Bioinformatics

Python Programming
Introduction to the Python Language

Sérgio Crisóstomo

2026

Departamento de Ciência de Computadores

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Contents

Why program?

What is computer programming?

- Implementation of computational methods to solve problems
- Analysis and comparison of different methods
- Combination of various skills:

  mathematics formal languages to specify processes;
  engineering combining components to form a system;
  evaluating pros/cons of alternatives
  natural sciences observing the behavior of complex systems;
  formulating hypotheses; testing predictions

Why learn to program?

- Scientific work requires data processing
- Facilitates the automation of repetitive tasks
- Many scientific applications are programmable (e.g., Excel, GNUplot, Matlab, Maple, Mathematica)
- Structures thinking for problem-solving
- Develops analytical thinking
- It is an intellectual challenge
- It is fun!

Why learn to program? (cont.)

Programming develops problem-solving skills:

- the ability to describe problems rigorously;
- creative thinking for possible solutions;
- expressing solutions clearly and precisely.

Programming Languages

Programming Languages

- Formal languages to express computation

  syntax: rules for forming sentences (grammar)

  semantics: meaning associated with each sentence

- Other examples of languages: arithmetic expressions, chemical symbols

	syntax	semantics
$3 \times (1 + 2)$	ok	9
$3 \times 1 + 2$	ok	5
$\times)1 + 2 + (3$	error	—
H_2O	ok	water
${}_2zZ$	error	—

Machine code

```
55 89 e5 83 ec 20 83 7d 0c 00 75 0f ...
```

- Numeric codes associated with basic operations
- Language specific to each microprocessor
- The only language directly executable by the computer
- Difficult to write programs directly in machine code
- Designed to facilitate implementation using electronic circuits

Assembly Language

```
55                  push   %ebp
89 e5               mov    %esp,%ebp
83 ec 20            sub    $0x20,%esp
83 7d 0c 00         cmpl   $0x0,0xc(%ebp)
75 0f               jne    1b
...                 ...
```

- Representation of machine code using mnemonics
- More readable than machine language
- Can be automatically translated into machine code
- Still specific to each microprocessor
- Very Low-level: requires slow, tedious, and error-prone programming
- Used only in very specific contexts

High-level languages

Example: calculate the factorial of 10 in Python.

```
n = 10
p = 1
for i in range(2, n+1):
  p = p*i
print("factorial of ", n, " = ", p)
```

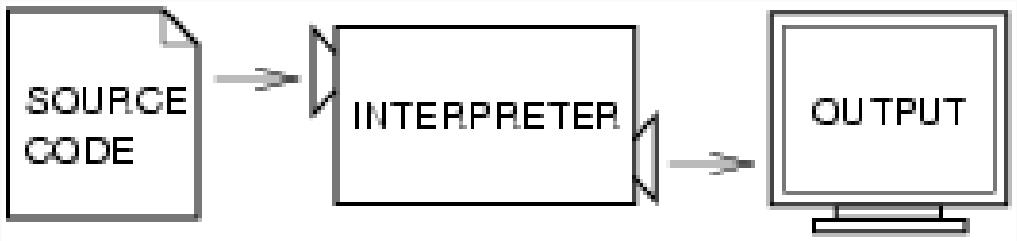
- Closer to the mathematical formulation of problems
- Allow faster program development and make it easier to detect and correct errors
- Allow the development of portable programs e.g., independent of the specific processor in each computer

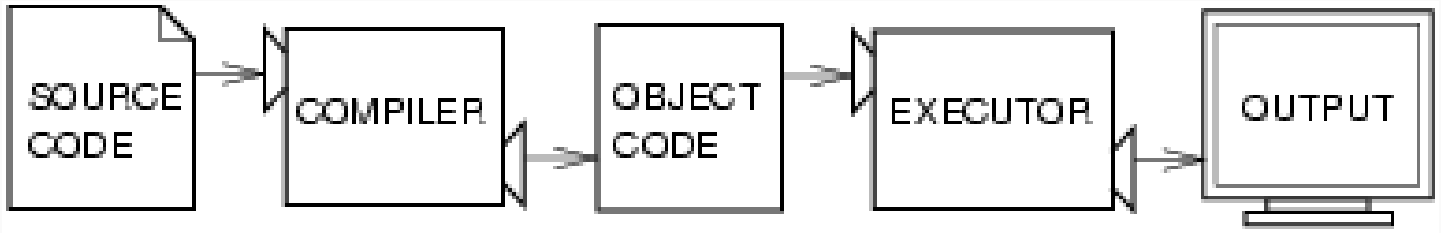
Interpreters vs. Compilers

High-level languages are translated into machine code by special programs:

interpreter the translation is performed each time the program is run

compiler the translation is performed only once





Interpreters vs. Compilers (cont.)

Advantages of interpreters:

- allow quick interactive use
- make it easier to test fragments of programs
- are simpler to implement

Advantages of compilers:

- allow generating more efficient machine code;
- generate independent programs
  (the compiler does not need to be present during execution)

Timeline of Some Programming Languages

1956	Fortran I
1958	Lisp
1960	Cobol, Algol 60
1970	Pascal
1976	Fortran 77
1978	C (K&R)
1980	Smalltalk 80
1984	Common Lisp, C++
1986	Perl
1990	Fortran 90, Python, Haskell
1995	Java, JavaScript
2000	Python 2, C#
2008	Python 3

Why so many languages?

- Different levels of abstraction:

  highest level: closer to the problem formulation; facilitates programming, detection, and correction of errors

  lowest level: closer to the machine; potentially more efficient

- Different types of problems:

  numerical computation: Fortran, C, C++, Python

  operating systems: C, C++

  critical systems: Ada, C, C++

  web systems: Java, JavaScript, Python

Why so many languages? (cont.)

- Different paradigms:

  imperative: Algol, Pascal, C

  functional: Lisp, Scheme, ML, OCaml, Haskell

  logic: Prolog

  object-oriented: Smalltalk, C++, Java, C#

- Subjective preferences (style, elegance, readability)

The Python Language

The Python Language

- High-level language
- Simple syntax, easy to learn
- Can be used on Windows, Linux, FreeBSD, Mac OS, etc...
- Standard implementation distributed as open-source code
- Supports procedural and object-oriented programming
- Many libraries available
- Widely used: Google, Microsoft, Dropbox, NASA, Lawrence Livermore Labs, Industrial Light & Magic...
- Official site: `http://www.python.org`

The Python Language

Implemented with a hybrid interpreter:

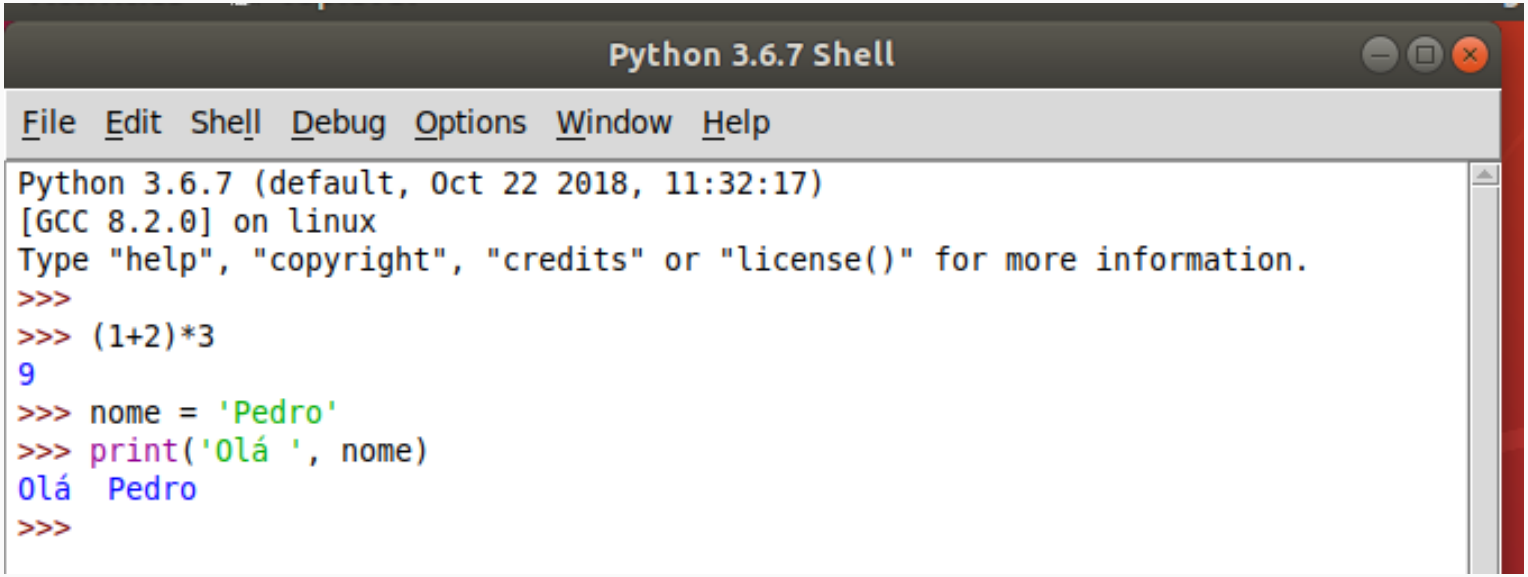
- the program is translated into a “pseudo” machine code (byte-code)
- the byte-code is executed by an interpreter

Advantages:
- easy to use interactively
- easy to test and modify components
- more efficient than a classic interpreter

Disadvantage: not as efficient as a compiled language (e.g., C, C++, or Fortran)

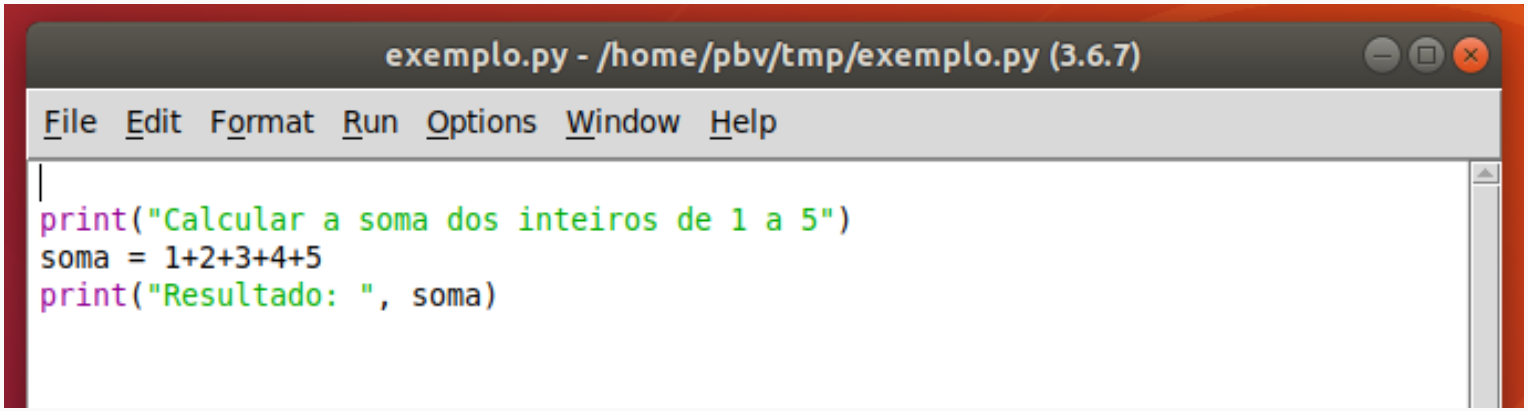
Interactive Usage

Running the Python interpreter (IDLE), we can write commands and immediately see the results.



Using a Script

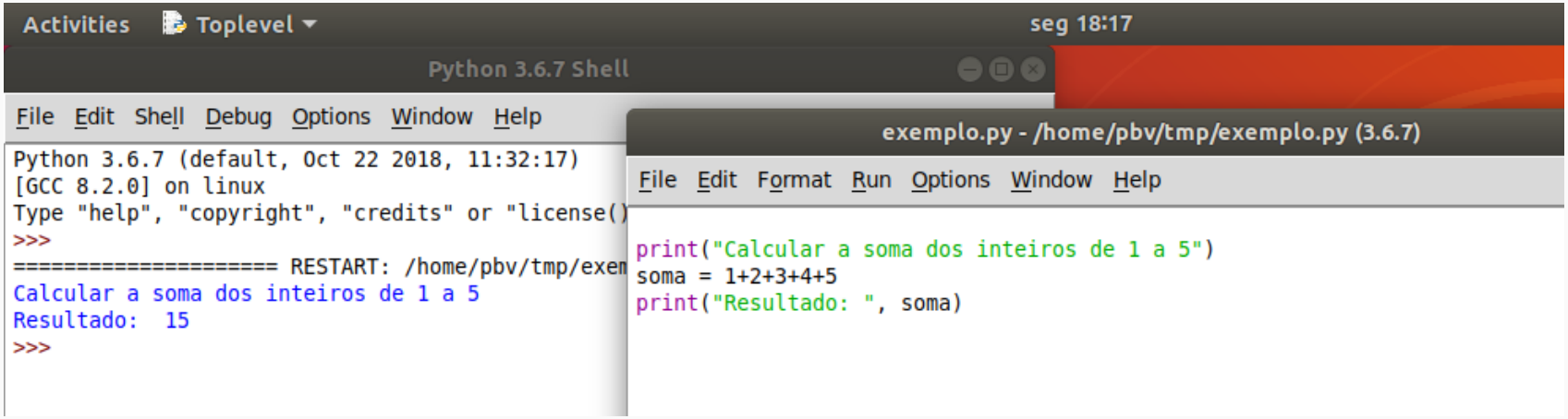
Alternatively, we can write a program in a text file (“script”) and execute it all at once.



Convention: Python programs have the extension `.py`

Using a Script (cont.)

- The interactive form is used to test small code snippets
- To write programs with more than a few lines, we should edit a "script"
- The IDLE development environment combines:
  - an interactive command window (“Python shell”)
  - one or more windows for files (“scripts”)



Using Python as a Calculator

- Basic arithmetic operators:

  addition and subtraction `+` `-`
  multiplication and division `*` `/`
  exponentiation `**`
  parentheses `(` `)`
- Integer and fractional numbers: `42` `-7` `3.1416`
- Incorrect expressions: `SyntaxError`

Using Python as a Calculator (cont.)

Operator precedence (order of calculation):

1. parentheses `(` `)`
2. exponentiation `**`
3. multiplication and division `*` `/`
4. addition and subtraction `+` `-`

Operators with the same precedence group from left to right.

Examples:

```
>>> 1+2-3+4
4
>>> 1+2-(3+4)
-4
```

```
>>> 2**3*4+1
33
>>> 2**3*(4+1)
40
```

Mathematical Functions

Many mathematical functions and constants are available in the *math* module.

To use it, we must start by importing the module.

```
>>> import math
```

Function names start with the prefix "`math`":

```
>>> math.sqrt(2)
1.414213562373095 1
>>> math.pi
3.141592653589793
```

Mathematical Functions (cont.)

Some functions and constants from the *math* module:

square root	`sqrt`
trigonometric functions	`sin`, `cos`, `tan`
inverse trigonometric functions	`asin`, `acos`, `atan`, `atan2`
exponential and logarithms	`exp`, `log`, `log10`
e, π	`e`, `pi`

For more information:

```
>>> help(math)
>>> help(math.log)
```

general information

specific about a function

Types

Values are classified into different types.

Some operations are only possible with certain types:

```
>>> "Hello world!" + 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str
```

Basic Types

	type	examples
integers	`int`	`1  -33  29`
floating-point	`float`	`1.0  -0.025  3.14156`
text strings	`str`	`"Hello world!"  'ABC'  '1.23.99'`

Type of a Result

In the interpreter, we can use `type(...)` to obtain the type of the result of an expression:

```
>>> (1+2+3)*5-1
29

>>> type((1+2+3)*5-1)
<class 'int'>

>>> type(1.234)
<class 'float'>

>>> type('ABC')
<class 'str'>
```

Basic Types

Types of Numbers

In *Python* we distinguish between integers and rationals (floating-point numbers), and complex numbers, associating them with different types.

	type	examples
integers	`int`	`42  -7`
floating-point	`float`	`42.0  -7.0  -0.0254`
complex numbers	`complex`	`3+4j  -2j  1.5-0.5j`

Types of Numbers (cont.)

Arithmetic operations work with both types:

```
>>> 1+2
3
```
int + int => int

```
>>> 1.0+2.0
3.0
```
float + float => float

We can also mix integers and *floats* in an operation; the result will be a *float*:

```
>>> 1 + 2.5
3.5
```
int + float => float

Types of Numbers (cont.)

Division between integers results in a *float*:

```
>>> 17/5
3.4
```

We can obtain the *quotient* and *remainder* of integer division using the `//` and `%` operators:

```
>>> 17//5
3
>>> 17%5
2
```

quotient of integer division

remainder of integer division

Rounding Errors

Integers can be represented exactly in the computer.[1]

Floating-point numbers are finite approximations of real numbers:

```
>>> 8/3
2.6666666666666665
```

Successive operations on these numbers can accumulate rounding errors.

The control of these errors in computation is studied in Numerical Analysis.

[1] Only limited by available memory.

Rounding Errors

Using exact algebra:

$$\left(\frac{100}{3} - 33\right) \times 3 = 100 - 33 \times 3 = 1$$

However, using floating-point operations we obtain different results:

```
>>> (100.0/3.0 - 33.0) * 3.0
1.0000000000000007
>>> 100.0 - 33.0*3.0
1.0
```

The rounding error was

$$1.0000000000000007 - 1 \approx 7 \times 10^{-15}$$

Automatic Conversion Between Numeric Types

$$\texttt{int} + \texttt{int} \Rightarrow \texttt{int}$$

$$\texttt{float} + \texttt{float} \Rightarrow \texttt{float}$$

$$\texttt{int} + \texttt{float} \Rightarrow \texttt{float}$$

$$\texttt{float} + \texttt{int} \Rightarrow \texttt{float}$$

Also with the arithmetic operators $-$, $*$, and $**$.

Division (in *python 3*) is a special case:

$$\texttt{int/int} \Rightarrow \texttt{float}$$

$$\texttt{int//int} \Rightarrow \texttt{int}$$

$$\texttt{int\%int} \Rightarrow \texttt{int}$$

Explicit Type Conversion

```
>>> int(2.71)
2

>>> round(2.71)
3

>>> float(-33)
-33.0
```

```
>>> str(-3.134)
'-3.134'

>>> float("3.14")
3.14

>>> float("thirty-three")
ValueError
```

Note:

`round(...)` rounds to the nearest integer;

`int(...)` truncates the fractional part;

Strings

Strings are values of type `str` (*string*).

We write the text between single or double quotes:

```
>>> "Hello, world!"
'Hello, world!'

>>> 'abracadabra'
'abracadabra'

>>> type('abracadabra')
<class 'str'>
```

Strings (cont.)

We can use triple quotes to introduce multi-line strings.

```
>>> '''Good morning!
--- Hello, world!'''
'Good morning!\n--- Hello, world!'
```

Operations on Strings

Concatenation `str + str ⇒ str`

Repetition `int * str ⇒ str`

```
>>> "Hello" + " " + "World"
'Hello World'

>>> 3*"Hello" + " World!"
'HelloHelloHello World!'

>>> 3*"Hello " + "World!"
'Hello Hello Hello World!
```

Variables and Assignments

Variables

- Names to represent *quantities* or *properties* of a problem
- Start with a letter, followed by letters, numbers, or underscore
- May have letters with accents
- Cannot contain spaces or tabs
- Cannot be *reserved words*:

and	def	exec	if	not	return
assert	del	finally	import	or	try
break	elif	for	in	pass	while
class	else	from	is	print	yield
continue	except	global	lambda	raise	

Variables (cont.)

Examples of valid variable names:

```
name   age   Max_Price   area2
```

Examples of names we cannot use:

```
76trombones   more$   lambda
```

Assignments

Assigns the value of an expression to a variable:

$$name = expression$$

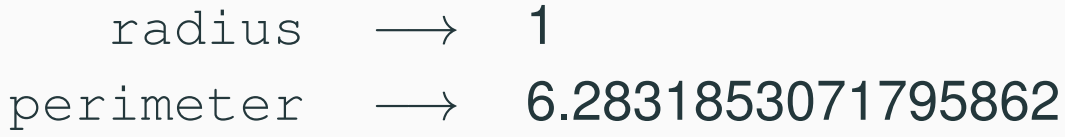
```
>>> radius = 1
```

$$\texttt{radius} \longrightarrow 1$$

Assignments (cont.)

After defining a variable, we can use it in subsequent expressions:

```
>>> import math
>>> perimeter = 2*math.pi*radius
>>> perimeter
6.2831853071795862
```



Assignments (cont.)

Note that assignment is a command, not an equation.

Example:

```
>>> radius = 2
>>> perimeter
6.283185307179586 2
```

`radius`	$\longrightarrow$	2
`perimeter`	$\longrightarrow$	6.283185307179586

The `perimeter` does not automatically change when we change `radius`...

Assignments (cont.)

If we want to recalculate the perimeter, we must execute the assignment again:

```
>>> perimeter = 2*math.pi*radius
>>> perimeter
12.566370614359172
```

`radius`	$\longrightarrow$	2
`perimeter`	$\longrightarrow$	12.566370614359172

Order of Assignments

The order of assignments is important!

Example: let's annotate the values of *p* and *n* after each instruction.

Program 1

```
p = 1
n = 2
p = p*n
n = n+1
```

Program 2

```
p = 1
n = 2
n = n+1
p = p*n
```

Order of Assignments

The order of assignments is important!

Example: let's annotate the values of *p* and *n* after each instruction.

Program 1

```
p = 1
n = 2
p = p*n
n = n+1
```

Program 2

```
p = 1
n = 2
n = n+1
p = p*n
```

Order of Assignments

The order of assignments is important!

Example: let's annotate the values of p and n after each instruction.

Program 1

```
p = 1        p → 1
n = 2
p = p*n
n = n+1
```

Program 2

```
p = 1        p → 1
n = 2
n = n+1
p = p*n
```

Order of Assignments

The order of assignments is important!

Example: let's annotate the values of p and n after each instruction.

Program 1

```
p = 1        # p → 1
n = 2        # p → 1  n → 2
p = p*n
n = n+1
```

Program 2

```
p = 1        # p → 1
n = 2        # p → 1  n → 2
n = n+1
p = p*n
```

Order of Assignments

The order of assignments is important!

Example: let's annotate the values of p and n after each instruction.

Program 1

`p = 1`	$p \to 1$
`n = 2`	$p \to 1 \quad n \to 2$
`p = p*n`	$p \to 2 \quad n \to 2$
`n = n+1`	

Program 2

`p = 1`	$p \to 1$
`n = 2`	$p \to 1 \quad n \to 2$
`n = n+1`	$p \to 1 \quad n \to 3$
`p = p*n`	

Order of Assignments

The order of assignments is important!

Example: let's annotate the values of p and n after each instruction.

Program 1

`p = 1`	$p \to 1$
`n = 2`	$p \to 1 \quad n \to 2$
`p = p*n`	$p \to 2 \quad n \to 2$
`n = n+1`	$p \to 2 \quad n \to 3$

Program 2

`p = 1`	$p \to 1$
`n = 2`	$p \to 1 \quad n \to 2$
`n = n+1`	$p \to 1 \quad n \to 3$
`p = p*n`	$p \to 3 \quad n \to 3$

Order of Assignments

The order of assignments is important!

Example: let's annotate the values of p and n after each instruction.

Program 1

`p = 1`	$p \to 1$
`n = 2`	$p \to 1 \quad n \to 2$
`p = p*n`	$p \to 2 \quad n \to 2$
`n = n+1`	$p \to 2 \quad n \to 3$

Final: $p \to 2 \quad n \to 3$

Program 2

`p = 1`	$p \to 1$
`n = 2`	$p \to 1 \quad n \to 2$
`n = n+1`	$p \to 1 \quad n \to 3$
`p = p*n`	$p \to 3 \quad n \to 3$

Final: $p \to 3 \quad n \to 3$

Complete Programs

Complete Programs

perimetro.py

```
#Calculate the perimeter of a circumference

import math

radius = 2.5
perimeter = 2*math.pi*radius
```

It runs correctly but does not show any results.

Commands for Input and Output

input(text) writes text (optional) and reads a string from the terminal (keyboard);

print(expr1, expr2, ...) writes results to the terminal

Revised Program

perimetro.py

```
#Calculate the perimeter of a circumference
import math

print("Enter the value of the radius:")
radius = float(input())
perimeter = 2*math.pi*radius
print("Perimeter of the circle:", perimeter)
```

Comments

```
# Calculate the perimeter of a circumference
```

- Start with the symbol `#` and extend to the end of the line
- Allow documentation for other programmers
- Also useful for the author (to remember how the program works)

Comments (cont.)

perimetro.py

```
# Calculate the perimeter of a circle

print("Enter the value of the radius:")

# read a string and convert to a number
radius = float(input())
perimeter = 2*math.pi*radius    # calculate the perimeter

# print the result
print("Perimeter of the circle:", perimeter)
```

Definition of Procedures and Functions

Definition of Procedures and Functions

- Previously we saw some predefined functions in the `math` module.
- We will now see how to define new procedures and functions.

Structured Programming
Decomposing a problem into simpler procedures until reaching elementary operations.

Definition of Procedures

Definitions

```
def name(parameter list):
    first instruction
    second instruction
    ⋮
    final instruction
```

- The *parameter list* can be empty.
- The *body* consists of one or more instructions
  - delimited by indentation.
  - aligned in the same column.

Example

```
def refrain():
    print("If one elephant bothers a lot of people,")
    print("Two elephants bother much more.")
```

Trying it in the interpreter:

```
>>> refrain()
If one elephant bothers a lot of people,
Two elephants bother much more.
```

Observations

- A procedure is a recipe for a computation.
- The definition of a procedure does not execute the instructions (it only registers the procedure).

```
def refrain():    # procedure definition
    ⋮
```

- The instructions are executed when we invoke the procedure.

```
>>> refrain()     # invoke the procedure
```

- We can invoke a procedure inside another; for example:

```
def repeat():
    refrain()    # first invocation
    refrain()    # second invocation
```

Parameters and Values

Let's generalize the previous procedure.

```
def refrain(n):
    print("If", n, "elephants bother a lot of people,")
    print(n+1, "elephants bother much more.")
```

The variable *n* is a parameter of the procedure.

To invoke the procedure, we must specify the value of *n*:

```
>>> refrain(3)
If 3 elephants bother a lot of people,
4 elephants bother much more.
```

Observations

When invoking the procedure, we must use the correct number of arguments.

```
>>> refrain()
TypeError: refrain() missing 1 required positional
argument: 'n'
>>> refrain(2,3)
TypeError: refrain() takes 1 positional argument
but 2 were given
```

Observations (cont.)

Consider this variation of the previous procedure:

```
def refrain(n):
    print("If n elephants bother a lot of people,")
    print("n+1 elephants bother much more.")
```

(Note the placement of the quotes.)

What happens when we run it?

```
>>> refrain(2)
```

(Try it in the interpreter!)

Definition of Functions

Defining Functions

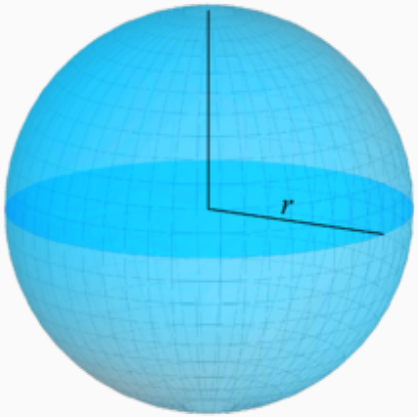
```
def function(arg1, arg2, ...):
    ⋮
    return result
```

- A function is a procedure that returns a result.
- The last instruction of a function must be `return`.
- The `return` statement can also be used in the middle of the body (to terminate the function).

Example: calculating the volume of a sphere

Volume V of a sphere of radius r.

$$V = \frac{4}{3}\pi r^3$$




```
import math
def volume(r):
    V = 4/3 * math.pi * r**3
    return V
```

Example: calculating the volume of a sphere (cont.)

Examples of usage:

```
>>> volume(1.0)
4.1887902047863905
>>> volume(1.5)
14.137166941154067
>>> volume(2.0)
33.510321638291124
```

Alternative Definition

```
import math
def volume(r):
    return 4/3 * math.pi * r**3
```

- We can return the value of an *expression* directly.
- It is not necessary to use the auxiliary variable V.
- However: using auxiliary variables can help document the program.

Return or Print?

```
def volume(r):
    V= 4/3*math.pi*r**3
    return V
```

```
>>> volume(1)+volume(2)
37.69911184307752
```

```
def volume(r):
    V= 4/3*math.pi*r**3
    print(V)
```

```
>>> volume(1)+volume(2)
4.188790204786390 5
33.510321638291124
TypeError:...
```

- The definition using print does not return the result.
- It is preferable to define procedures that return results.
- This allows the result to be used for other purposes (besides printing).

Variable Scope

- The parameters of a procedure are local variables.
  - They are associated with values only within the definition.
  - We don't have to worry if the same name is used in another context.
- The auxiliary variables defined within procedures are also local.

```
>>> r = 42
>>> volume(1)
4.1887902047863905
>>> r
42
>>> V
NameError: name 'V' is not defined
```

Documentation

We can document procedures and functions using

1. comments
2. *docstrings*

```
import math    # use mathematical definitions
def volume(r):
    "Calculates the volume of a sphere given radius r."
    V= 4/3*math.pi*r**3
    return V
```

Documentation (cont.)

- *Docstrings* are used by Python's documentation system.
- We use `help` to ask for help about a module or procedure:

  `>>> help(`*name*`)`
- Adding comments and *docstrings* helps the programmer understand a program or library.

```
>>> help(volume)
Help on function volume in module __main__:

volume(r)
    Calculates the volume of a sphere with radius r.
```

Type Annotations

- The arguments of procedures and functions must respect correct types.
- Example: the argument of the `refrain` procedure must be an integer (the number *n* of elephants).
- We can document this information using a type annotation.

```
def refrain(n: int):
    print("If", n, "elephants bother a lot of people,")
    print(n+1, "elephants bother much more.")
```

Type Annotations (cont.)

For functions, we annotate the types of the arguments and the result.

```
def volume(r: float) -> float:
    V= 4/3*math.pi*r**3
    return V
```

More generally:

```
def function(x: type1, y: type2, ...) -> typeR:
    ⋮
    return result
```

Type Annotations (cont.)

- Type annotations are optional in Python.
- Programs run without any annotations.
- However: annotations make it easier to detect logical errors in programs.
- The automatic testing system uses them to provide more informative error messages.