, clic de souris sur un bouton…). La méthode la plus simple que nous avons vu consiste à utiliser la fonction `input().` Cette fonction provoque une interruption dans le programme courant.

```
prenom = input("Entrez votre prénom : ")
print("Bonjour,", prenom)
```

Soulignons que la fonction `input()` renvoie toujours une ***chaîne de caractères***. Si vous souhaitez que l'utilisateur entre une valeur numérique, vous devrez convertir la valeur entrée (qui sera de tt façon un string) en numérique via `int()` ou `float().`

```
n = int(input("entrez un nombre n : "))
x = int(input("entrez un nombre x : "))
```

2.3. Importer un module de fonctions prédéfinies

Vous avez déjà rencontrés d'autres ***fonctions intégrées à Python***, comme la fonction len ( ), qui par exemple, permet de connaître la longueur d'une chaîne de caractères. Il existe un nombre important de fonctions prédéfinies, mais c'est souvent les mêmes qui reviennent et que je vous exposerai ci-dessous. Les autres sont regroupées dans des ***modules*** (fichiers qui regroupent un ensemble de fonctions) séparés comme le module ***math***.

```
# Démo : utilisation des fonctions du module <math>

from math import *

nombre = 121
angle = pi/6        # soit 30°
                    # (la bibliothèque math inclut aussi la définition de pi)
print("racine carrée de", nombre, "=", sqrt(nombre))
print("sinus de", angle, "radians", "=", sin(angle))
```

ffichage suivant :

```
racine carrée de 121 = 11.0
sinus de 0.523598775598 radians = 0.5
```

Les fonctions prédéfinies les ***plus utilisées dans python*** sont les suivantes :

... il y en a beaucoup ☺

2. Built-in Functions

The Python interpreter has a number of functions built into it that are always ava
They are listed here in alphabetical order.

		Built-in Functions		
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	apply()
delattr()	help()	next()	setattr()	buffer()
dict()	hex()	object()	slice()	coerce()
dir()	id()	oct()	sorted()	intern()

Voici 2 exemples d'application de ces fonctions intrinsèques de Python :

- `eval` : renvoie l'évaluation d'une expression contenue dans une chaîne

```
>>> x=56.3
>>> rep = raw_input("Expression en x: ")
Expression en x: x**2-100
>>> eval(rep)
3069.6899999999996
>>> print eval(rep)
3069.69
```

```
x=56.2
rep=input("entrer une expression: ")
print(rep)

print()

print(eval(rep))
```

- Les fonctions `min`, `max` et `sum` acceptent une liste (un itérable) en argument

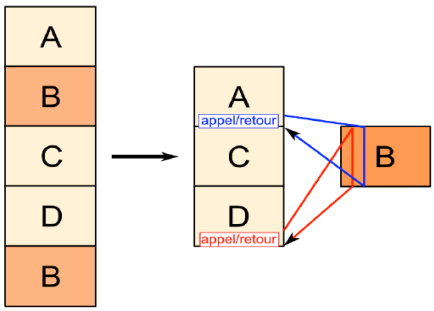
```
>>> L1 = [10, 11, 12, 13, 14]
>>> sum(L1)
60
>>> min(L1), max(L1)
(10, 14)
```

Les fonctions

3. Définir une fonction

Les scripts que vous avez écrits jusqu'à présent étaient à chaque fois très courts, car leur objectif était limité pour acquérir les compétences en programmation progressivement. Dans des projets plus importants, vous serez confrontés à des problèmes ou les lignes de programme vont s'accumuler…

L'approche efficace d'un problème complexe consiste souvent à le ***décomposer en plusieurs sous problèmes simples*** qui seront étudiés séparément. On va faire de même avec les programmes en ***créant des fonctions.***



(a) Évite la duplication de code.

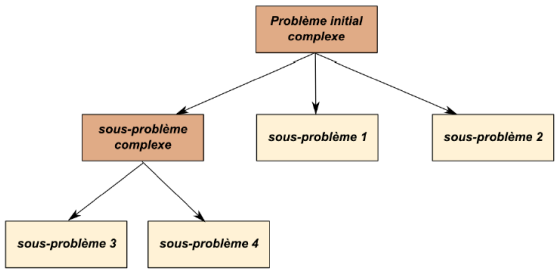
```
# util.py

def proportion(chaine, motif):
    """Fréquence de <motif> dans <chaine>."""

    n = len(chaine)
    k = chaine.count(motif)

    return k/n
```

(b) Met en relief entrées et sorties.



(d) Améliore la conception.

FIGURE 5.1 – Les avantages de l'utilisation des fonctions

☺ Une fonction est un ***ensemble d'instructions*** regroupées sous un nom et s'exécutant à la demande dans un script.

4. Syntaxe

La définition d'une fonction est composée :
✔ du mot clé « `def` » suivi de l'identificateur de la fonction, de parenthèses entourant les paramètres de la fonction séparés par des virgules, et du caractère ***« : »*** qui termine l'instruction,
✔ du bloc d'instructions indenté = ***corps de la fonction***,
✔ l'appel de la fonction dans le ***main.***

Voici une fonction qui renvoit le résultat de x^n.

```
#definition de la fonction

def puissance(x,n):
    puiss=x**n
    return puiss

# appels des paramètres
x=int(input("entrez la valeur de x:"))
n=int(input("entrez la valeur de n:"))

# main (affichage resultats)
print(x, "puissance", n, "=", puissance(x,n))
```

Les fonctions

5. Fonction simple sans paramètres

Pour notre 1ère approche, nous allons définir une fonction très simple qui renvoie l'affichage de la table de multiplication (0 à 10) de 7.

```
#definition de la fonction

def table7():
    n=1
    while n<11:
        print(n*7,end=" ")
        n=n+1

# main (affichage resultats)
table7()
```

Pour utiliser cette fonction et l'appeler dans un programme, il suffit de l'appeler par son nom, ainsi :

```
>>> table7()
```

Et provoque l'affichage suivant

```
7 14 21 28 35 42 49 56 63 70
```

6. Fonction simple avec paramètre

Et si maintenant nous voulions afficher les termes de la table de multiplication par 9 au lieu de 7 ? Nous allons pour cela définir une fonction qui sera capable ***d'afficher n'importe quelle table***.
Lors de l'appel de la fonction, il nous faudra saisir quelle table nous souhaitons afficher. Cette information que nous voulons transmettre s'appelle un ***argument***. Exemple : sin(a), la fonction sin( ) utilise la valeur de a comme argument.
Mais il faut prévoir une variable particulière pour recevoir l'argument transmis. Cette variable s'appelle un ***paramètre***.

```
#definition de la fonction

def table(base):
    n=1
    while n<11:
        print(n*base,end=" ")
        n=n+1

# appel du paramètre
base=int(input("de quel chiffre voulez vous afficher la table de multiplication ? "))

# main (affichage resultats)
table(base)
```

table( ) = fonction
base = paramètre
13 = argument

Pour tester cette fonction, nous l'appelons avec un argument :

```
>>> table(13)
13 26 39 52 65 78 91 104 117 130

>>> table(9)
9 18 27 36 45 54 63 72 81 90
```

Les fonctions

7. Utilisation d'une variable comme argument

L'argument que nous utilisons dans l'appel d'une fonction peut aussi être une variable. Dans l'exemple ci-dessous, l'argument que nous passons à la fonction table ( ) est le contenu de ***la variable a***. A l'intérieur de la fonction, cet argument est affecté au paramètre base, qui est une tout autre variable.

Exercice : Rajouter au programme les blocs d'instruction permettant d'afficher successivement les tables de 1 à 20.

```
#definition de la fonction

def table(base):
    n=1
    while n<11:
        print(n*base, end=" ")
        n=n+1

# main (affichage resultats: tt tables de multiplication de 1 à 20)
a=1
while a<20:
    table(a)
    a=a+1
    print()
```

8. Fonction avec plusieurs paramètres

La fonction table ( ) est certainement intéressante mais elle n'affiche que les 10 premiers termes de la table. Si on souhaite afficher plus ou moins de termes, on fait comment ?
Nous allons l'améliorer en ***ajoutant des paramètres***. La fonction s'appellera : `tableMulti().`

```
#definition de la fonction

def tableMulti(base, debut,fin):
    print("Fragment de la table de multiplication par",base,":")
    n=debut
    while n<=fin:
        print(n,"*",base,"=",n*base)
        n=n+1

#appels des parametres
base=int(input("entrez la table de multiplication voulue:"))
debut=int(input("entrez la valeur de debut:"))
fin=int(input("entrez la valeur de fin:"))


# main (affichage resultats)

tableMulti(base, debut, fin)
```

La fonction utilise ***3 paramètres*** : la base, l'indice du 1er terme à afficher, l'indice du dernier. Essayons en entrant par exemple :

```
>>>
entrez la table de multiplication voulue:8
entrez la valeur de debut:13
entrez la valeur de fin:17
Fragment de la table de multiplication par 8
13 * 8 = 104
14 * 8 = 112
15 * 8 = 120
16 * 8 = 128
17 * 8 = 136
>>>
```

9. Utilisation du retour: return

L'instruction `return` permet de « retourner » le résultat.

```
def le_plus_grand(a,b):
    if a>=b:
        max=a
    else:
        max=b
    return max

a=int(input("entrez a:"))
b=int(input("entrez b:"))

print(le_plus_grand(a,b))
```

```
def le_plus_grand(a,b):
    if a>=b:
        return a
    return b

a=int(input("entrez a:"))
b=int(input("entrez b:"))

print(le_plus_grand(a,b))
```

Attention au danger du print dans une fonction !!

```
def f(x):
        return x
a=2
if a>f(1):
        a=a+1
print(a)
```

Résultat: 3

```
def f(x):
        print(x)
a=2
if a>f(1):
        a=a+1
print(a)
```

Résultat: `TypeError: unorderable types: int() > NoneType()`

Exercice : Avec une fonction qui affiche plusieurs résultats, décoder une date rentrée en jjmmaa en jj / mm / aa.

```
#definition de la fonction

def decode_date(date):
    jour=date//10000
    mois=(date%10000)//100
    annee=date%100

    return jour,mois,annee

# main
date=int(input("entrez une date de type jjmmaa: "))
jour,mois,annee=decode_date(date)

# Affichage de la date
print(jour,"/",mois,"/",annee)
```

Attention ordre résultats !

```
date (jjmmaa) : 261191
26 / 11 / 91
```

10. Variables locales, variables globales

Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont ***accessibles qu'à la fonction*** elle-même = ***variables locales***. Ex : base, début, fin.

Chaque fois que la fonction `tableMulti()` est appelée, python réserve pour elle (dans mémoire PC) un nouvel espace de noms. Les contenus des variables base, début… sont inaccessibles de l'extérieur de la fonction :

```
>>> print(base)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'base' is not defined
```

Les variables définies à l'extérieur d'une fonction sont des ***variables globales***. Leur contenu est visible de l'intérieur de la fonction, mais la fonction ne peut pas le modifier.

Exemple : Que va afficher le programme suivant ?

```
#definition de la fonction

def mask():
    p=20
    print(p,q)

p,q=15,38

mask()
print(p,q)
```

p = variable locale

p,q = variables globales

```
>>>
20 38
15 38
```

On peut néanmoins modifier une variable locale pour la mettre en global via l'instruction : `global`.

```
>>> def monter():
...     global a
...     a = a+1
...     print(a)
...
>>> a = 15
>>> monter()
16
>>> monter()
17
>>>
```

La variable a est accessible et modifiable.

SAVOIR-FAIRE Du bon usage des variables globales

De façon générale, une bonne pratique consiste à utiliser les variables globales pour représenter les *constantes* du problème. En pratique, on ne devrait pas recourir souvent à la construction `global` de Python.

Comme pour les fonctions, il est préférable de donner aux variables globales des noms longs et explicites, ce qui les distinguera de fait des variables locales qui portent habituellement des noms courts (comme les paramètres formels).

11. Spécification des données d'une fonction

Aux concours, il peut vous être demandé de montrer ***comment spécifier les données*** attendues en entrée, et fournies en sortie/retour. Nous allons vous montrer comment ***accompagner vos fonctions d'une spécification***.

11.1. La spécification

Voici les étapes à suivre pour définir la spécification d'une fonction.

1. Définir l'objectif de la fonction
2. Identifier les paramètres formels. Pour chaque paramètre :
a. Identifier son rôle (description informelle)
b. Identifier son mode :
- entrée si utilisé par la fonction
- sortie si élaboré par la fonction
c. Choisir un nom qui synthétise le rôle
d. Choisir un type
3. En déduire un nom significatif pour la fonction (le nom caractérise le résultat).
4. Identifier les préconditions (conditions d'utilisation du sous-programme) et les post-conditions (effet du sous-programme sur ses paramètres).
5. Rédiger la spécification du sous-programme à partir des informations ci-dessus.

11.2. Exemple de la fonction pgcd (a,b)

Voici la spécification de la fonction pgcd (a,b) :

```
def pgcd(a, b):

 """Le pgcd de a et b.

    Préconditions:
        a > 0
        b > 0

    Post-conditions
        result le plus grand entier qui divise a et b

    :param a: premier entier
    :type a:  int
    :param b: deuxième entier
    :type b:  int
    :return:  le pgcd de a et b
    :rtype:   int

 """
```

12. Assertion

Lorsqu'on définit une nouvelle fonction, et qu'on la spécifie, il ***faut minimiser le nombre de préconditions si on veut la rendre robuste***, c'est-à-dire ***résistante à des mauvaises utilisations***. Par contre, si on écrit une fonction à usage interne, c'est moins critique, surtout si le nombre de personnes qui vont l'utiliser n'est pas trop élevé et qu'elles sont de confiance. Dans ce dernier cas, le code de la fonction sera plus simple.

On peut vouloir vérifier que des conditions qui sont censées être satisfaites le sont effectivement, à l'aide du ***mécanisme d'assertion proposé par Python***. Voyons comment l'utiliser pour vérifier les préconditions de la fonction `percentage` :

```
def percentage(score, total):
    assert total > 0, 'total doit être strictement positif'
    assert 0 <= score, 'score doit être positif'
    assert score <= total, 'score doit être inférieur à total'
    return score / total * 100
```

Trois instructions `assert` *ont été utilisées pour vérifier les préconditions. Une telle instruction se compose d'une condition (une expression booléenne) éventuellement suivie d'une virgule et d'une phrase en langue naturelle, sous forme d'une chaine de caractères. L'instruction* `assert` *teste si sa condition est satisfaite. Si c'est le cas, elle ne fait rien et sinon elle arrête immédiatement l'exécution du programme en affichant éventuellement la phrase qui lui est associée.*

Dans le programme d'exemple suivant, le premier appel s'exécutera sans erreur tandis que le second provoquera une erreur d'exécution due à une assertion pas satisfaite :

```
print(percentage(15, 20), '%')
print(percentage(22, 20), '%')
```

```
75.0 %
Traceback (most recent call last):
  File "program.py", line 8, in <module>
    print(percentage(22, 20), '%')
  File "program.py", line 4, in percentage
    assert score <= total, 'score doit être inférieur à total'
AssertionError: score doit être inférieur à total
```

Le mécanisme d'assertion est là pour ***empêcher des erreurs qui ne devraient pas se produire, en arrêtant prématurément le programme, avant d'exécuter le code*** qui aurait produit une erreur. Si une telle erreur survient, c'est que le programme doit être modifié pour qu'elle n'arrive plus. Dans cet exemple, on se rend immédiatement compte qu'un appel ne respectant pas les préconditions a été fait, et qu'il faut donc le changer. C'était peut-être 2/20 au lieu de 22/20... Enfin, il faut savoir que le mécanisme d'assertion est une aide au développeur, et ne doit en aucun cas faire partie du code fonctionnel d'un programme. En supprimant toutes les instructions `assert`, le programme doit continuer à fonctionner normalement.