Advanced Programming

Federico Bruzzone

20 settembre 2022

Indice

1 Python

1.1 Python's whys & hows

1.1.1 What is Python

Python is a general-purpose high-level programming language

- it pushes code readability and productivity;
- it best fits the role of scripting language.

Python support multiple programming paradigms

- imperative (function, state, ...);
- object-oriented/based (objects, methods, inheritance, ...);
- functional (lambda abstractions, generators, dynamic typing, ...).

Python is

- interpreted, dynamic typed and object-based;
- open-source.

1.1.2 How to use Python

We are condidering Python 3+

- version > 3 is incompatible with previus version;
- version 2.7 is the current version.

A python program can be:

- edited in the python shell and executed step-by-step by the shell;
- edited and run through the iterpreter.

1.2 Overview of the Basic Concepts

1.2.1 Our first Python program

```
1 SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
2             1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
3 def approximate_size(size, a_kilobyte_is_1024_bytes=True):
4   ''' Convert a file size to human-readable form. '''
5   if size < 0:
6     raise ValueError('number must be non-negative')
7   multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
8   for suffix in SUFFIX[multiple]:
9     size /= multiple
10    if size < multiple:
11      return '{0:.1f} {1}'.format(size, suffix)
12    raise ValueError('number too large')
13
14 if __name__ == '__main__':
15   print(approximate_size(1000000000000, False))
16   print(approximate_size(1000000000000))
```

Listing 1: humanize.py

1.2.2 Declaring function

Python has function

- no header files à la C/C++;
- no interface/implementation à la Java.

```
1 def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

1. **def**: function definition keyword;
2. **approximate_size**: function name;
3. **a_kilobyte_is_1024_bytes**: comma separate argument list;
4. =**True**: default value.

Python has function

- no return type, it always return a value (**None** as a default);
- no parameter types, the interpreter figures out the parameter type.

1.2.3 Calling Functions

Look at the bottom of the *humanize.py* program

```
1 if __name__ == '__main__':
2   print(approximate_size(1000000000000, False))
3   print(approximate_size(1000000000000))
```

2 in this call to **approximate_size()**, the **a_kilobyte_is_1024_bytes** parameter will be **False** since you explicitly pass it to the function;

3 in this row we call **approximate_size()** with only a value, the parameter **a_kilobyte_is_1024_bytes** will be **True** as defined in the function declaration.

Value can be passed by name as in:

```
1 def approximate_size(a_kilobyte_is_1024_bytes=True, size=1000000000000)
```

Parameters' order is not relevant

1.2.4 Writing readable code

Documentation Strings A python function can be documented by a documentation string (docstring for short).

"' Convert a file size to human-readable form. "'

Triple quotes delimit a single multi-string

- if it immediatly follows the function's declaration it is the doc-string associated to the function;
- docstrings can be retrieved at run-time (they are attributes).

Case-Sensitive All names in Python are case-sensitive

1.2.5 Everything is an object

Everything in Python is an object, functions included

- **import** can be used to load python programs in the system as modules;
- the dot-notation gives access to the the public functionality of the imported modules;
- the dot-notation can be used to access the attributes (e.g., the **__doc__**)
- **humanizeàpproximate_size.__doc__** gives access to the docstring of the **approximate_size()** function; the docstring is stored as an attribute.

1.2.6 Everything is an object (Cont'd)

In python is an object, better, is a first-calss object

- everything can be assigned to a variable or passed as an argument

---

```
1 h1 = humanize.approximate_size(9128)
2 h2 = humanize.approximate_size
```

- **h1** contains the string calculated by **approximate_size(9128**;
- **h2** contains the "function" object **approximate_size()**, the result is not calculated yet;
- to simplify the concept: **h2** can be considered as a new name of (alias to) **approximate_size**.

1.2.7 Indenting code

No explicit block delimiters

- the only delimiter is a column (':') and the code indentation;
- code blocks (e.g., functions, if statements, loops, ...) are defined by their indentation;
- white spaces and tabs are relevant: use them consistently;
- indentation is checked by the compiler.

1.2.8 Exceptions

Exceptions are Anomaly Situations

- C encourages the use of return codes which you check;
- Python encourages the use of exceptions which you handles.

Raising Exceptions

- the **raise** statement is used to rise an exception as in:

```
1 raise ValueError('number must be non-negative')
```

- syntax recalls function calls: **raise** statement followed by an exception name with an optional argument;

- exceptions are relized by classes.

No need to list the exceptions in the function declaration handling Exceptions

- an exception is handled by a **try** ... **except** block.

```
1 try:
2   from lxml import etree
3 except ImportError:
4   import xml.etree.ElementTree as etree
```

1.2.9 Running scripts

Look again, at the bottom of the *humanize.py* program:

```
1 if __name__ == '__main__':
2   print(approximate_size(1000000000000, False))
3   print(approximate_size(1000000000000))
```

Modules are Objects

- they have a built-in attribute **__name__**

The value of __name__ depends on how you call it

- if imported it contains the name of the file without path and extension.

2 Primitive Datatypes & recursion in Python

Python's Native Datatypes

2.1 Primitive types

2.1.1 Introduction

In python **every value has a datatype**, but you do not need to declare it.

How does that work?

Based on each variable's assignment, python figures out what type it is and keeps tracks of that internally.

2.1.2 Boolean

Python provides two constants

- True and **False**

Operations on Booleans

Logic operations: *and or not*

Relational operators: == != < > <= >=

Note that python allows chains of comparisons

```
1 >>> x = 3
2 >>> 1<x<=5
3 True
```

2.1.3 Number

Two kinds of number: integer and floats

- no class declaration to distinguish them
- they can be distiguished by the presence/absence of the decimal point

```
1 >>> type(1)
2 <class 'int'>
3 >>> isinstance(1, int)
4 True
5 >>> 1+1
6 2
7 >>> 1+1.0
8 2.0
9 >>>type(2.0)
10 <class 'float'>
```

- **type()** function provides the type of any value or variable;
- **isinstance()** check if a value or variable is of a given type;
- adding an int to an yields another int but adding it to a float yields a float.

2.1.4 Operations on numbers

Coercion & size

- **int()** function truncates a gloat to an integer;
- **float()** function promotes an integer to a float;

- integers can be arbitrarly large;
- float are accurate to 15 decimal places.

Operators (just a few)

+ -

* **

/ // %

2.2 Collection

2.2.1 Lists

A python list looks very closely to an array

- direct access to the members through [];

```
1 >>> a_list = ['1', 1, 'a', 'example']
2 >>> type(a_list)
3 <class 'list'>
```

But

- negative numbers give access ti the members backwards, e.g., **a_list[-2] == a_list[4-2] == a_list[2]**;
- the list is not fixed in size;
- the members are not homogeneous.

2.2.2 Lists: Slicing a List

A slice og a list can be yielded by the [:] operator and specifying the position of the first item you want in the slice and of the first you want to exclude

```
1 >>> a_list = [1, 2, 3, 4, 5]
2 >>> a_list[1:3]
3 [2, 3]
4 >>> a_list[:-2]
5 [1, 2, 3]
6 >>> a_list[2:]
7 [3, 4, 5]
```

Note that omitting one of the two indexes you get respectively the first and the last item in tbe list.

2.2.3 Lists: Adding items into the list

Four ways

- + operator concatenates two lists;
- **append()** method append an item to the end of the list;
- **extend()** method appends a list to the end of the list
- **insert()** method appends an item at given position.

2.2.4 Lists: Introspecting on the list

You can check if an element is in thelist

```
1 >>> a_list = [3,14, 1, 'c', 3.14]
2 >>> 3.14 in a_list
3 True
```

Count the number of occurrences

```
1 >>> a_list.count(3.14)
2 2
```

Look for an item position

```
1 >>> a_list.index(3.14)
2 1
```

Elements can be removed by

- position

```
1 >>> del a_list[2]
2 >>> a_list
3 [3,14, 1, 3.14]
```

- value

```
1 >>> a_list.remove(3.14)
2 >>> a_list
3 [3,14, 1]
```

In both cases the list is compacted to fill the gap.

2.2.5 Tuples

Tuples are immutable lists.

```
1 >>> a_tuple = (3,14, 1, 'c', 3.14)
2 >>> a_tuple
3 (3,14, 1, 'c', 3.14)
4 >>> type(a_tuple)
5 <class 'tuple'>
```

As a list

- parenthesis instead of square brackets;
- ordered set with direct access to the elements through the position;
- negative indexes count backward.

On the contrary

- no **append()**, **extend()**, **insert()**, **remove()** and so on.

2.2.6 Tuples (Cont'd)

Multiple assignments Tuple can be used for multiple assignments and to return multiple values.

```
1 >>> a_tuple = (1, 2)
2 >>> (a,b) = a_tuple
3 >>> a
4 1
5 >>> b
6 2
```

Benefits

- tuples are faster than lists;
- tuples are safer than lists;
- tuples can be used as keys for dictionaries.

2.2.7 Sets

Sets are unordered "bags" of unique values.

```
1 >>> a_set = {1, 2}
2 >>> a_set
3 {1, 2}
```

```
>>> len(a_set)
2
>>> b_set = set()
>>> b_set
set()  ''' empty set  '''
```

A set can be created out of a list

```
>>> a_list = [1, 'a', 3.14, "a string"]
>>> a_set = set(a_list)
>>> a_set
{'a', 1, 'a string', 3.14}
```

2.2.8 Sets: Modifying a set

Adding elements to a set

```
>>> a_set = set()
>>> a_set.add(7)
>>> a_set.add(3)
>>> a_set
{3, 7}
>>> a_set.add(7)
>>> a_set
{3, 7}
```

Sets do not admit duplicates so to add a value twice has no effects. **Union of sets**

```
>>> b_set = {3, 5, 3.14, 1, 7}
>>> a_set.update(b_set)
>>> a_set
{1, 3, 5, 7, 3.14}
```

2.2.9 Dictionaries

A dictionary is an unordered set of key-value pairs

- when you add a key to the dictionary you must also add a value for that key;
- a value for a key can be chenged at any time.

The syntax is similar to stes, but

- you list comma separate couples of key/value;
- is the empty dictionary.

Note that you cannot have more than one entry with the same key.

2.3 String

2.3.1 String

Python's string are a sequence of unicode characters String behave as lists: you can:

- get the string length with the **len** function;
- concatenate string with the + operator;
- slicing works as well.

Note that ", ' and ”' (three-in-a-row qoutes) can be used to define a string constant.

2.3.2 Formatting string

Python 3 support formatting values into strings.

that is, to insert a value into a string with a placeholder.

Looking back at the *humanize.py* example

```
1 for suffix in SUFFIX[multiple]:
2     size /= multiple
3     if size < multiple:
4       return '{0:.1f} {1}'.format(size, suffix)
5     raise ValueError('number too large')
```

- **{0}**, **{1}**, ... are placeholders that are replaced by the arguments of **format()**
- **:.lf** is a format specifier, itcan be used to add space-padding, align strings, control decimal precision and convert number to hexadecimal as in C.

2.3.3 Bytes

An immutable sequence of numbers (0-255) is a bytes object.

The byte literal syntax (b”) is used to define a bytes object

Each byte within the byte literal can be an ascii character or an encoded hexadecimal number from x00 to xff

2.4 Recursion

2.4.1 Definition: Recursive function

A function is called recursive when it is defined through itself.

Example: Factorial.

- 5! = 5*4*3*2*1
- Note that: 5! = 5*4!, 4! = 4*3! and so on.

Potentially a recursive computation

From the mathematical definition:

```
1 n! =
2 1          if n=0
3 n*(n-1)!   otherwise
```

When n=0 is the base of the recursive computation (axiom) whreas the second step is the inductive step.

2.4.2 What in python?

Still, a function is recursive when its execution implies another invocation to itself.

- directly, e.g., in the function body there is an explicit call to itself;
- indirectly, e.g., in the function calls another function that calls the function itself.

```
1 def fact(n):
2   return 1 if n<=1 else n*fact(n-1)
3
4 if __name__ == __main__:
5   for i in [5, 7, 15, 25, 30, 42, 100]:
6     print('fact({0:3d}) :- {1}'.format(i, fact(i)))
```

2.4.3 Execution: What's happen?

Still, a function is recursive when its execution implies another invocation to itself.

- directly, e.g., in the function body there is an explicit call to itself;

- indirectly, e.g., in the function calls another function that calls the function itself.

```
1 def fact(n):
2   return
3     1
4     if <=1
5     else n*fact(n-1)
```

It runs fact(4):

- a new frame with n=4 is pushed on the stack;
- n is greater than 1;
- it calculates 4*fact(3).

It runs fact(3):

- a new frame with n=3 is pushed on the stack;
- n is greater than 1;
- it calculates 3*fact(2).

It runs fact(2):

- a new frame with n=2 is pushed on the stack;
- n is greater than 1;
- it calculates 2*fact(1).

It runs fact(1):

- a new frame with n=2 is pushed on the stack;
- n is equal to 1;
- it returns 1.

2.4.4 Iteration is more efficient

The iterative implementation is more efficient...

The overhead is mainly due to the creation of the frame but this also affects the occupied memory.

As an example, the call fibo(1000)

- gives an answer if calculated by the iterative implementation;
- raises a RuntimeError Exception in the recursive solution.