Python

Advanced Topics in Programming Languages:
Seminary

Alessandro Sestini
6226094

What we'll see

- **Python Introduction**:

  what, where and why, dynamic typing, general aspects…

- **Functional Python**:

  functions, lambda, high order functions, recursion …

- **Python Semantic**:

  strict vs lazy, generators, strict vs lazy vs functional, …

Introduction

Python

- Python is an **interpreted and interactive high-level** language for general purpose programming, no compilation needed
- It was created by **Guido van Rossum** at “Centrum Wiskunde & Informatica” in Netherlands and first released in 1992
- It supports various programming paradigms: structured, object-oriented, imperative, procedural and **functional**

UNIVERSITÀ DEGLI STUDI FIRENZE

Python

"...In December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office ... would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus)."

– Guido van Rossum

UNIVERSITÀ DEGLI STUDI FIRENZE

Python

- It was designed to be an **highly readable** language, simple and consistent :

  “There should be one - and preferable only one - obvious way to do it”

- It uses **implicit and dynamic typing** (though now is a little bit of static) and dynamic name resolution - **late binding**

- But it is also a **strongly typed** language: a **value** can’t change his type (and the type combinations must be consistent)

Python - Why

- As we see Python is a **very simple way** to write codes, but is a real programming language
- Due to its interpreted nature and its dynamic and implicit typing, it's a very **"easy language"**
- It can be used for **prototyping** real programs, like HCI projects
- Or it can be used **to test some machine learning** models and for the preprocessing step of datasets

Python - Useful Aspects

- **Indentation**: Python uses **whitespaces** to delimit control flow blocks

```
def foo(x):
    if x == 0:
        bar()
        baz()
    else:
        qux(x)
        foo(x-1)
```

- It uses a ***call-by-object reference*** system, that is:

x = 2

translates to "(generic) name x receives a reference to a separate, dynamically allocated object of numeric (int) type of value 2"

- Later in the code we can write *x = "String"*, because *x* is only a **reference** to a location in memory, it doesn't have a type
- Due to the dynamic typing, in Python **values carry type**, not variables

Python - Typing

- As we saw earlier, Python provides **implicit and dynamic** typing; this **combination** leads Python to be an interactive and fast language

- The user **doesn't have to write types**, and the type checker doesn't look at the code **until the coder executes it**: recall that Python is **Strongly Typed**, so it must check the types

- This features let the user write code in a very simple way, without worrying if a statement is badly typed - **it doesn't raise any error**

- Only when the interpreter reaches the "bad" statement at **run-time**, it will interrupt the execution and will raise the error

Python - Typing

- So it's obvious how these features highlight the *"smartness and interactivity"* of the language:

    - Even if we do not explicit the types of the values, since the inspection is done at run-time, the type checker **learns the context step by step** and it must infere the types **only when it needs them**

    - If we have an explicit and static typing, the coder must put **a lot of affords** writing the statements of his programs, but the type checker **already knows the types** and it's simpler for it to verify the type system

    - If we have an implicit and static typing, the type checker must infere the type **without any "active" context, so it's more difficult**

Python - Typing

- Example:

```
>>> def function():
...     string = "Dynamic" #We don't write types
...     return string + 9 #The checker doesn't raise the error
...
>>> a = 3
>>> a = "Implicit"
>>> function()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in function
TypeError: cannot concatenate 'str' and 'int' objects
```

- Although, for complex program we can't be sure of the *"complete safeness"* of the code, this is why Python is mostly used for prototyping

Python - SubTyping

- In the concept of Object Oriented, an important feature of Dynamic Typing is **Duck Typing**

- The idea is that it doesn't actually matter what type my data is - just whether or not **I can do what I want with it** - it's a sort of **structured subtyping**

"If it walks like a duck and it quacks like a duck, then it must be a duck"

- We will not go into this detail in the seminary

Python - Mutable vs Immutable

- Python is mainly Object Oriented, so everything in it is an ***Object,*** and every variable holds a reference to it

- An object's type is **defined at *run-time*** and once it's set, it can't be changed. However, if it's **mutable**, its **state can be changed**

- **Immutable** built-in types: int, float, bool, str, tuple
  Mutable built-in types: list, set, dict

Python - Mutable vs Immutable

Examples:

```
>>> x = 10
>>> y = x
>>> id(x) == id(y)
True
>>> x = x + 1
>>> id(x) == id(y)
False

>>>
>>> m = list([1,2,3])
>>> n = m
>>> id(m) == id(n)
True
>>> m.pop()
3
>>> id(m) == id(n)
True
```

Python - Collections

- Python **built-in collections** are very important, as we'll see in the Strict vs Lazy section

- This types have **built-in methods** like *__getitem__(i)* or *__setitem__(i,x)* (the immutable once doesn't have *setitem*)

- Type of collections: list, string, tuple, dict, …

```
>>> tuple()
()
>>> tuple("abc")
('a', 'b', 'c')
>>>
>>> dict()
{}
>>> dict([("a", 14), ("b", 13)])
{'b': 13, 'a': 14}
```

Python - Iterators

- To be **iterable** an object must have the special method *__iter__()*; an iterable can return an **iterator**, another object that is able to get all the elements of the collection

- An iterator must have 2 characteristics: a special method ***__next__()*** and the ability to raise the exception ***StopIteration*** when there are no more elements

- An iterator can iterate **through the elements of a collection**, and usually is used automatically within a *for cycle*:

```
>>> for line in open('file.txt'):
...     print('Iterator')
```

Python - Other Generals

- Assignment uses **=** and comparison uses **==**. For numbers, + - * / % act as expected
  - Special use of + for **string concatenation**
  - Special use of % for **string formatting** (as with printf in C)
- Logical operators are **words** (and, or, not) and not symbols. The basic printing command is *print()*
- The **first assignment** to a variable **creates it**
- It is **case sensitive**

UNIVERSITÀ DEGLI STUDI FIRENZE

Functional Python

Functional Python

- In a pure functional paradigm we only create programs **evaluating functions**, returning functions and using functions as parameters
- Python is **not a pure functional** language, because uses some non-functional features internally to let coders use functional interface of some sort
- **Lambda expressions** and **efficient recursion**

Functional - Lambda Abstraction

- Lambda calculus:

$$\lambda x.x * 2$$

- Python:

```
>>> def double(x):
...     return x*2
...
```

```
>>> lambda x : x*2
```

Functional - Lambda Application

- Lambda calculus:

$$(\lambda x.x * 2)2$$

- Python:

```
>>> double(2)
4
```

```
>>> (lambda x : x*2)(2)
4
```

Functional - Lambda

- In lambda calculus we can assign a name to a function:

$$square = \lambda x.x * x$$

- Instead, in Python we can assign **reference** to a lambda:

```
>>> square = lambda x : x*x
```

Functional - Lambda Examples

- Boolean functions in lambda calculus:

$$true = \lambda x.\lambda y.x$$
$$false = \lambda x.\lambda y.y$$
$$and = \lambda x.\lambda y.xyfalse$$

- Python:

```
>>> true = lambda x: lambda y: x
>>> false = lambda x: lambda y: y
>>> and_func = lambda x: lambda y: x(y(false))
>>> result = and_func(true)(false)
>>> result(2)(3)
3
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Functional - Typing

- Due to Python dynamic typing, when we define a lambda expression the checker only says to you that a **generic *"function"* has been wrote**: it doesn't know if the statement inside the body is well or badly typed, and **it doesn't care**

```
>>> square = lambda x : x*x
>>> type(square)
<type 'function'>
```

- So, if a lambda has a "badly typed" body, the language will raise the **error only during run-time** (recall that Python is **Strongly Typed**)

Functional - Typing

- **When we execute** the code the checker learns the context step by step and, when it arrives to execute the body of the lambda, **it checks if it's well typed**

- If there's an error, it will be generated when the **interpreter reaches** the "bad" statement:

```
>>> funct = lambda : "Error" + 2
>>> funct()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <lambda>
TypeError: cannot concatenate 'str' and 'int' objects
```

Functional - High Order Functions

- A major feature of functional programming is **High Order Function**: functions can return functions and can accept other functions as arguments

- In Python, lambda expressions are **First Class Citizens**: they are treated like objects

- This feature leads to **partial evaluation** and **unary functions**

Functional - High Order Functions

- In lambda calculus functions are **unary** and we can use **partial evaluation**:

$$mult = \lambda x.\lambda y.x * y$$
$$multY = mult(2)$$
$$result = multY(4)$$

- Same in Python:

```
>>> mult = lambda x : lambda y : x*y
>>> multY = mult(2)
>>> result = multY(4)
>>> print(result)
8
```

Functional - High Order Functions

- Same as:

```
>>> mult = lambda x : lambda y : x*y
>>> result = mult(2)(4)
>>> print(result)
8
```

- But we can also define a 2 inputs function:

```
>>> mult = lambda x,y : x*y
>>> result = mult(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: <lambda>() takes exactly 2 arguments (1 given)
>>> result = mult(2,4)
>>> print(result)
8
```

Functional - Recursion

- In lambda calculus, for recursion we had to introduce the **fix** construct

- In Python, we can assign a reference to the lambda and call it **recursively** - then the **type checker** will verify that the reference point to a real function:

```
>>> fib = lambda n: n if n < 2 else fib(n-1) + fib(n-2)
>>> fib(10)
55
```

Functional - Recursion

- But we can also define a **fixed point combinator** like the fix construct in lambda calculus for **recursion**:

```
>>> Z = lambda f: (lambda x: f(lambda y: x(x)(y)))(lambda x: f(lambda y: x(x)(y)))
>>> fac = lambda f: lambda n: (1 if n<2 else n*f(n-1))
>>> Z(fac)(6)
720
```

- This is an implementation of the **Z Combinator**:

$$fix = \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))$$

Functional - Why

- So Python provides a useful **interface for functional programming** (built internally with object-oriented constructs)

- The functional paradigm provides an **even simpler way** to write programs, because we **only use functions**, always using implicit and dynamic typing

- But Python is **not** a pure functional language, so we can use the lambda constructs in **combinations** with other object-oriented features (like list-comprehensions)

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy

- Python is **Strict** by default:

```
>>> def noreturn(x):
...     while True:
...             x = -x
...     return x
>>> 2 in [2, 4, noreturn(5)]
```

- In this example, even if *noreturn(5)* is useless, it will always be evaluated and the interpreter will enter in a infinite loop

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy

- While the **conditional is lazy:**

```
>>> def cond():
...     if True:
...             return 2
...     else:
...             return noreturn(5)
...
>>> cond()
2
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy

- Therefore, the semantic of the lambda expressions is **Strict**. If we take the Omega function that we have defined in lambda calculus:

$$\Omega = (\lambda x.xx)(\lambda x.xx)$$

- We know that if we apply this to an other function, in a strict strategy we will enter in a **infinite loop**

$$(\lambda z.(\lambda y.y))(\lambda x.xx)(\lambda x.xx)$$

- Because we evaluate all the statements, even if these are useless, so we **never reach the end**

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy

- The **same** happens in Python:

```
>>> omega = lambda x: x(x)
>>> func = lambda y: "We reach the end!"
>>> (func)((omega)(omega))
RuntimeError: maximum recursion depth exceeded
```

- But Python lets coders use some **Lazy evaluations**

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy - *range()*

- In the previous slide we saw the **collections**: string, list, tuple, dict. We saw also that a collection can be mutable or immutable
- And we also saw the **iterators** (and iterables)
- An easy way to create a list and iterate it is the method ***range****([start], stop, [step])*

```
>>> range(5)
[0, 1, 2, 3, 4]
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy - Generators

- In Python 2, the *range()* function has a **strict semantic**, that is, when we call the function, **all the elements are created**
- There's a special type of iterator that has a **Lazy Semantic**, called **Generator**
- Like with the iterator objects, with the generators we can use the method *next()*, although they don't have the method *__getitem__*
- There are 2 sub-type of generators: **Generator Expressions** and **Generator Functions**

Strict vs Lazy - Generator Expressions

- The simplest type of a Generator Expression is the **list-comprehension**; this construct can return either an iterable or a generator:

```
>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> (x for x in range(10))
<generator object <genexpr> at 0x10373caa0>
```

- In the first case, we create a strict list, instantiating all the elements; instead in the second case we create a generator that **doesn't create any elements** at the beginning, but **only when the element is needed**

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy - Generator Expressions

- Example:

```
>>> strict = [x**2 for x in range(50,100)]
>>> strict
[2500, 2601, 2704, 2809, 2916, 3025, 3136, 3249,
3364, 3481, 3600, 3721, 3844, 3969, 4096, 4225,
4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329,
5476, 5625, 5776, 5929, 6084, 6241, 6400, 6561,
6724, 6889, 7056, 7225, 7396, 7569, 7744, 7921,
8100, 8281, 8464, 8649, 8836, 9025, 9216, 9409,
9604, 9801]
>>>
>>> lazy = (x**2 for x in range(50,100))
>>> lazy
<generator object <genexpr> at 0x10373cb40>
>>> lazy.next()
2500
>>> lazy.next()
2601
```

Strict vs Lazy - Generator Expressions

- Another simple way to create a generator is the function ***xrange()***, that is the lazy version of *range()* (in Python 3 *range()* is already lazy, no need for *xrange()*):

```
>>> strict = range(5)
>>> strict
[0, 1, 2, 3, 4]
>>> lazy = xrange(5)
>>> lazy
xrange(5)
>>> lazy[3]
3
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy - Generator Functions

- We can also create **custom generators through functions**
- This kind of object (that is a real generator) can be used through the ***next*** method (like the others generators)
- The keyword to define a generator function is ***yield***, that must be used instead of *return*

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy - Generator Functions

- Example:

```
>>> def myrange(a,b):
...     while b > a:
...             yield a
...             a += 1
...
>>> mylist = myrange(1,10)
>>> mylist
<generator object myrange at 0x10373cb90>
>>> mylist.next()
1
>>> mylist.next()
2
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy - Fibonacci Sequence

- Strict Fibonacci:

```
>>> def fib_strict(n):
...     list = []
...     i = 0
...     while i < n:
...             if i <= 1:
...                     list.append(1)
...             else:
...                     list.append(list[i-1] + list[i-2])
...             i += 1
...     return list
...
>>> strict = fib_strict(10)
>>> strict
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy - Fibonacci Sequence

- Lazy Fibonacci:

```
>>> def fib_lazy(n):
...     i = 0
...     a = 1
...     b = 1
...     while i < n:
...             yield a
...             tmp = a
...             a = b
...             b = tmp + b
...             i = i + 1
>>> lazy = fib_lazy(10)
>>> lazy
<generator object fib_lazy at 0x10373cc30>
>>> for x in lazy:
...     print(x, end = " ")
...
1 1 2 3 5 8 13 21 34 55
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy - Fibonacci Sequence

- Strict Fibonacci and Lazy Fibonacci in time (for n = 100000):

```
Time to create and evaluate strict list:  0.46762800216674805
Time to create lazy list:  1.9073486328125e-06
Time to evaluate lazy list:  0.15849709510803223
Time to create and evaluate lazy list:  0.15849900245666504
Size in byte for strict list:  824464
```

Strict vs Lazy - Fibonacci Sequence

- With this lazy strategy we can manage **infinite lists**:

```
>>> def inf_fib():
...     a,b = 1,1
...     while True:
...             yield a
...             tmp = a
...             a = b
...             b = tmp + b
>>> lazy = inf_fib()
>>> lazy
<generator object inf_fib at 0x10596bb90>
>>> for x in lazy:
...     print(x)
...     if x > 55:
...             break
1 1 2 3 5 8 13 21 34 55
```

- **This is impossible** in a strict strategy, because when we initialize the list, we will enter in a **infinite loop**

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy - Why

- So a lazy strategy evaluates an element **only when needed**, and with this semantic we are able to create **infinite collections**

- We can use the lazy evaluation when:
  - We have to manage **very big data structures**
  - We have **a lot of trivial computations**

- But we shouldn't use a lazy strategy when we have to **compute an element more times**

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy - Why

- Practical example:

"I often face the problem of needing to migrate data from one website into another. Some of the sites I'm migrating have over a decade of history, gigabytes of content. Using the generator-based migration tool called "collective.transmogrifier" (https://pypi.python.org/pypi/collective.transmogrifier) I can read data from the old site, make complex, interdependent updates to the data as it is being processed, and then create and store objects in the new site all in constant memory"

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy vs Functional

- Python 3 provides two built-in functions that **duplicate the features of generator expressions** and act like the list-comprehension

- This functions are usually used **in combination of lambda expressions**

- **Map**: applies a function to all the elements of a list

- **Filter**: applies a boolean function to all the elements of a list, returning all the original elements that satisfy the function

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy vs Functional

- Examples:

```
>>> map = map(lambda x: x**2, [2,3,4])
>>> map
<map object at 0x1045c8cc0>
>>> list(map)
[4, 9, 16]
>>>
>>>
>>> filter = filter(lambda x: x%2 == 0, range(10))
>>> filter
<filter object at 0x1045c8cf8>
>>> list(filter)
[0, 2, 4, 6, 8]
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Strict vs Lazy vs Functional

- Example with lazy collections:

```
>>> map = map(lambda x: x**2, lazy_fib(10))
>>> list(map)
[1, 1, 4, 9, 25, 64, 169, 441, 1156, 3025]
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Python - Conclusions

- In this seminary we saw an introduction to the Python programming language
- Especially its functional paradigm, and how we can create functional programs with lambda expressions (like we saw with lambda calculus)
- And we also pointed out the difference between a strict evaluation and a lazy strategy, and how we can use infinite lazy generators
- In conclusion, we saw that Python is a simple programming language, designed to be easy to use and to have simple constructs that provide different programming paradigms, and it uses the combination of dynamic and implicit typing to enhance its interactive nature

UNIVERSITÀ DEGLI STUDI FIRENZE

Python - Conclusions

- Nowadays Python is used for a lot of things, mainly because it's becoming an increasingly powerful and fast language
- Many libraries have been created that improve the power of Python and *"if you want to do something in Python, chances are pretty good someone else already has, and you don't need to start from scratch"*
- The PyPy project aims to speed up Python as a whole (and is doing a great job of it)
- Disney uses Python to help power their creative process (CGI projects)
- Mozilla uses Python to explore their extensive code base and releases tons of open source packages built in python

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Python

Advanced Topics in Programming Languages:
Seminary

Alessandro Sestini
6226094