An introduction to Python

Fabrice Rossi

CEREMADE
Université Paris Dauphine

2020

Python

https://www.python.org/

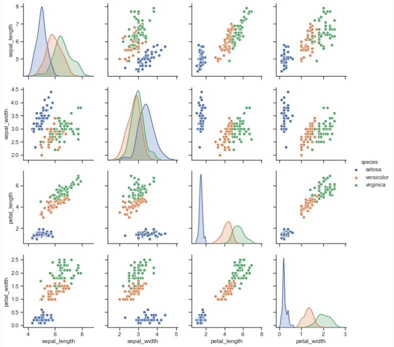
- Python is a high level programming language
- Python's reference implementation is a multiplatform free software
- Python can be extended by thousands of libraries
- Python is generally considered to be easy to learn

```python
name = input("What's your name? ")
print("Hello", name)
```

```
What's your name? John Doe
Hello John Doe
```

Python and data science

- Python is one of the two de facto standard languages for data science (with R)
- Python has a large collection of high performance data science oriented libraries
- Python is generally considered to be easy to read



Python

Pros

- open source implementation
- full-fledged programming language
- strong support from a large community
- broad coverage of data science, statistics, etc.
- high performance libraries
- high quality graphics
- curated distribution

Cons

- limited point-and-click support
- rather steep learning curve compared to an integrated software
- naive code has low performances
- “old” language (1990) with a lack of modern constructs

Recommended installs

- Anaconda (with Python 3.x)
  - https://www.anaconda.com/distribution/
  - a python distribution: python + libraries + tools
  - data science oriented
  - anaconda navigator for managing the distribution
- recommended tools (in Anaconda)
  - VS code or Spyder for Python programming
  - JupyterLab for literate programming
- other IDE include PyCharm
- do not use Python 2.7

Outline

Introduction

Core concepts

Control structures

Functions

Exception handling

Outline

Programming Language

Definition

- a formal language with a strict mathematical definition
- defines syntactically correct programs
- associated to a semantics
  - (formal) model of the computer
  - effects of a program on the model

Programming Language

Definition

- a formal language with a strict mathematical definition
- defines syntactically correct programs
- associated to a semantics
  - (formal) model of the computer
  - effects of a program on the model

In other words...

- a programming language can be used to write programs $\simeq$ texts
- a programming language has a strict syntax
  - lexical aspects $\simeq$ word spelling
  - grammatical aspects $\simeq$ sentence level
- when a program follows the syntax, it has a proper meaning i.e. an effect on the computer on which it runs

A computer

Turing Machine

- standard mathematical model
- too low level to a daily use

Other models

- data oriented models
- a model of the data
- together with a model of the execution of a program
  - effects of instructions/statements on the data $\simeq$ sentence level
  - global flow and organization on a program $\simeq$ text level
- include input/output aspects

Interactive mode

Standard program execution

- a program is written in a file (or a set of files)
- in some languages the file can be translated to a more efficient language
- the file (or its translation) is executed on a computer

Console/Shell

- some languages have an associated “console” or “shell” (e.g. Python and R)
- one can type interactively program sentences and get associated results
- simplifies learning and testing

Python Shell

- Python provides a shell for interactive use
- in general integrated in a specific window of a programming environment
- can be launched from the command line (`python`)
- command prompt `>>>`

Python Shell

- Python provides a shell for interactive use
- in general integrated in a specific window of a programming environment
- can be launched from the command line (`python`)
- command prompt >>>

```
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license"
   for more information.
>>>
```

Python Shell

- Python provides a shell for interactive use
- in general integrated in a specific window of a programming environment
- can be launched from the command line (`python`)
- command prompt `>>>`

```
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license"
   for more information.
>>> 2 + 2
```

Python Shell

- Python provides a shell for interactive use
- in general integrated in a specific window of a programming environment
- can be launched from the command line (`python`)
- command prompt `>>>`

```
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license"
   for more information.
>>> 2 + 2
4
>>>
```

Python Shell

- Python provides a shell for interactive use
- in general integrated in a specific window of a programming environment
- can be launched from the command line (`python`)
- command prompt `>>>`

```
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license"
   for more information.
>>> 2 + 2
4
>>> 4 ** 3
```

Python Shell

- Python provides a shell for interactive use
- in general integrated in a specific window of a programming environment
- can be launched from the command line (`python`)
- command prompt `>>>`

```
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license"
   for more information.
>>> 2 + 2
4
>>> 4 ** 3
64
>>>
```

Python Shell

- Python provides a shell for interactive use
- in general integrated in a specific window of a programming environment
- can be launched from the command line (`python`)
- command prompt `>>>`

```
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license"
   for more information.
>>> 2 + 2
4
>>> 4 ** 3
64
>>>
```

Warning

The behavior of a program in the shell is not exactly the same as the behavior of a program outside of the shell

Python Shell as a calculator

```
>>> 2.5 / 1.3
1.923076923076923
>>> 2,5 / 1,3
(2, 5.0, 3)
>>> 1 + 2 / 3
1.6666666666666665
>>> (1 + 2) / 3
1.0
>>> 5 / 2
2.5
>>> 5 // 2
2
>>> 5 % 2
1
>>> -5 // 2
-3
>>> 5 ** 5
3125
>>> 2 ** 0.5
1.4142135623730951
```

```
>>> 12.5 - 4 / 5
11.7
>>> _ + 2
13.7
>>> 4.5 > 3.5
True
>>> (2.5 >= 3) or (2.5 < 3)
True
 >>> -1 ** 0.5
-1.0
>>> (-1) ** 0.5
(6.123233995736766e-17+1j)
>>> _ ** 2
(-1+1.2246467991473532e-16j)
>>> 0j
0j
>>> 1j ** 2
(-1+0j)
```

Basic data model

Numerical values

- integers
- real numbers
  - decimal point
  - classical scientific notation e.g. `1.5e-3`
- complex numbers
  - automatically used in some situations
  - `real + img j`

Arithmetic operations

- standard operations
- integer oriented

Basic data model

Numerical values

- integers
- real numbers
  - decimal point
  - classical scientific notation e.g. `1.5e-3`
- complex numbers
  - automatically used in some situations
  - `real + img j`

Arithmetic operations

- standard operations
- integer oriented

Logical expressions

- boolean (a.k.a. truth value)
- **`True`** and **`False`** values
- automatic integer conversion to 1 and 0, respectively (and vice versa)
- logical operators **`and or not`**
- numerical comparisons
  - `==` and `!=`
  - `<=` and `<` (and reversed ones)

Basic data model

Syntax

- literal values (spelling)
  - e.g. numbers and truth values
  - Python specifies how to write them
  - e.g. `1,5` is not a real number!
- operations (grammar)
  - writing rules are similar to mathematical ones
  - with exceptions such as
    - `==` for equality
    - `**` for exponentiation
    - etc.

Semantics

- interpretation off the symbols and of the expressions such as:
  - calculation ordering
  - `==` tests for equality
  - `**` can produce complex numbers
  - _ is the last value computed
- error cases

```
>>> 0/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Objects and variables

Objects

- Python manipulates objects
- each object has a type
  - specifies the possible values
  - specifies the possible operations
- examples
  - `2` is an `int`
  - `2.5` is a `float`
  - **`True`** is a `bool`
  - `1+2j` is a `complex`

Variables

- objects can be named
- a variable is a name for an object
- setting/binding a name:

  `variable = object`
- when a name appears in an expression it is replaced by the object
- example

```
>>> x = 2
>>> 2 * x
4
```

Examples

```
>>> x = 4
>>> y = 3
>>> y / x
0.75
>>> z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
>>> y / X
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'X' is not defined
>>> z = x
>>> z
4
>>> x = 3
>>> z
4
>>> y = z < x
>>> y
False
```

Key points

- (obvious) sequential model
- no default binding
- case dependant
- aliases: several names for a given object
- `variable = variable` does not bind the names together
- unconstrained rebinding

Texts

```
>>> "A text"
'A text'
>>> 'Another text'
'Another text'
>>> '''yet
... another
... text'''
'yet\nanother\ntext'
>>> ''
''
>>> u = 'my text'
>>> u * 2
'my textmy text'
>>> t = ' is mine!'
>>> u + t
'my text is mine!'
>>> 2 + t
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Strings

- type `str` (string)
- literal `' '` or `" "`
- multiline with `''' '''`
- concatenation
- types are not compatible in general!

Slices

```
>>> x = 'abcdefg'
>>> x[0]
'a'
>>> x[4]
'e'
>>> x[-1]
'g'
>>> x[0:3]
'abc'
>>> x[:4]
'abcd'
>>> x[2:]
'cdefg'
>>> x[-3:]
'efg'
>>> x[:-3]
'abcd'
>>> x[7]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Indexing

- some Python objects can be indexed
- `[index]`
- numbering always start at 0
- negative indexing enables reverse ordering
- slices `a:b`
  - from `a` to `b-1`
  - missing `a`: 0
  - missing `b`: last index + 1
- negative slicing: same logic

Functions

Additional actions

- objects can be manipulated with more than operators
- functions provide such additional actions
- a function
  - has a name
  - needs 0 or more argument(s)
  - possibly returns an object
- using a function
  - function call
  - `function(argument_1, argument_2)`
  - `function()`

```
>>> len('abcd')
4
>>> type(2)
<class 'int'>
>>> type('2')
<class 'str'>
>>> complex(2,-1)
(2-1j)
>>> int(2.4)
2
>>> round(17.23,1)
17.2
>>> str(3)
'3'
>>> abs(-4)
4
>>> x = 2
>>> 'x=' + str(x)
'x=2'
```

Functions

Functions are objects

- type `function` for general functions
- specific type for built in functions
- all standard properties apply
  - new names
  - function as an argument to a function

```
>>> len
<built-in function len>
>>> type(len)
<class 'builtin_function_or_method'>
>>> foo = len
>>> foo
<built-in function len>
>>> foo('abc')
3
>>> str(foo)
'<built-in function len>'
```

Methods

Functions for objects

- *methods* are specific functions associated to some object types
- special calling syntax `object.function()`
- equivalent to `Type.function(object)`

```
>>> 'bla'.capitalize()
'Bla'
>>> 'tototi'.find('t')
0
>>> 'tototi'.find('ti')
4
>>> foo = 'et' * 3
>>> foo
'etetet'
>>> foo.upper()
'ETETET'
>>> foo.count('et')
3
```

Modules

Extending Python

- *modules* provide new functions and types
- a module must be imported to have access to its content
- default module `sys`

Importing modules

- **`import module`** gives access to the names in the module via `module.name`
- **`import module as bla`** turns that into `bla.name`

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.factorial(20)
2432902008176640000
>>> math.log(2)
0.6931471805599453
>>> math.ceil(3.4)
4
>>> import random as rd
>>> rd.random()
0.9786544666626154
>>> rd.random()
0.7496554473100112
>>> rd.randint(1,10)
2
>>> rd.randint(1,10)
8
```

Outline

Input and output

Console limitation

- has to be used interactively
- commands are not saved
- reproducibility is not guaranteed

Scripts

- normal simple python programs are script
- a script: a text file (generally ending with .py)
- a script is executed by the python interpreter

Outputs

- to output something, use the `print` function
- for instance

```
print(2, 'toto')
x = 3
print(x, 2 * x, 2 ** x)
```

will print

```
2 toto
3 6 8
```

Inputs

- to input something, use the `input` function
- returns always a string `str`
- convert if needed

Conditional execution

Execute if...

- programs can include parts that are executed only if some condition is fulfilled
- the condition is written as a Boolean expression

General form

```
if expression:
   statement_1
   statement_2
   ...
   statement_n
rest of the program
```

`if` in Python

- `if` is a *compound* statement
- it consists a *clause* comprising
  - a *header*
    `if expression:`
  - a *suite* whose execution is controlled by the *header*
- in general the *suite* (a.k.a. the *body*) is made of a series of *indented* statements

Semantics

the body is executed if and only if the expression of the clause evaluates to **`True`**

More conditional execution

Other clauses in `if`

- a `if` statement can contain
  - one or more `elif` clauses
  - one `else` clause
- general form

```
if expression_1:
   statement_1
   ...
   statement_n
elif expression_2:
   ...
elif expresion_3:
   ...
else:
   ...
rest of the program
```

Semantics

The compound instruction is executed as follows

- the `expression` of the `if` header is evaluated
- if the value is `True` then the body is executed and the execution resumes for the rest of the program
- if the value is `False` the body is ignored the execution resumes on the second clause
- for each `elif` header, the execution follows the same pattern:
  - if the corresponding `expression` is `True` the body of the clause is executed, followed by the rest of the program
  - if not the execution resumes on the next clause
- if all expressions evaluate to `False` the body of the `else` clause is executed

Repeating instructions

Multiple executions

- programs can include parts that are executed several times
- repetitions can be conditional or numbered

Conditional loop

```
while expression:
   statement_1
   ...
   statement_n
rest of the program
```

`while` in Python

- *compound* statement (single clause)
- `while expression:` is the header of the clause

Semantics

- the expression of the header is evaluated
- if the value is `True`
  - the body is executed
  - the execution resumes on clause itself!
- if the value is `False` the execution resumes for the rest of the program

Iteration

Iterable objects

- objects which can be decomposed into several other objects
- the content of an iterable object is arranged in a certain order
- *iterating* over the object means accessing in order to its elements

Strings

- string "content": characters
- iterating a string: in character order!
- `'foobar'` gives `'f'`, `'o'`, `'o'`, `'b'`, `'a'` and `'r'`

Iterating iterables

For loops

- specific loop for iterables
- the loop execute a code for each value contained in the iterable

General form

```
for variable in expression:
   statement_1
   ...
   statement_n
rest of the program
```

`for` in Python

- *compound* statement (single clause)
- `for variable in expression:` is the header of the clause
- the expression of the header must evaluate to an iterable object

For semantics

Semantics

- the expression is evaluated to get an iterable object
- for each object in the iterable
  - the `variable` is bound to the object
  - the body of the clause is executed
- then the execution of the rest of the program resumes
- if the iterable is empty, the for loop does not execute (no error)

Example

The program

```
for x in 'foobar':
   print(x)
```

prints

```
f
o
o
b
a
r
```

Ranges

Repeating *n* times some operations

- very common case
- easy to do with a **while** but not immediately obvious

```
k = 0
while k < n:
  something
  to repeat
  n times
  k = k + 1
more statements
```

`range` objects

- integer range iterable
- `range(n)`: integers from 0 to `n-1` (`n` values)
- simpler solution

```
for k in range(n):
  something
  to repeat
  n times
more statements
```

- clearer for python programmer

Ranges

More ranges

- `range` operates in a similar way to slices
- `range(end)`: integers from 0 to `end-1`
- `range(begin, end)`: integers from begin to `end-1`
- `range(begin, end, step)`: integers from begin to `end-1` by increments of `step`
  - `range(1, 4, 2)` : 1 and 3
  - `range(1, 5, 2)` : 1 and 3
- works with negative increments
  - `range(5, 2)` : empty
  - `range(5, 2, -1)`: 5, 4 and 3

Outline

Introduction

Core concepts

Control structures

Functions
- Defining functions
- Namespaces
- Recursive functions
- Parameters and arguments

Exception handling

Defining functions

Benefits of user defined functions

- provide program organization
- reduce code repetition
- enable using generic functionalities

Example

```
def onemore(x):
    return x + 1
```

General form

```
def function_name(p_1,...,p_n):
    statement_1
    ...
    statement_n
```

`return`

- the `return` statement defines the *value* of the function
- it terminates the function execution

Vocabulary

- the code above is a *function definition*
- `p_1,...,p_n` are the *formal parameters* of the function (possible none)
- the statements form the *body* of the function

Calling a function

Definition versus call

- the function definition only makes it available in the rest of the program
- a (standard) function call is needed to use it
  `function_name(a_1,...,a_n)`
- the expression `a_1,...,a_n` are the *arguments* of the *call*

Semantics

a function call is evaluated as follows

1. arguments are evaluated
2. a new *namespace* is created
3. formal parameters become variables in the new *namespace* and are bound to the corresponding arguments
4. the body of the function is executed
5. the *namespace* is discarded
6. the value of the function call is the result of the execution of the body

Example

Program

```
def onemore(x):
    return x + 1

a = 2
b = onemore(a + 2)
```

Execution

Namespaces

Example

Program

```
1 def onemore(x):
2     return x + 1
3
4 a = 2
5 b = onemore(a + 2)
```

Execution

- lines 1 and 2:
  - function definition
  - `onemore` is added to the global namespace
  - no other statement are executed

Namespaces

global
- `onemore`

Example

Program

```
1 def onemore(x):
2     return x + 1
3
4 a = 2
5 b = onemore(a + 2)
```

Namespaces

global

- `onemore`
- `a` → 2

Execution

- lines 1 and 2:
  - function definition
  - `onemore` is added to the global namespace
  - no other statement are executed
- line 4: `a` added to the global namespace with value 2

Example

Program

```
1    def onemore(x):
2        return x + 1
3
4    a = 2
5    b = onemore(a + 2)
```

Namespaces

global

- onemore
- a → 2

Execution

- lines 1 and 2:
  - function definition
  - `onemore` is added to the global namespace
  - no other statement are executed
- line 4: `a` added to the global namespace with value `2`
- line 5:

Example

Program

```
1 def onemore(x):
2     return x + 1
3
4 a = 2
5 b = onemore(a + 2)
```

Namespaces

global

- onemore
- a → 2

Execution

- lines 1 and 2:
  - function definition
  - `onemore` is added to the global namespace
  - no other statement are executed
- line 4: `a` added to the global namespace with value `2`
- line 5:
  - `a + 2` is evaluated to `4`

Example

Program

```
1  def onemore(x):
2      return x + 1
3
4  a = 2
5  b = onemore(a + 2)
```

Namespaces

global
- onemore
- a → 2

Execution

- lines 1 and 2:
  - function definition
  - `onemore` is added to the global namespace
  - no other statement are executed
- line 4: `a` added to the global namespace with value `2`
- line 5:
  - `a + 2` is evaluated to `4`
  - a local namespace is created

Example

Program

```
1 def onemore(x):
2     return x + 1
3
4 a = 2
5 b = onemore(a + 2)
```

Namespaces

global
- `onemore`
- `a` → `2`

local
- `x` → `4`

Execution

- lines 1 and 2:
  - function definition
  - `onemore` is added to the global namespace
  - no other statement are executed
- line 4: `a` added to the global namespace with value `2`
- line 5:
  - `a + 2` is evaluated to `4`
  - a local namespace is created
  - formal parameters are bound to arguments

Example

Program

```
1  def onemore(x):
2      return x + 1
3
4  a = 2
5  b = onemore(a + 2)
```

Namespaces

global
- onemore
- a → 2

local
- x → 4

Execution

- lines 1 and 2:
  - function definition
  - `onemore` is added to the global namespace
  - no other statement are executed
- line 4: `a` added to the global namespace with value `2`
- line 5:
  - `a + 2` is evaluated to `4`
  - a local namespace is created
  - formal parameters are bound to arguments
  - line 2 is executed
    - `x + 1` is evaluated to `5`
    - the return value of `onemore` is bound to `5`

Example

Program

```
1 def onemore(x):
2     return x + 1
3
4 a = 2
5 b = onemore(a + 2)
```

Namespaces

global

- `onemore`
- `a` → `2`

Execution

- lines 1 and 2:
  - function definition
  - `onemore` is added to the global namespace
  - no other statement are executed
- line 4: `a` added to the global namespace with value `2`
- line 5:
  - `a + 2` is evaluated to `4`
  - a local namespace is created
  - formal parameters are bound to arguments
  - line 2 is executed
    - `x + 1` is evaluated to `5`
    - the return value of `onemore` is bound to `5`
  - the local namespace is discarded

Example

Program

```
1 def onemore(x):
2     return x + 1
3
4 a = 2
5 b = onemore(a + 2)
```

Namespaces

global
- onemore
- a → 2

Execution

- lines 1 and 2:
  - function definition
  - `onemore` is added to the global namespace
  - no other statement are executed
- line 4: `a` added to the global namespace with value `2`
- line 5:
  - `a + 2` is evaluated to `4`
  - a local namespace is created
  - formal parameters are bound to arguments
  - line 2 is executed
    - `x + 1` is evaluated to `5`
    - the return value of `onemore` is bound to `5`
  - the local namespace is discarded
  - `onemore(a + 2)` is evaluated to `5`

Example

Program

```
1   def onemore(x):
2       return x + 1
3
4   a = 2
5   b = onemore(a + 2)
```

Namespaces

global

- `onemore`
- `a` → 2
- `b` → 5

Execution

- lines 1 and 2:
  - function definition
  - `onemore` is added to the global namespace
  - no other statement are executed
- line 4: `a` added to the global namespace with value `2`
- line 5:
  - `a + 2` is evaluated to `4`
  - a local namespace is created
  - formal parameters are bound to arguments
  - line 2 is executed
    - `x + 1` is evaluated to `5`
    - the return value of `onemore` is bound to `5`
  - the local namespace is discarded
  - `onemore(a + 2)` is evaluated to `5`
  - `b` is bound `5` in the global namespace

Return

Semantics

- `return` both
  - binds the value of the function
  - interrupts its execution
- a function can contain multiples `return` statements (only one will be executed)
- when a function contains no `return` statement
  - its value is `None`
  - its execution continues until the end of its body

Examples

Multiple **return**

```
1 def my_fun(x, y):
2     if x > y:
3         return x
4     else:
5         return y
```

- the function value is obviously the largest of its two arguments
- if the first argument is the largest one, the first **return** statement is executed and thus only lines 2 and 3 are executed
- in the other case, the second **return** statement is executed

No **return**

```
1 def foo(x):
2     x = x + 1
3     print(x)
```

- lines 2 and 3 are always executed
- the value of the function is **None**
- do not confuse printing and returning a value! The program

```
def foo(x):
    x = x + 1
    print(x)

y = foo(2)
print(y)
```

prints

```
3
None
```

Namespaces

Definition

A namespace binds names to objects

Examples

- the *built-in* namespace (with `type`, `len`, etc.)
- the *global* namespace of a program
- the *local* namespace of a function (during its execution)

Important aspects

- namespaces are runtime dynamical entities
- two different namespaces can contain the same name bound to different objects

Scopes

Definition

A scope is a textual part of a program in which a namespace is *directly* accessible

Examples

- a Python program is a scope (associated to the global namespace of the program) which is *enclosed* in the scope of the built-in namespace
- a function definition defines a scope which is *enclosed* in the global scope

Access rules

Directly accessible

- names in the namespace of the local scope are directly accessible (those are *local* names)
- names in namespaces associated to enclosing *function* scopes are directly accessible (when a function is defined inside another function)
- global names are accessible (names in the global enclosing namespace)
- built-in names are accessible
- names are searched for in order from the local scope to the built-in one: the first match is used!

Example

Non local access

This program

```
x = 1 # global scope

def f(y):
    # local scope of f
    return max(x, y)

print(f(2))
x = 3
print(f(2))
```

prints

```
2
3
```

Scopes

1. built-in
2. global (the program)
3. local to `f`

Accesses

- `max` is accessible as a name of the built-in namespace
- `y` is accessible in `f` as a name of the namespace created when `f` is executed and attached to the scope of `f`
- `x` is accessible in `f` as a name of the global namescape attached to the global scope which encloses the scope of `f`

Example

Non local access

This program

```
x = 1 # global scope

def f(y):
    # local scope of f
    return max(x, y)

print(f(2))
x = 3
print(f(2))
```

prints

```
2
3
```

Do not do that!

Scopes

1. built-in
2. global (the program)
3. local to `f`

Accesses

- `max` is accessible as a name of the built-in namespace
- `y` is accessible in `f` as a name of the namespace created when `f` is executed and attached to the scope of `f`
- `x` is accessible in `f` as a name of the global namescape attached to the global scope which encloses the scope of `f`

Examples

Cannot access enclosed scopes

In this program

```
1 def f(z):
2     return z + 1
3
4 print(f(2))
5 print(z)
```

line 5 prints an error of the form

```
NameError: name 'z' is not defined
```

`z` is not accessible in the global scope.

Priority

This program

```
def g(x):
    return x + 1

x = 2
print(g(3))
print(x)
```

prints

```
4
2
```

- `x` is both a local name (parameter) and a global one
- the name is searched first in the local namespace and then in enclosing ones

Recursive functions

Calling oneself

- a function body may contain calls to itself
- leverage dynamic namespaces: each call has its own namespace

Example

```
def facto(n):
    if n <= 1:
        return 1
    else:
        return n * facto(n-1)
```

Recursive functions

Calling oneself

- a function body may contain calls to itself
- leverage dynamic namespaces: each call has its own namespace

Example

```python
def facto(n):
    if n <= 1:
        return 1
    else:
        return n * facto(n-1)
```

Analyzing a call

```
facto(4)
        n → 4
        facto(3)
                n → 3
                facto(2)
                        n → 2
                        facto(1)
                                n → 1
                                return 1
                        n * facto(1) → 2
                        return 2
                n * facto(2) → 6
                return 6
        n * facto(3) → 24
        return 24
```

Matching parameters and arguments

Positional matching

- standard case
- definition
  `def function_name(p_1,...,p_n)`
- call `function_name(a_1,...,a_n)`
- constraints and semantics
  - exactly as many arguments as formal parameters
  - `p_k` is bound to `a_k`
- the position of the argument decides its formal parameters

Example

The program

```
def f(x, y):
    return x - y

x = 2
y = 3
print(f(y, x))
```

prints

```
1
```

Keyword Arguments

Name matching

- definition
  ```
  def function_name(p_1,...,p_n)
  ```
- call
  ```
  function_name(p_1 = a_1,...,p_n = a_n)
  ```
- constraints and semantics
  - exactly as many arguments as formal parameters
  - `p_k` is bound to the argument associated to its name in the call
- only the names are used, not the positions
  ```
  function_name(p_n = a_n,...,p_1 = a_1)
  ```

Example

The program

```
def f(x, y):
    return x - y

print(f(y = 3, x = 2))
```

prints

```
-1
```

Mixing both types

Rules

- a function call may mix positional arguments and keyword arguments
- positional arguments must appear first
- when a keyword argument is used, all subsequent arguments must use the keyword mode

Examples

- with
  ```python
  def f(a, b, c):
      ...
  ```
- incorrect calls
  - `f(2,b=3,4)`
  - `f(b=3,c=4,1)`
- correct calls
  - `f(2,b=3,c=4)` (`a` is bound to 2)
  - `f(2,c=5, b=2)` (`a` is bound to 2)

Default arguments

Main use of keyword arguments

- to enable function calls with missing arguments
- via default values for missing arguments

Function definition

with default values

```
def f_n(p_1=d_1,...,p_n=d_n):
    statement_1
    ...
    statement_n
```

Rules and semantics

- `p_k=d_k` specifies both a formal parameter `p_k` and its default value `d_k`
- defaults values are optional (may be given for a subset of the parameters only)
- when a parameter has no matching argument in a call, it is bound to its default value

Example

The program

```
1  def foo(a, b = 2, c = 3):
2      return (a + c) / b
3
4  print(foo(2))
5  print(foo(4, c = 2))
6  print(foo(3, 5))
7  print(foo(c = 4, a = 8))
```

prints

```
2.5
3.0
1.2
6.0
```

Interpretation

default values are underlined

line	a	b	c
4	2	2	3
5	4	2	2
6	3	5	3
7	8	2	4

Outline

Problems in programs

Errors and Exceptions

- Syntax errors: the program is not an acceptable python program and cannot be executed
- Exceptions: errors detected during execution

Syntax error

Running

```
y = 5
x = 3 +/ y
```

prints

```
Traceback (most recent call last):
  File "error.py", line 2
    x = 3 +/ y
           ^
SyntaxError: invalid syntax
```

Exception

Running

```
y = 5
x = 3 +/ y
```

prints

```
Traceback (most recent call last):
  File "zero.py", line 3, in <module>
    z = x/y

ZeroDivisionError: division by zero
```

Exceptions

Handling exceptions

- normal behavior: an exception stops the program
- desirable behavior: fix the problem and continue
- mechanism
  - **`try`** something
  - if it does not work and induces an exception do something else

Example

```
try:
    answer = input('Enter an integer = ')
    x = int(answer)
except ValueError:
    print(answer,' is not an integer')
    x = 0
print(x)
```

Normal output

```
Enter an integer = 5
5
```

Exceptional output

```
Enter an integer = foo
foo  is not an integer
0
```

Handling Exceptions

try statement

- **try** is a compound statement which starts with a **try** clause
- followed by
  - a single **finally** clause
  - or at least one **except** clause with possibly an **else** clause and a **finally** clause

Short version

```
try:
  body_t
finally:
  body_f
```

Long version

```
try:
  body_t
except type_1:
  body_e_1
except type_2:
  body_e_2
  ...
else:
  body_el
finally:
  body_f
```

Handling Exceptions

```
try:
  body_t
except type_1:
  body_e_1
except type_2:
  body_e_2
  ...
else:
  body_el
finally:
  body_f
rest of the program
```

Semantics

- Python tries to execute `body_t`
- if this does not produce any exception, the execution continues through the **`else`** and then through the rest of the program
- if an exception of type `T` is raised
  - Python search for a matching type in the **`except`** headers in order (an empty type in a **`except`** matches any exception type)
  - if a matching type is found, the corresponding body is executed and then the rest of the program is executed
- the **`finally`** clause is always executed, even if an exception occurs in the **`except`** or **`else`** clause

Example

```
try:
    v = input('x = ')
    x = int(v)
    print(1/x)
except ValueError:
    print(v,'is not an integer')
except ZeroDivisionError:
    print('no inverse for',x)
else:
    print('ok')
finally:
    print('this is the end')
print('rest of the program')
```

Interactions

- if the user inputs `2.5`, she gets

```
2.5 is not an integer
this is the end
rest of the program
```

- if the user inputs `4`, she gets

```
0.25
ok
this is the end
rest of the program
```

- if the user inputs `0`, she gets

```
no inverse for 0
this is the end
rest of the program
```

Missing an exception

An exception is unhandled

- when it occurs in the **`try`** clause and is not matched
- when it occurs in a **`except`** clause
- when it occurs in a **`else`** clause
- when it occurs in a **`finally`** clause

and it is passed to the enclosing environment

Example

The following program

```
try:
    try:
        x = int('2.5')
    except ValueError:
        print('got it')
        print(1/0)
    except ZeroDivisionError:
        print('missed')
    finally:
        print('exiting')
except ZeroDivisionError:
    print('caught')
```

prints

```
got it
exiting
caught
```

Next Steps

1. Data structures in Python
2. Data manipulation in Python

Licence

Version

Last git commit: 2019-12-09
By: Fabrice Rossi (Fabrice.Rossi@apiacoa.org)
Git hash: ba0fa5483950d448f7a9210e4ac63ceabff8fb3f

Changelog

- November 2019: added exception handling
- October 2019: initial version