Scientific Programming

Lecture A01 – Introduction to Python

Andrea Passerini

Università degli Studi di Trento

2019/10/22

Acknowledgments: Alberto Montresor, Stefano Teso

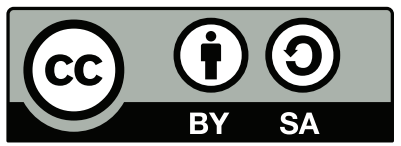


Table of contents

A bit of history

Python is a widely used high-level programming language for general-purpose programming, created by Guido van Rossum.

1991	Python 1.0. Discontinued
1995	Guido van Rossum proclaimed BDFL (Benevolent Dictator for Life)
2000	Python 2.0. Current version 2.7.14. End-of-life: 2020
2008	Python 3.0. Current version 3.6.2
2018	Guido stepped down from BDFL

https://en.wikipedia.org/wiki/Guido_van_Rossum#/media/File:Guido_van_Rossum_OSCON_2006.jpg

Discussion: 2.x vs 3.x

The *pythonic* way

Python has a design philosophy that emphasizes code readability

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated
- Readability counts

Hello World: Syntactic sugar is bad for your health

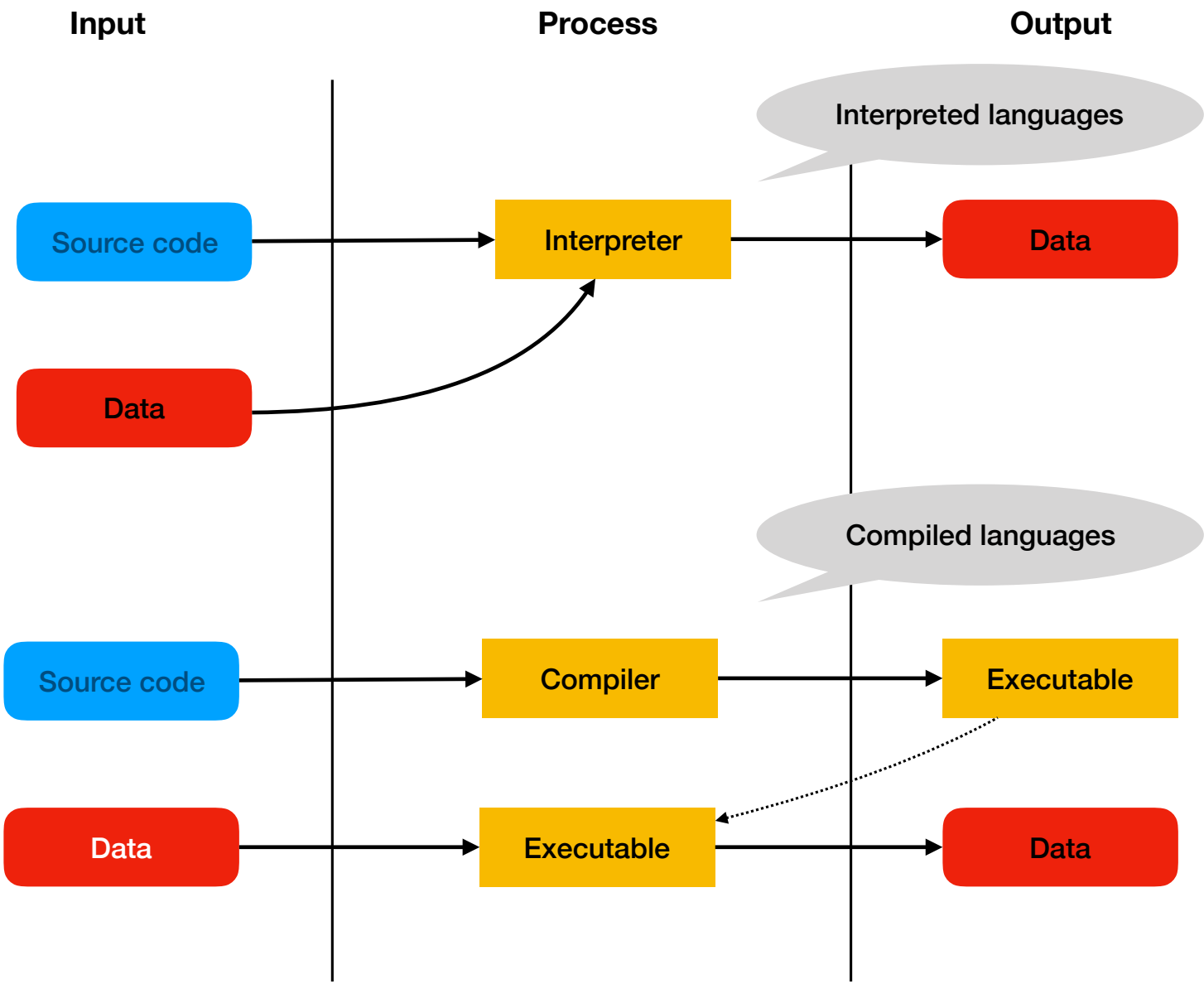
Java

```
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello, World");
    }

}
```

Python

```
print("Hello, World")
```

Interpreted vs compiled languages



Interpreted vs compiled languages

Compiler	**Interpreter**
Takes an entire program as input	Takes lines of input one by one
Generates intermediate object code	Doesn't generate object code
Executes faster	Executes slower
Hard to debug	Easy to debug

Python prompt vs Python programs

Python prompt

```
[andrea@praha ~]$ python
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
>>>
```

Python prompt vs Python programs

Python program: firstprogram.py

```
print("This is my first program")
print(2+3)
```

Execution

```
[andrea@praha ~]$ python firstprogram.py
```

```
This is my first program
5
```

Inside the online books

During the labs

- You will write Python programs using
  - an IDE (Integrated Development Environment)
  - or, your preferred editor
- You will execute Python programs
  - through the IDE
  - or, through the command line

In the online books

- `https://runestone.academy/runestone/static/thinkcspy/GeneralIntro/SpecialWaystoExecutePythoninthisBook.html`

Programs

- **Input**
  Get data from the keyboard, a file, or some other device.
- **Math and logic**
  Perform basic mathematical operations like addition and multiplication and logical operations like and, or, and not.
- **Conditional execution**
  Check for certain conditions and execute the appropriate sequence of statements.
- **Repetition**
  Perform some action repeatedly, usually with some variation.
- **Output**
  Display data on the screen or send data to a file or other device.

Errors and debugging

Syntax errors

```
print("I hate syntax errors)
```

```
File "<stdin>", line 1
  print("I hate syntax errors)
SyntaxError: EOL while scanning string literal
```

Runtime errors

```
a=0
print(10/a)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Errors and debugging

Semantic errors

A program containing a semantic error will run successfully in the sense that the computer will not generate any error messages.

- However, the program will not do the right thing.
- It will do something else.
- Specifically, it will do what you told it to do.

During the course

The more errors,
the better!

After the course

The less errors,
the best grade!

Table of contents

Objects

Object

An object represents a datum (for instance, a number, some text, a collection of things, etc.) that a program manipulates.

Objects are composed of:

- type: the kind of data represented by the object
- value: the data itself

Examples

In the programs we have seen before,

- 2, 3 and 5 are objects (integers)
- "This is my first program" is an object (string)

Python data types (3.x)

Type	**Meaning**	**Domain**	**Mutable?**
`bool`	Condition	`True`, `False`	No
`int`	Integer	$\mathbb{Z}$	No
`float`	Rational	$\mathbb{Q}$ (more or less)	No
`str`	Text	Text	No
`list`	Sequence	Collections of things	Yes
`tuple`	Sequence	Collections of things	No
`dict`	Map	Maps between things	Yes

Variables

Variable

- Variables are references to objects
- You can view them as names for the objects that they refer to.
- The type of a variable is given by the type of the object it refers to

Example

```
pi = 3.1415926536
print(pi)
print(type(pi))
```

```
3.1415926536
<class 'float'>
```

Choosing variable names

- Variable names are a choice of the programmer
- They must be as significant as possible

Variable names

Rules to create name variables

- They can only contain letters,numbers and the underscore `a-z,A-Z,0-9,_`
- They cannot start with a number
- Some words are reserved keywords of the language

Examples – Invalid syntax

```
76trombones = "big parade"
more\$ = 1000000
class = "Computer Science 101"
```

Reserved keywords

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

Assignment operator

Don't confuse the assignment operator = with the equality operator in mathematics

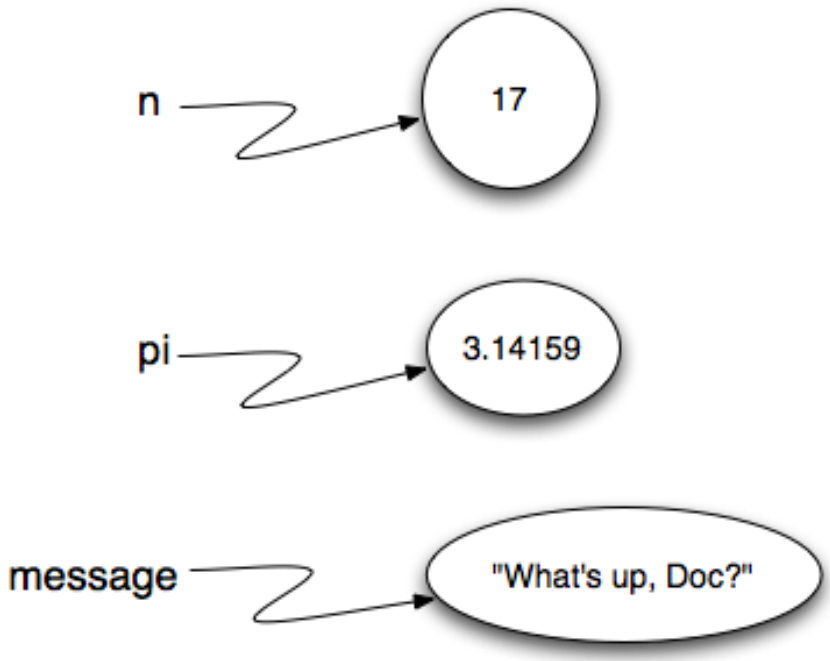
```
# This is not a valid Python statement
17 = n
```

```
  File "<stdin>", line 2
SyntaxError: can't assign to literal
```

Variables and memory

Objects live in the memory cells of the computer, and variables are references to those cells.

```
message = "What's up, Doc?"
n = 17
pi = 3.14159
```



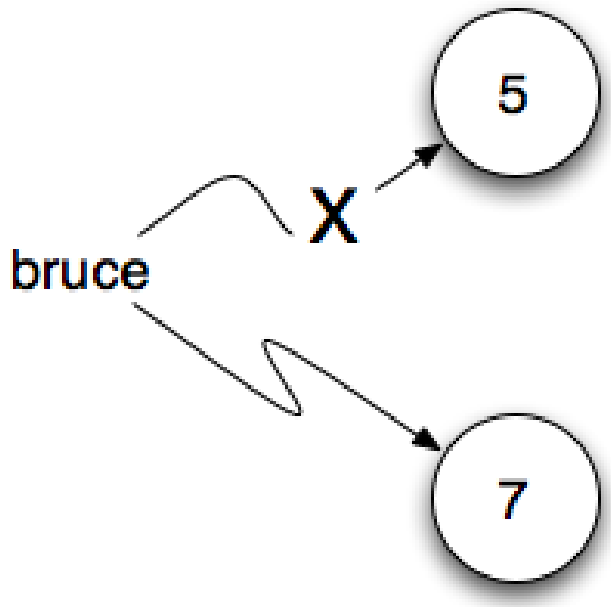
https://runestone.academy/runestone/static/thinkcspy/SimplePythonData/Variables.html

Re-assignment

Variables may change their values during their life.
(Otherwise, why call them "variables"???)

```
bruce = 5
print(bruce)
bruce = 7
print(bruce)
```

```
5
7
```



https://runestone.academy/runestone/static/thinkcspy/SimplePythonData/Reassignment.html

Undefined variables

Variables must be initialized before they can be used

```
print(r*r*3.14)
r = 2
```

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

Assignment from other variables

- It is very common to copy a variable into another
- After the 1st assignment, **a** and `b` refer to the same object
- After the 2nd assignment, **a** and `b` refer to different objects

```
a = 5
b = a
print(a, b)
a = 3
b = 4
print(a, b)
```

```
5 5
3 4
```

https://runestone.academy/runestone/static/thinkcspy/SimplePythonData/Reassignment.html

Updating variables

One of the most common forms of reassignment is an update where the new value of the variable depends on the old.

```
x = 6          # initialize x
print(x)                                  6
x = x + 1      # update x
print(x)                                  7
```

https://runestone.academy/runestone/static/thinkcspy/SimplePythonData/UpdatingVariables.html

Multiple types

Variables may have multiple types during their life.
Normally, this is not a good idea!

```
var = 3
print(var*2)
print(type(var))
```

```
6
<class 'int'>
```

```
var = 3.1415926536
print(var*2)
print(type(var))
```

```
6.2831853072
<class 'float'>
```

```
var = "3.14"
print(var*2)
print(type(var))
```

```
3.143.14
<class 'str'>
```

Numeric types

How to write numeric values

```
x = 10          # Integer
y = 123.45      # Float
z = 1.2345e2    # Float
```

Numeric operators

+,-,*	sum,difference,product
/	division
//	integer division
%	remainder
**	power

Difference between 2.x and 3.x

```
# Python 2.x
>>> 2/3
0
>>> 2//3
0
```

```
# Python 3.x
>>> 2/3
0.6666666666666666
>>> 2//3
0
```

Type conversions

Type conversion

- Automatic conversions
- From `float` to `int`: `int(2.5)`
- From `int` to `float`: `float(2)`

```
print(1+1.0)
print(type(1+1.0))
print(int(3.0))
print(float(1))
print("The value is " + str(1))
```

```
2.0
<class 'float'>
3
1.0
The value is 1
```

Boolean values (`True`, `False`) and operators

"Multiplication tables" (Tabelline)

and	False	True
False	False	False
True	False	True

or	False	True
False	False	True
True	True	True

a	not a
False	True
True	False

Truth tables

a	b	a and b	a or b
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

Comparators

Comparators

`a == b`	`True` if and only if $a = b$
`a != b`	`True` if and only if $a \neq b$
`a < b`	`True` if and only if $a < b$
`a > b`	`True` if and only if $a > b$
`a <= b`	`True` if and only if $a \leq b$
`a >= b`	`True` if and only if $a \geq b$

Boolean: rules and examples

Associativity

```
a and (b and c) == (a and b) and c
a or (b or c) == (a or b) or c
```

Commutativity

```
a and b == b and a
a or b == b or a
```

Example

```
x = 17
print(x <= 34 and (x >= 12 and x % 2 == 1))
print(12 <= x <= 34 and (x % 2 == 1))
```

Boolean: rules and examples

Associativity

```
a and (b and c) == (a and b) and c
a or (b or c) == (a or b) or c
```

Commutativity

```
a and b == b and a
a or b == b or a
```

Example

```
x = 17
print(x <= 34 and (x >= 12 and x % 2 == 1))
print(12 <= x <= 34 and (x % 2 == 1))
```

```
True
True
```

Boolean: rules and examples

De Morgan Laws

```
not (a and b) == (not a) or (not b)
not (a or b) == (not a) and (not b)
```

Example

```
x = 17
print( not (x >= 12 and x <= 34) )
print( x < 12 or x > 34 )
```

Boolean: rules and examples

De Morgan Laws

```
not (a and b) == (not a) or (not b)
not (a or b) == (not a) and (not b)
```

Example

```
x = 17
print( not (x >= 12 and x <= 34) )
print( x < 12 or x > 34 )
```

```
False
False
```

Boolean: rules and examples – Short Circuiting

Annihilator

If the first part of a boolean expression decide the final result, the evaluation of the rest can be skipped (short circuit)

```
False and a == False
```

```
True or a == True
```

Example

```
a = 0
b = 2
print(a > 0 and 12/a >= 0)
print(b > 0 and 12/b == 6)
print(12/a)
```

Boolean: rules and examples – Short Circuiting

Annihilator

If the first part of a boolean expression decide the final result, the evaluation of the rest can be skipped (short circuit)

```
False and a == False
```

```
True or a == True
```

Example

```
a = 0
b = 2
print(a > 0 and 12/a >= 0)
print(b > 0 and 12/b == 6)
print(12/a)
```

```
False
True
ZeroDivisionError
```

Boolean: rules and examples

Identity

```
True and a == a
False or a == a
```

Example

```
print( 372 > 0 and 372 % 3 == 0) )
print( 372 % 3 == 0) )
```

Boolean: rules and examples

Identity

```
True and a == a
False or a == a
```

Example

```
print( 372 > 0 and 372 % 3 == 0) )
print( 372 % 3 == 0) )
```

```
True
True
```

Boolean: rules and examples

Distributivity

```
a and (b or c) == (a and b) or (a and c)
a or (b and c) == (a or b) and (a or c)
```

Example

```
a = 5
b = 7
print( a < b and (a < 7 or b < 10 ) )
print( (a < b and a < 7) or (a < b < 10) )
```

Boolean: rules and examples

Distributivity

```
a and (b or c) == (a and b) or (a and c)
a or (b and c) == (a or b) and (a or c)
```

Example

```
a = 5
b = 7
print( a < b and (a < 7 or b < 10 ) )
print( (a < b and a < 7) or (a < b < 10) )
```

```
True
True
```

Operator precedence

`**`	Power (Highest precedence)
`+,-`	Unary plus and minus
`* / // %`	Multiply, divide, floor division, modulo
`+ -`	Addition and subtraction
`<= < > >=`	Comparison operators
`== !=`	Equality operators
`not or and`	Logical operators (Lowest precedence)

Example

```
2+3*4**2 == 23+3**3 and 3*-1**2+7 != 10
```

Operator precedence

`**`	Power (Highest precedence)
`+,-`	Unary plus and minus
`* / // %`	Multiply, divide, floor division, modulo
`+ -`	Addition and subtraction
`<= < > >=`	Comparison operators
`== !=`	Equality operators
`not or and`	Logical operators (Lowest precedence)

Example

```
2+3*4**2 == 23+3**3 and 3*-1**2+7 != 10

(2+(3*(4**2))) == (23+(3**3)) and (((3*(-(1**2)))+7) != 10)
```

Expressions vs Statements

Expression

- An expression is a combination of values, variables, operators, and calls to functions
- When you type an expression at the prompt, the interpreter evaluates it

Example

```
>>> (2+3)*5
25
>>> 42
42
>>> n = 17
>>> n+25
42
```

Expressions vs Statements

Statement

- A statement is a unit of code that has an effect, like creating a variable or displaying a value.
- Expressions by themselves are not statements

Example

```
a=(2+3)*5
a*2
print(a)
```

```
25
```

Assignment

Assignment operator

Operator	Before	After
`a = 3`	?	3

Compund assignment operators

Operator	Before	After
`a += 3`	3	6
`a -= 3`	6	3
`a *= 3`	3	9
`a /= 3`	9	3
`a **= 3`	3	27

Functions and methods

Function

A function takes zero or more objects as inputs (its arguments), performs some operations on them, and returns zero or more objects (its results).

Function

You invoke a function by writing the name of the function, followed by a pair of parenthesis containing objects for each of the parameters. You can optionally collect the result of the function.

```
result = f(par1, par2, ...)
```

(Some) Built-in functions

`abs()`	Return the absolute value of a number
`max()`	Return the maximum between two or more values
`min()`	Return the maximum between two or more values
`round()`	Round a floating point number to a desired number of digits
`print()`	Print the arguments

Example

```
val = abs(-3)
print(val, max(2,3), round(3.1415926536, 2))
```

```
3 3 3.14
```

Functions

Python functions and mathematical functions

Python functions and mathematical functions share some similarities: informally, both evaluate their arguments (inputs) to produce a result (output).

Differences

- Not all Python functions require arguments
- Python functions may return an empty result
- Mathematical functions can only operate on their arguments, while Python functions can access additional resources
- Mathematical functions cannot modify the "external environment". Python functions can have side-effects

Simple input

`input()`

You can use the built-in function `input()` to obtain data from the user. It has no input parameter and returns a single string. The string has to be converted to a number, if necessary.

Example

```
val = int(input())
print(val*val)
```

Methods

Methods

A method is exactly like a function, except that it is provided by a given type and is applied to a specific instance of that type.

Example

Type `string` provides a method `upper()` that returns an upper-case version of the original string, without modifying it.

```
s = "hello world"
print(s)
print(s.upper())
print(s)
```

```
hello world
HELLO WORLD
hello world
```

References and Exercises

Book

Chapter 1 and 2

Exercises

`https://runestone.academy/runestone/static/thinkcspy/SimplePythonData/Exercises.html`