9. Scripting Languages and Python

Slides:

- *25 Scripting languages*
- *26 Python*
- *26b Python functions*
- *27 Advanced Python*
- *28 Python under-the-hood*
- *29 Scripting in Python*

Common Characteristics

- Both batch and interactive use, is compiled/interpreted line-by-line
- Economy of expression, concise syntax, avoid top-level declaration
- Simple default scoping rule, that can be explicitly overridden
- Flexible dynamic typing
- Easy access to system facilities
- Sophisticated pattern matching and string manipulation thanks to Regular Expressions
- High level data types: associative arrays implemented as hash tables.

Scripting languages domain

- Shell: manipulate files, command and arguments, glue for other languages.
- Text processing and report generation: RegEx, logging.
- Mathematics, physical and statistics: Mathematica and MatLab, R and S.
- “Glue” languages (Perl) and general-purpose scripting (Python).
- Extension languages (Tlc): allow user to create new commands, generally using existing commands as primitives.

Python

Key notions

- The block structure is based on indentation. For that reason if you want to split a command on multiple lines, you have to use a *newline* (`\`) at the end of the line.
- Variable types don't need to be declared, is dynamically typed.
- The first assigned to a variables declares it.
- Assignment for reference, not copy, like Java.

Syntax:

- Assignment `=` and comparison `==`. Multiple assignment: `x, y = 2, 3`
- Names are case sensitive, can contain letters, numbers, and underscores and cannot start with a number. Function starting with underscore are considered private, even if with the visibility is always public. Variables and functions must be in lower case.
- Operation on numbers `+`, `-`, `*`, `/` and `%` as usual.
- Integer division return a float (`5/2 == 2.5`) but can be forced to return int: `5//2 == 2`.
- Strings both with double-quotes (`"`) and single-quotes (`'`), double-quotes string can contain single-quotes (`"for'example"`), triple double-quotes can contain both (`"""a'new"example"""`) and also stand on multiple lines.
- Print with `print()`, `%` for string formatting
- Logical operators are words, not symbols: `and`, `or`, `not`.
- Boolean comparisons with `==`, `!=`, `<`, `<=`, etc. `is` to check if is the same object (checks the reference). Boolean are `True` and `False`. Conditional expression are subject to lazy evaluation.
- `if`, `while` and `for` don't need parentheses but the condition must end with a comma (`:`) and the body stands always on a new line, indented. In python exists only the for-each, the classic for is implemented in that way: `for x in range(n)`.
- Comments with `#`, DocString right under the function or class declaration, inside triple double-quotes: first line is one-line summary of the function, the other lines are an extended description of the function.
- `in` operator to check if a value is inside a collection (list, string, etc.).
- Strings, tuples and lists concatenation with `+` (creating a new one).

Conventions

- snake_case for variables and functions, CamelCase for classes
- spaces around operators and after commas
- two blank lines to separate functions, one for logical sections inside functions
- in comments `do_something() # two space before and one after the sharp`.

Data structures

- Tuples are defined using parentheses `(23, 'abc', (2,3), 4.34)`, lists using square brackets `[23, "abc", 4.34]`. Both can contain values of different types. Tuples are immutable (and faster), list mutable (and slower). Conversion between tuple and list with `l = list(t)` and `t = tuple(l)`.
- Sets: `{'apple', 'orange', 'pear'}`. Duplicates are automatically removed. Have mixed type, indexing not supported.
- Dictionaries: `d = {'name':'Mark', 'age':23}`. Keys must be unique and hashable. Add element with `d['city'] = "Pisa"` and remove with `del d['age']`. Other operations: `d.get('age', 0)` and `d.keys()`.

Operations on list

- Negative lookup, count from right starting with –1: `l[n-1] == l[-1]`.
- Subset: `l[2:5]`. Can be omitted if equals to beginning or end: `l[:5] == l[0:5]`, `l[2:] == l[2:-1]`. Copy the list with `t[:]`.
- Add elements with `l.extend(['a', 'b', 'c']` and `l.append('d')`. Remove with `l.remove('c')` or `del l[2]`.
- IndexOf with `index`
- `count`
- `reverse`
- `sort`
- List comprehensions: `[ expression for name in list if filter ]`

Classes and functions

- Class definition: `class MyClass:`
- Constructor: `def __init__(self, param):`
- Function definition: `def my_function(self, param):` (self is omitted outside a class)

Private methods

There are no private methods of a class, butfor convention functions starting with underscore (`_my_prv_fun`) are treated as non-public part of the API, even if is possible to call them.

Name mangling

To avoid clashes between class-private names and names defined by subclasses, all the functions starting with double underscore (`__myfun`) are textually preponed by the classname preponed by an underscore (`_classname__myfun`).

Special methods

`__str__(self)` is the equivalent of the `toString` in Java.

`__iter__(self)` returns an iterator for the class (for example a collection).

Operators:

Operator	Class method
-	`__sub__(self, other)`
+	`__add__(self, other)`
*	`__mul__(self, other)`
/	`__truediv__(self, other)`
+ (unary)	`__neg__(self)`
- (unary)	`__pos__(self)`
==	`__eq__(self, other)`
!=	`__ne__(self, other)`
<	`__lt__(self, other)`
>	`__gt__(self, other)`
<=	`__le__(self, other)`
>=	`__ge__(self, other)`

Static method

A method is static if has the `@staticmethod` decorator in the line over the function definition.

Similarly to static method there are class methods (`@classmethod`), that have a first parameter `cls` that represent the class on which is called (`class.my_classmethod()` invokes `my_classmethod(cls)` passing "class" as value of cls).

Generators

Generators are functions that returns data with the `yield` statement. Each time the `next()` is called the generator resumes where it left-off, keeping all the data values and remembering which statements were already executed. The `__iter__()` and `next()` methods are created automatically. Indeed, a function is an object (of class function), and hence can have methods.

Classes

Classes in Python are like structs in C: contains attributes (functions are object, so attribute too). For that reason we cannot have function overloading (same function with different number of args) since we will have more than one attribute with the same name.

Class inheritance

Python support multiple inheritance: `class SubClass(Super1, ..., SuperN):`.

Lambdas

```
lambda x : x + 1
```

Functional Programming

Map

`map(func, iter1 [,iter2 [...]])` returns an iterator: laziness.

It is possible to obtain a list with `list(map(func, *iterables))`.

Using a lambda: `map(lambda x : x + 1, range(4))`.

On multiple lists: `list(map(lambda x, y : x+y, range(4), range(10)))` (will cut `range(10)` in `range(4)` because the first list has only 4 elements).

Zip

`zip(iter1 [,iter2 [...]])` return an iterator that tuples together the element of all the lists in the same position.

`list(zip(['a', 'b', 'c'], [1, 2, 3])) == [('a', 1), ('b', 2), ('c', 3)].`

Filter

`filter(func, iterable)` return an iterator that yields the values in the iterable for whom the function is true. In func is None return only the elements evaluated as True (not False, not None, not 0 ...).

functools

Higher-order functions on iterables. E.g. reduce(function, iterable[, initializer])

itertools

Functions for creating iterators:

- `count(10) --> 10 11 12 13...`
- `cycle('ABCD') --> A B C D A B...`
- `repeat(10, 3) --> 10 10 10`
- `takewhile(lambda x: x < 5, [1, 4, 6, 4, 1]) --> 1 4`
- `accumulate([1, 2, 3, 4, 5]) --> 1 3 6 10 15`

Decorators

A decorator modify a function, method or class definition. Decorator is passed the original object is being defined and returns the modified object.
Decorators exploit Python higher-order features:

- Passing functions as argument
- Nested definition of functions
- Returning functions

Definition:

```
def my_decorator(func):  # function as argument
  def wrapper():  # defines an inner function
    ...  # as before
  return wrapper  # returns the inner function
```

Usage:

```
@my_decorator
def my_function():
  ...  # body of the original function
```

Functions with parameters using `*args` and `**kwargs`:

```
def do_twice(func):
  def wrapper_do_twice(*args, **kwargs):
    func(*args, **kwargs)
    func(*args, **kwargs)
  return wrapper_do_twice

@do_twice
def echo(str):
  print(str)

>>> echo("Hello world!")

Hello world!
Hello world!
```

Example: measuring run time

```
def timer(func):
  """Print the runtime of the decorated function"""
  @functools.wraps(func)
  def wrapper_timer(*args, **kwargs):
    start_time = time.perf_counter()
    value = func(*args, **kwargs)
    end_time = time.perf_counter()
    run_time = end_time - start_time
    print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
    return value
  return wrapper_timer
```

Exception Handling

```
def divide(x, y):
  try:
    result = x / y
  except ZeroDivisionError:
    print "Error: division by zero"
      else:
    print("Result: " + result)
  finally:
    print("Executing finally clause.")  # always executed
```

Namespaces and Scopes

The namespace is implemented as a dictionary.

- Built-in names (pre-defined functions, etc.) are created at interpreter startup.
- Global names of a module are created when the module definition is read.
- Local names are created when the function is invoked and deleted when it completes.
- Names of classes

Scope is determined statically but is used dynamically.

During the execution at least this 3 namespaces are accessible, in that order:

1. local names
2. [enclosing functions, containing non-local and non-global names]
3. next-to-last scope with module's global names
4. outermost scope with built-in names

Non-local variables can be accessed using `nonlocal` or `global`.

For each statement (field or function) in a class a new namespace is created where goes all the names introduced in the statement.

The built-in function `dir()` returns a sorted list of strings containing all names defined in a module.

Important: while functions and classes introduces a new scope, if statements, for loops, while loops, etc do not. If you declare a variable inside a loop, it is visible also outside.

Functions' parameters

Variables passed as parameters are copied, but variables are just reference to objects, so every parameter is a reference. That means that if a mutable object is passed, caller will see changes (we can modify a list passed as parameter without returning a new modified list).

Parameter of form `*args` captures excess positional args: that allow to call functions with a non-fixed number of arguments. `args` is seen as a tuple.

`**kwargs` captures all excess keyword arguments. E.g. in `my_fun(5, 8, p=2)` 5 and 8 are args, p is a keyword argument. `kwargs` is a tuple of tuple: `((karg1, val1), (karg2, val2), ...)`.

Garbage Collector

The memory is managed by a reference counting + mark&sweep cycle collector scheme. Reference counting means that each object has a counter storing the number of references to it. When becomes 0 can be destroyed.

Pros are a simple implementation and the fact that the memory is reclaimed ASAP, without freezing the execution.

Cons are the additional memory requirement and the fact that cyclic structures in the garbage cannot be identified (thus the need of mark&sweep)

Global Interpreter Lock (GIL)

The CPython interpreter ensures that only one thread at time executes Python bytecode, using the Global Interpreter Lock.

The current thread must hold the GIL before it can safely access Python objects. This makes the object model implicitly safe the concurrent access.

However, the GIL degrade performances, adding a significant overhead on multicore hardware. For that reason, the GIL should be released when doing some computationally-intensive tasks or I/O.