Scientific Computing 272

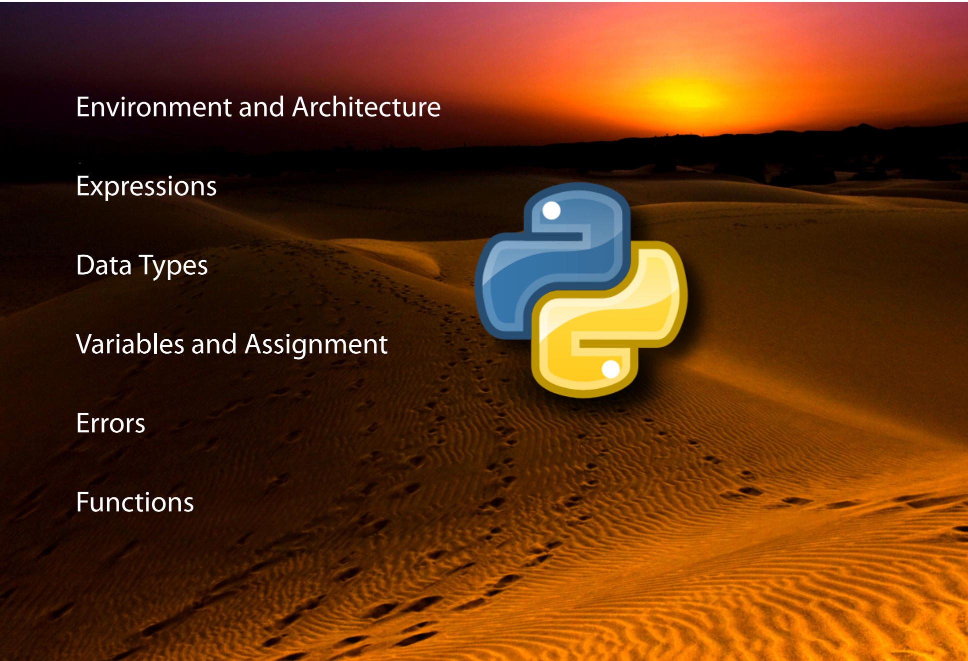
Section 1: Introduction to Programming with Python

Last updated: 22 February 2019

UNIVERSITEIT STELLENBOSCH UNIVERSITY

Willem Bester

Section Outline



Environment and Architecture

Expressions

Data Types

Variables and Assignment

Errors

Functions

The “Big Picture”

- A computer is assembled from **hardware**:
  - A **processor** that can do arithmetic and execute instructions
  - Storage such as a **hard disk** or **flash drive**
  - Input/output devices such as the monitor, keyboard, and network card
- The **operating system** (OS) is the only piece of **software** that is allowed direct access to the hardware
- **Application programs** either run on a specific OS
- Or they run on a **virtual machine** or through an **interpreter**
- Python is an interpreted language

Expressions

- Python commands are called **statements**
- An **expression** is a certain kind of statement
- Mathematical expressions, like $4 + 9$ and $11 \times 2 - 5$, consist of
  - **values**, like 4 and 11
  - **operators**, like $+$ and $\times$
- The values on which an operator acts are called its **operands**
- An operator combines its operands to give a result
- Python can **evaluate** mathematical expressions

The Python Interpreter

Example (Addition and Subtraction)

```
>>> 4 + 9
13
>>> 4 - 9
-5
```

The Prompt

- `>>>` is called a **prompt**
- It prompts the user to type something
- We do not type it in; Python displays it

Arithmetic Operators

Table: Arithmetic Operators

Operator	Operation	Example	Result
`-`	Negation	`-5`	`-5`
`+`	Addition	`3 + 4`	`7`
`-`	Subtraction	`3 - 4`	`-1`
`*`	Multiplication	`5 * 2`	`10`
`/`	(True) Division	`5 / 2`	`2.5`
`//`	(Floor) Division	`5 // 2`	`2`
`%`	Remainder (Modulo)	`8 % 5`	`3`
`**`	Exponentiation	`2 ** 5`	`32`

- Note the operators for the bottom five operations
- Remainder is often also called **modulo**

Data Types

- Every value in Python has a particular **type**
- The combination of type and operator determines how values behave when operated on
- Does the following make sense?

Example (Integer Division)

```
>>> 8 / 5
1.6
>>> 8 // 5
1
```

- We know $8 \div 5 = 1\frac{3}{5} = 1.6$
- So is Python mad when doing “floor” divison?

Integers

- Integers are the "whole" numbers, denoted by $\mathbb{Z}$:
  $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$
- In Python, we say they are of type `int`
- In algebraic terms, the integers form an integral domain
- This means addition, subtraction, and multiplication of integers yields another integer
- Division is not closed over the integers
- Therefore, for $x, y \in \mathbb{Z}$, Python defines $x \mathbin{//} y = \lfloor x/y \rfloor$
- The **floor** function, denoted by $\lfloor\ \rfloor$, gives the largest integer not greater than a given real number
- So, floor division where both operands are integers yields another integer

Integer Division

Example (Integer Division)

```
>>> 8 // 5
1
>>> -8 // 5
-2
>>> 8 // -5
-2
```

- Do those expressions with negative operands make sense?
- Yes! Why?

$$\lfloor -8/5 \rfloor = \lfloor 8/-5 \rfloor = \lfloor -1.6 \rfloor = -2$$

Integer Modulo

For positive operands, the modulo operator % gives the remainder for division of the first operand by the second.
Python defines

$$x \,\%\, y = x - \left\lfloor \frac{x}{y} \right\rfloor y.$$

For positive operands, this behaves as we expect: The remainder of 19 after division by 5 is

$$19 - \lfloor 19/5 \rfloor 5 = 19 - (3)(5) = 19 - 15 = 4.$$

Example (Integer Modulo)

```
>>> 19 % 5
4
```

Integer Modulo with Negative Operands

For negative operands, the modulo operation is also **well-defined**:

- The definition is not ambiguous
- For two particular operand values, there is a unique result

Example (Integer Modulo with Negative Operands)

```
>>> -19 % -5
-4
```

Why?

$$-19 - \lfloor -19 / -5 \rfloor (-5) = -19 - (3)(-5) = -19 + 15 = -4$$

Integer Modulo with Negative Operands

Sign of a Modulo Result

When using modulo, the sign of the result matches the sign of the second operand.

Example (More Negative Integer Modulo)

```
>>> 19 % -5
-1
>>> -19 % 5
1
```

- Why?

$$19 - \lfloor 19/-5 \rfloor(-5) = 19 - (-4)(-5) = 19 - 20 = -1$$
$$-19 - \lfloor -19/5 \rfloor(5) = -19 - (-4)(5) = -19 + 20 = 1$$

Floating-Point Numbers

Python has type `float` to represent numbers with fractional parts. **Floating point** comes from the scientific notation, and refers to the decimal point that moves between the digits of the numbers.

Example (Scientific Notation in Python)

The number $0.0013 = 1.3 \times 10^{-3}$ can be written as `1.3e-3`:

```
>>> 1.3e-3
0.0013
```

Python sees a number with a decimal point as a `float`.

Example (Floating-Point Division)

```
>>> 13 // 10
1
>>> 13.0 // 10
1.0
```

Automatic Conversion

Automatic Type Conversion

If one operand in an expression is a `float`, and the other an `int`, Python automatically **converts** the `int` to a `float`.

Example (Automatic Conversion)

```
>>> 13 + 10.0
23.0
>>> 13.0 - 10
3.0
```

The following shortcut is possible, but some consider it bad style.

```
    13. - 10
```

Floating-Point Modulo

Modulo may also be computed for `float`s. The definition is the same as for `int`s.

Example (Floating-Point Modulo)

```
>>> 8.5 % 3.5
1.5
```

Why?

$$\begin{aligned} 8.5 - \lfloor 8.5/3.5 \rfloor (3.5) &= 8.5 - \lfloor 2.\overline{428571} \rfloor (3.5) \\ &= 8.5 - (2)(3.5) = 8.5 - 7 = 1.5 \end{aligned}$$

Finite Precision

Floating-point numbers are not exactly the same as the real numbers you are used to working with on paper.

Example (Finite Precision)

```
>>> 1 / 3
0.3333333333333333
```

But, wait:

$$0.3333333333333333 = \frac{3\,333\,333\,333\,333\,333}{10\,000\,000\,000\,000\,000} \neq \frac{1}{3} = 0.\dot{3}$$

Computers do not have unlimited memory. To keep things simple and fast, each `float` occupies a finite number of memory cells, limiting the information for each number. 0.3333333333333333 is the closest to $\frac{1}{3}$ the computer can actually store under this scheme.

Finite Precision

The effects of finite precision shows up in other ways as well.

Example (Distributivity of multiplication over addition)

```
>>> 10 * 0.1 + 10 * 0.2
3.0
>>> 10 * (0.1 + 0.2)
3.0000000000000004
```

Does this mean we can’t trust Python?

Not quite. It just means we have to be careful in the same way we are careful when, for example, we multiply large with small quantities in Physics.

Operator Precedence

Example (Convert Fahrenheit to Celsius)

To convert Fahrenheit to Celsius, subtract 32 from the temperature in Fahrenheit and then multiply by $\frac{5}{9}$. So, for 212 °F:

```
>>> 212 - 32 * 5 / 9
194.22222222222223
```

- This should be 100
- The problem is that `*` and `/` have higher **precedence** than `-`
- An operator with higher precedence is evaluated before one with lower precedence
- What was calculated here is $212 - ((32 \times 5)/9)$

Operator Precedence

In mathematics, we use parentheses to change the precedence of operators. We can do the same in Python.

Example (Correct Conversion of Fahrenheit to Celsius)

Use parentheses to group parts of an expression.

```
>>> (212 - 32) * 5 / 9
100.0
```

Example (Parenthesizing for Clarity)

We sometimes parenthesise complicated expressions for clarity. To show that we think of the ratio $\frac{5}{9}$, we may parenthesise thus:

```
>>> (212 - 32) * (5 / 9)
100.0
```

Operator Precedence and Associativity

Table: Arithmetic Operators by Precedence

Operator	Operation
`**`	Exponentiation
`-`	Negation
`*`, `/`, `//`, `%`	Multiplication, divisions, and modulo
`+`, `-`	Addition and subtraction

- Note that exponentiation is **right-associative**; this means `2 ** 3 ** 4` is evaluated as 2^{3^4}, **and not** $(2^3)^4$
- The other operators are **left-associative**; for example, `2 - 3 - 4` is evaluated as $(2-3)-4$, **and not** $2-(3-4)$
- Associativity is important when we have more than one operator of the same level of precedence

Operator Precedence and Associativity

Example (Operator Precedence)

```
>>> 2 * 3 ** 4
162
>>> 2 ** 3 * 4
32
>>> 2 ** (3 * 4)
4096
```

Example (Operator Associativity)

```
>>> 2 ** 3 ** 4
2417851639229258349412352L
>>> (2 ** 3) ** 4
4096
```

Variables

- In mathematics, a **variable** is a symbol that represents an arbitrary element of some set
- In Python, a **variable** is a name with an associated value
- Variable names must start with a letter or the underscore symbol; thereafter it may contain letters, underscores, and digits

Example (Variable Names)

The variable names `x`, `degrees_celsius`, and `parent1` are allowed. However, `777` and `degrees-celsius` is not; the former starts with a digit and is a number, and the latter contains punctuation, which is interpreted as an operator.

Variable Assignment

- We **define** a new variable by **assigning** it a value
- Read the statement `x = 7` as "*x* gets (the value) 7"
- This kind of statement is called an **assignment**, and is executed thus:
  1. Evaluate the expression on the right of `=`
  2. Store this result value in the variable on the left of `=`
- Typing an already defined variable name on its own makes Python display its value

Example (Assignment)

```
>>> x = 7
>>> x
7
```

Variables in Expressions

- When a variable is written in an expression, Python uses its (current) value in the calculation
- You must define a variable before using it
- If you do not, you will get an error; see later

Example

```
>>> x = 7
>>> 100 - x
93
>>> 2 * x - 143.0
-129.0
```

Variables

We can create new variables from existing ones.

Example

```
>>> x = 13
>>> y = 100 - x
>>> y
87
```

A variable name on its own is just an expression without any operators, just as Python will display the value of a number.

Example

```
>>> 107
107
```

Variable Values

Variables are variable in that their values can change during execution. Results of calculations before a variable was changed **do not also change**.

Example

```
>>> z = 3
>>> double = 2 * z
>>> double
6
>>> z = 17
>>> double
6
```

Why? Once a value is associated with a `double`, it stays associated with `double` until it is explicitly overwritten by the program

Variables

We may use a variable on both sides of the assignment operator =.

Example

```
>>> x = 19
>>> x = 2 * x
>>> x
38
>>> x = x * x
>>> x
1444
```

Would these Python statements make sense in mathematics?

Assignment and Equality

Assignment v. Equality

- In mathematics, = means “is equal to”; it states a fact
- In Python, `=` means “is assigned the value”; it is an operation

A statement with the same variable on both sides of the assignment operator `=` is evaluated thus:

1. Get the value currently associated with the variable
2. Calculate the expression on the right of `=`
3. Assign the result to the variable

Actually, this is exactly the same as previously.

Combined Operators

- Since we frequently have the same variable on both sides of `=`, Python provides a shorthand notation
- E.g., instead of writing `x = x * x`, we write `x *= x`
- A **combined operator** is evaluated thus:
    1. Evaluate the expression on the right of `=`
    2. Apply the operator attached to `=` to the variable on the left and the result of the expression
    3. Assign the result to the variable on the left
- The combined form is available for all the arithmetic operators

Combined Operators

Example

```
>>> number = 100
>>> number -= 90
>>> number
10
>>> number *= 2 + 3
>>> number
50
>>> number *= number
>>> number
2500
```

Errors

What happens if we try to use a variable that we have not defined yet (by assigning it a value)?

Example (Error: Undefined Name)

```
>>> 3 * temperature
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'temperature' is not defined
```

Python's response is not too cryptic; the last line gives

- the type of error: NameError, a problem with a name
- a description of the error: in this case, the (variable) name has not been defined yet

Syntax

- Python is governed by its **syntax**—the rules specifying what is legal, how statements and expressions are structured, etc.
- What happens if we break the syntax rules?

Example (Syntax Error)

```
>>> 3 *
  File "<stdin>", line 1
    3 *
      ^
SyntaxError: invalid syntax
```

Functions

- A mathematician would write the general procedure for converting Fahrenheit to Celsius as

$$f(t) = \frac{5}{9}(t - 32),$$

  where t is the temperature in Fahrenheit
- To convert 80 °F to Celsius, we substitute $t = 80$ into the equation:

$$f(80) = \frac{5}{9}(80 - 32) = 26\frac{2}{3}$$

- A **function** f from S to T, written $f : S \to T$, where S and T are non-empty sets, associates with each element in the **domain** S a unique element in the **codomain** T

Python Functions

- We can define functions in Python too
- As in mathematics, they may be used to define common formulae

Example (Fahrenheit-to-Celsius Function)

```
>>> def to_celsius(t):
...     return (5 / 9) * (t - 32)
...
>>> to_celsius(80)
26.666666666666668
>>> to_celsius(78.8)
26.0
```

Python Functions

- A function definition is just a Python statement: It defines a name that is associated with the statements to be performed
- The **keyword** `def` tells Python to define a new function
- As a matter of good style, we use a descriptive name, like `to_celsius`, rather than `f`
- Use a colon instead of an equals sign
- The actual formula for the function is on the next line
- The line is indented with spaces or tabs
- The line is marked with the keyword `return`
- The triple-dot prompt is displayed automatically

Function Calls

Example (Function Calls)

```
>>> to_celsius(10)
-12.222222222222223
>>> to_celsius(212)
100.0
```

- Each of the above statements is called a **function call**—we are calling up the function to do some computation or other work
- We only have to define the function once
- We can call it any number of times

General Form of a Function

General Form of a Function

```
def ⟨function name⟩(⟨parameters⟩):
    ⟨block⟩
```

- **⟨function name⟩** is an identifier; the same rules as for variable names apply
- Zero or more parameters, separated by commas
- A **parameter** is a variable that is given a value when the function is called
- **⟨block⟩** is a block of statements and specifies what the function does
- Note that block is indented

Returning from a Function

Return Statement

`return` ⟨**expression**⟩

- Returns ("sends back") a computed value
- Evaluated as follows:
  1. Evaluate the expression on the right of the keyword `return`
  2. Use that value as the result of the function

Function Definitions v. Function Calls

- When a function is **defined**, Python records it, but does not execute it
- When a function is **called**, Python jumps to the first line of that function, and starts executing it; when it has finished, Python returns to the place where it was called from

Returning Multiple Values

- It is possible to return more than one value from a function
- Simply use more than one expression in a `return` statement, **and separate the expressions by commas**

Example (Multiple Return Values)

```
>>> def one_stddev_limits(mean, stddev):
...     return mean - stddev, mean + stddev
...
>>> one_stddev_limits(100, 13)
(87, 113)
```

Do not worry about the parentheses in the output for now. Once we talk about sequences later, they will make sense.

Local Variables

Often it makes sense to divide a problem into smaller steps.

Example (Local Variables)

```
>>> def polynomial(a, b, c, x):
...     first = a * x * x
...     second = b * x
...     third = c
...     return first + second + third
...
>>> polynomial(2, 3, 4, 0.5)
6.0
>>> polynomial(2, 3, 4, 1.5)
13.0
```

Local Variables

- Variables created inside a function are called **local variables**
- They exist only while the enclosing function is executing
- The **scope** of a variable is the area of a program that can access that variable
- The scope of a local variable runs from the line on which it is first defined to the end of the function in which it is defined

Example (Scope of Local Variables)

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

Built-in Functions

Example (Built-in Functions)

```
>>> abs(-9)
9
>>> round(3.8)
4
>>> round(3.4)
3
>>> pow(2, 5)
32
>>> int(37.8)
37
>>> float(21)
21.0
```

Note that, while `round` correctly rounds up or down, `int` simply truncates the fractional part.