Script language: Python
Functions

Cédric Saule

Technische Fakultät
Universität Bielefeld

12. Februar 2016

Python

–

Functions

Functions

- Why functions/methods

Functions

- Why functions/methods
  - No code duplication
  - Factorizing
  - Structuring
  - No spaghetti code

Functions

- Why functions/methods
  - No code duplication
  - Factorizing
  - Structuring
  - No spaghetti code
- In Python: functions

Functions

- Why functions/methods
  - No code duplication
  - Factorizing
  - Structuring
  - No spaghetti code
- In Python: functions
- **Functions are objects!!!**

Writing a function

- A function must have
  - A name.
  - An interface (with no, one or several parameters).
  - Have a returned value (Default: None).
- Keyword: `def`

```
def Function_name (p1, p2, ..., pn):
    Instruction1
    Instruction2
    ...
    return returned_value
```

Exercise

Write a fonction 'reverse' that reverses a string and returns it.

Positional vs. keyword parameters

so far Positonal parameters
Order is important.

alternative Keyword parameters.

- Order is not important.
- Parameters are specified by keywords.

```
def El_Mamun(calif, sons, baggage_camels, costs, additional_costs):
    return (calif+sons)*baggage_camels*costs+additional_costs

El_Mamun(1,2,3,4,5)
El_Mamun(sons=2, calif=1,baggage_camels=3, additional_costs=5, costs=4)
```

Optional parameters

- ipython: `?range`
  What conclusions may you draw from the online help?

Optional parameters

- ipython: `?range`
  What conclusions may you draw from the online help?
- ```
  def power(x,y=2):
      return x ** y
  ```
  ```

Optional parameters

- ipython: `?range`
  What conclusions may you draw from the online help?
- ```
  def power(x,y=2):
      return x ** y
  ```
- Optional parameters must always be at the end of the declaration, why?
  ```

Any number of parameters

Python allows the passage of the two forms (positional and keyword); an unspecified number of parameters are proccessed.

- 'positional' parameters

```
def fct_1(*param):
print param
```

- 'keyword' parameters

```
def fct_2 (**param):
    print param
```

- Test both functions in a Python shell, what do you notice?

Documentation

- Python provides an easy way to document self-written functions.
- Used by the online help of the [i]python shell.

```
def test():
    "Function␣'test'␣is␣an␣example␣..."

?test()
```

Multiple returned values

It is a problem in lots of programming language, not in Python.

```
def power(n):
    return n*n, n*n*n

2_power_2, 2_power_3 = power(2)
3_power_2_AND_3_power_3 = power(3)
```

Exercise

Write a function that returns a sequence in FASTA format in which the size of the sequence in the output has to be given as a parameter.

- What is a FASTA sequence? What does it define?
- 'Positional' or 'Keywords' parameters - What is the best ?

Lexical scoping [1]

- Local vs. global variables.

```
def f(a):
    return a

a  = 100
f(111)
a
```

- Read access to global variables.

```
def g():
    print s
s = 1
g()
```

Lexical scoping [2]

- Try this:

```
def h():
    s = 2
    print s
s = 1
h()
s
```

Lexical scoping [2]

- Try this:

```
def h():
    s = 2
    print s
s = 1
h()
s
```

- Write access to global variables.

```
def h():
    global s
    s = 2
    print s
```

Local functions

- Functions in a function are called 'local' functions.
- Valid only within the namespace of a function.

```
def global_function(n):
    def locale_function(n):
        return n*n

    return locale_function
```

Anonymous functions

- Keyword `lambda` - λ function.
- But simply to define the advantages of a 'real' function.

```
> power = lambda x,y:x**y
> power(2,2)
> power(2,3)
```

- No control structure.
- As efficient as other structures.
- Direct calling:

```
> (lambda x,y:x**y)(2,2)
```

Exercise

Write a λ function that takes a list of numbers in argument and returns their sum!

Python

–

Errors and exceptions

Semantic != Syntax

Even if a code is semantically right, that does not mean it is syntactically right. The reverse statement is also true.

Syntax error Right semantic evaluation, but not syntactically equivalent to the Python guidelines

```
> while True print 'Hello␣world'
  File "<ipython-input-5-f4b9dbd125c8>", line 1
    while True print 'Hello␣world'
                     ^
SyntaxError: invalid syntax
```

Semantic != Syntax

Exception Syntactically right, but is not semantically equivalent to logical prerequisites.

```
> 10 * (1/0)
ZeroDivisionError: integer division or modulo
  by zero

> 4 + spam*3
NameError: name 'spam' is not defined

> '2' + 2
TypeError: cannot concatenate 'str' and
  'int' objects
```

Handling errors

```
try:
  <INSTRUCTIONS>
except <ERROR>:
  <INSTRUCTIONS>
[except <ERROR>:
  <INSTRUCTIONS>
.
.
.]
[else:
  <INSTRUCTION>]
```

try Unsafe block

except Error trap.<ERROR>:

- **IOError as io** Errors available as an object in further treated part.
- **(RuntimeError, TypeError)** treats all errors of the specified type.
- **ARGUMENTLESS** Handle each error !

else Runs when no error occurred.

Handling errors

```
>for arg in sys.argv[1:]:
> try:
>    f = open(arg, 'r')
> except IOError:
>    print 'cannot␣open', arg
> else:
>    print arg, 'has', len(f.readlines()), 'lines'
>    f.close()
```

It is better to write in the `try` block only the code that can raise an exception.

Throwing exceptions

We can throw exceptions by using `raise`.

```
> try:
>  raise NameError('HiThere')
> except NameError:
>  print '\"An␣exception␣flew␣by!\"'
>  raise

"An␣exception␣flew␣by!"
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

Forcing at each state

The `finally` block is always executed. This block is referred to a *clean-up action*.

```
> def divide(x, y):
    try:
      result = x / y
    except ZeroDivisionError:
      print "division␣by␣zero!"
    else:
      print "result␣is", result
    finally:
      print "THX␣for␣using␣me"
```

Exercise – Errors hierarchy

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
```

Create a function that expects a list as input, an object and returns `True` when the object is located and otherwise a `LookupError` with the corresponding error message (parameters) to generate the list. Pay attention to the

correct *exception handling* used in the `index()` method!