L1-Python-from-Scratch

October 16, 2015

1 About Python

Python is a widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java. The language provides constructs intended to enable clear programs on both a small and large scale.

The language Python is named after the BBC show *Monty Python's Flying Circus* and has nothing to do with reptiles.

Python supports multiple programming paradigms, including *object-oriented*, *imperative* and *functional programming* or procedural styles. It features a dynamic type system and automatic memory management and has a large and comprehensive standard library.

Python was conceived in the late 1980s, and its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands as a successor to the ABC language (itself inspired by SETL) capable of exception handling and interfacing with the Amoeba operating system. Van Rossum is Python's principal author, and his continuing central role in deciding the direction of Python is reflected in the title given to him by the Python community, benevolent dictator for life.

1.1 Features

The core philosophy of the language is summarized by the document "PEP 20 (The Zen of Python)", which includes aphorisms such as:

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated
- Readability counts

Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:

- the high-level data types allow you to express complex operations in a single statement;
- statement grouping is done by **indentation** instead of beginning and ending brackets;
- no variable or argument declarations are necessary.

Python is extensible: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library). Once you are really hooked, you can link the Python interpreter into an application written in C and use it as an extension or command language for that application.

2 Installing Python

On Linux machine, Python is usually installed by default. To install Python on other systems, check out the Python Setup and Usage section in Python help documentation.

3 Using Python

The Python interpreter is usually installed as `/usr/bin/python` on those machines where it is available.

- To start a Python interpreter, type the command in your terminal: `python`.
- To terminate the Python interpreter, type an end-of-file character (`Control-D` on Unix, `Control-Z` on Windows) at the primary prompt. If that doesn't work, you can exit the interpreter by typing the following command: `quit()`.

```
fli@carbon:~$ python
Python 2.7.9 (default, Mar  1 2015, 12:57:24)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
```

3.1 Executing Python scripts

- Write down your Python script and name it as `hello.py` with `.py` extension
- Your script contents look like this

  ```
  #!/usr/bin/python
  print('Hello World')
  ```

- Go to your terminal, make your script *executable*

  ```
  chmod +x hello.py
  ```

- Run the script in your terminal

  ```
  ./hello.py
  ```

Note The line `#!/usr/bin/python` should appear at the very beginning of your file. Alternatively, you can let your environment variable to specify which Python to use as

```
#!/usr/bin/env python
```

It is possible to use encodings different than ASCII in Python source files. The best way to do it is to put one more special comment line right after the `#!` line to define the source file encoding:

```
# -*- coding: utf-8 -*-
```

Your `hello.py` will be able handle more complicated texts. By using UTF-8, characters of most languages in the world can be used simultaneously in string literals and comments. Using non-ASCII characters in identifiers is not supported. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.

3.2 Using IPython shell

IPython is a command shell for interactive computing in multiple programming languages, originally developed for the Python programming language, that offers introspection, rich media, shell syntax, tab completion, and history. IPython provides the following features:

- Interactive shells (terminal and Qt-based).
- A browser-based notebook with support for code, text, mathematical expressions, inline plots and other media.
- Support for interactive data visualization and use of GUI toolkits.
- Flexible, embeddable interpreters to load into one's own projects.
- Tools for parallel computing.

To start iPython interactive environment, type `ipython` in your terminal.

4 Getting help

- Help is available in Python sessions using `help(function)` .
- Some functions (and modules) have very long help files. When using IPython, these can be paged using the command `?function` or `function?` so that the text can be scrolled using page up and down and `q` to `quit`. `??function` or `function??` can be used to type the entire function including both the `docstring` and the code.

5 Libraries

Python has a large standard library, commonly cited as one of Python's greatest strengths, providing tools suited to many tasks. This is deliberate and has been described as a "batteries included" Python philosophy. For Internet-facing applications, a large number of standard formats and protocols are supported. Modules for creating graphical user interfaces, connecting to relational databases, pseudo random number generators, arithmetic with arbitrary precision decimals, manipulating regular expressions, and doing unit testing are also included.

Some parts of the standard library are covered by specifications, but the majority of the modules are not. They are specified by their code, internal documentation, and test suite (if supplied). However, because most of the standard library is cross-platform Python code, there are only a few modules that must be altered or completely rewritten by alternative implementations.

The standard library is not essential to run Python or embed Python within an application. Blender 2.49, for instance, omits most of the standard library.

As of August 2015, the Python Package Index, the official repository of third-party software for Python, contains more than 65,000 packages offering a wide range of functionality, including:

- graphical user interfaces, web frameworks, multimedia, databases, networking and communications
- test frameworks, automation and web scraping, documentation tools, system administration
- scientific computing, text processing, image processing

6 Text Editors/IDEs

To edit Python code, you just need a handy text editor. There are many available, check out the following pages

- Text editors in Python Wiki
- Comparison_of_text_editors in Wikipedia
- Integrated Development Environments

7 Using Python as a Calculator

```
In [1]: 3 + 2
```

```
Out[1]: 5
```

```
In [2]: 5 + 4*3
```

```
Out[2]: 17
```

7.1 Division

The return type of a division (/) operation depends on its operands. If both operands are of type int, floor division is performed and an int is returned. If either operand is a float, classic division is performed and a float is returned. The // operator is also provided for doing floor division no matter what the operands are.

```
In [3]: 8/5 #In Python 2, int / int -> int, you will get 1 instead of 1.6. But in Python 3 will get pre

Out[3]: 1.6

In [4]: 8/5.0 # int / float -> float

Out[4]: 1.6

In [5]: 8//5.0 # explicit floor division discards the fractional part

Out[5]: 1.0
```

The remainder can be calculated with the % operator

```
In [6]: 17 % 3

Out[6]: 2
```

Note To always get precise division in Python 2, use

```
from __future__ import devision
```

7.2 Powers

With Python, it is possible to use the ** operator to calculate powers. **Note** Caret (ˆ) in Python behaves differently. It invokes the exclusive OR of the object: a logical operation that outputs true only when both inputs differ (one is true, the other is false).

```
In [7]: 5**2

Out[7]: 25

In [8]: 2^3

Out[8]: 1
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
In [9]: a = 3
        b = 5
        c = a + b
        c

Out[9]: 8
```

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations.

```
In [10]: 100/3.0

Out[10]: 33.333333333333336

In [11]: _

Out[11]: 33.333333333333336
```

7.3 Strings

Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result.

```
In [12]: LastName = "Li"

In [13]: FirstName = "Feng"
```

can be used to escape quotes:

```
In [14]: print("Hello \n World!")

Hello
 World!
```

If you don't want characters prefaced by to be interpreted as special characters, you can use raw strings by adding an `r` before the first quote:

```
In [15]: print(r"Hello \n World!")

Hello \n World!
```

Strings can be concatenated (glued together) with the `+` operator, and repeated with `*`:

```
In [16]: "I " + 'L' + 'o'*5  + 've' + ' you'

Out[16]: 'I Looooove you'
```

The built-in function `len()` returns the length of a string:

```
In [17]: len("Feng Li")

Out[17]: 7
```

Two or more string literals (i.e. the ones enclosed between quotes) next to each other are automatically concatenated. This feature is particularly useful when you want to break long strings:

```
In [18]: "Feng" "Li"

Out[18]: 'FengLi'

In [19]: print("Hi, my name is Feng Li."
               " And I am from Beijing.")

Hi, my name is Feng Li. And I am from Beijing.
```

Strings can be indexed (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
In [20]: Name = "Feng Li"
         Name[0]

Out[20]: 'F'

In [21]: Name[-1]

Out[21]: 'i'

In [22]: Name[-2]
```

```
Out[22]: 'L'
```

In addition to indexing, slicing is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain a substring:

```
In [23]: Name[0:4]

Out[23]: 'Feng'

In [24]: Name[:4]

Out[24]: 'Feng'

In [25]: Name[5:]

Out[25]: 'Li'

In [26]: Name[-2:]

Out[26]: 'Li'
```

However, out of range slice indexes are handled gracefully when used for slicing:

```
In [27]: Name[5:100]

Out[27]: 'Li'
```

7.4 Lists

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
In [28]: values = [1,5,7,9,12]

In [29]: len(values)

Out[29]: 5

In [30]: values[0]

Out[30]: 1

In [31]: values[-2:]

Out[31]: [9, 12]
```

Lists also supports operations like concatenation:

```
In [32]: values + ["22","33"]

Out[32]: [1, 5, 7, 9, 12, '22', '33']
```

Lists are a mutable type, i.e. it is possible to change their content:

```
In [33]: values = [1,2,3,4,67,22]
         values

Out[33]: [1, 2, 3, 4, 67, 22]
```

```
In [34]: values[2] = 1000
         values

Out[34]: [1, 2, 1000, 4, 67, 22]
```

You can also add new items at the end of the list, by using the append() method

```
In [35]: values.append(9999)
         values

Out[35]: [1, 2, 1000, 4, 67, 22, 9999]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
In [36]: values[2:4] = [2,3,4]
         values

Out[36]: [1, 2, 2, 3, 4, 67, 22, 9999]

In [37]: values[:] = []
         values
         len(values)

Out[37]: 0
```

7.5 Building Functions

The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order. Use e.g. `help(abs)` to see the function help.

```
abs()   divmod()    input()     open()  staticmethod()
all()   enumerate()     int()   ord()   str()
any()   eval()  isinstance()    pow()   sum()
basestring()    execfile()  issubclass()    print()     super()
bin()   file()  iter()  property()  tuple()
bool()  filter()    len()   range()     type()
bytearray()     float()     list()  raw_input()     unichr()
callable()  format()    locals()    reduce()    unicode()
chr()   frozenset()     long()  reload()    vars()
classmethod()   getattr()   map()   repr()  xrange()
cmp()   globals()   max()   reversed()  zip()
compile()   hasattr()   memoryview()    round()     __import__()
complex()   hash()  min()   set()   apply()
delattr()   help()  next()  setattr()   buffer()
dict()  hex()   object()    slice()     coerce()
dir()   id()    oct()   sorted()    intern()
```

7.6 Import modules

To import a module (like `math`) that is not in Python's default module, use

```
In [38]: import math
```

Then you can use all the mathematical functions inside `math` module as:

```
In [39]: math.exp(0)
```

```
Out[39]: 1.0
```

Alternatively, you can do the following changes

```
In [40]: import math as mt
         mt.exp(1)

Out[40]: 2.718281828459045
```

If you just want to import one or two functions from a module

```
In [41]: from math import exp
         exp(3)

Out[41]: 20.085536923187668

In [42]: from math import exp as myexp

         myexp(1)

Out[42]: 2.718281828459045
```

7.7 Control Flow Tools

7.7.1 The if statements

Perhaps the most well-known statement type is the if statement. For example:

```
In [43]: x = -3

         if x < 0:
             x = 0
             print('Negative changed to zero')
         elif x == 0:
             print('Zero')
         elif x == 1:
             print('Single')
         else:
             print('More')

Negative changed to zero
```

Note

- the comma/colon sign(:) should be right after `if`, `elif` and `else` statement.
- the indentation is very important. The first non-blank line after the first line of the string determines the amount of indentation for the entire documentation string.

7.7.2 The for Statements

```
In [44]: words = ['cat', 'window', 'defenestrate']
         for w in words:
             print(w, len(w))

cat 3
window 6
defenestrate 12
```

7.8 Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or **docstring**. There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

```
In [45]: def fib(n):    # write Fibonacci series up to n
             """Print a Fibonacci series up to n."""  # the function help
             a, b = 0, 1
             while a < n:
                 print(a)
                 a, b = b, a+b
```

```
In [46]: help(fib)
```

```
Help on function fib in module __main__:

fib(n)
    Print a Fibonacci series up to n.
```

```
In [47]: fib(2000)
```

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
```

7.8.1 Function with default values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
In [48]: def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
             while True:
```

```
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print(complaint)
```

Note

`raw_input()` only works with Python 2. To make it it work with Python 3, change `raw_input()` into `input`.

The `return` statement returns with a value from a function. return without an expression argument returns None. Falling off the end of a function also returns None.

```
In [49]: ask_ok("Do you really want to go?")
```

```
Do you really want to go?yes
```

```
Out[49]: True
```

7.8.2 Anonymous functions

Small anonymous functions can be created with the `lambda` keyword. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope. The example uses a lambda expression to return a function

```
In [50]: def make_incrementor(n):
             return lambda x: x + n
         f = make_incrementor(42)
```

```
In [51]: f(0)
```

```
Out[51]: 42
```

```
In [52]: f(1)
```

```
Out[52]: 43
```

8 Coding Style

- Use 4-space indentation, and no tabs. 4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.
- Wrap lines so that they don't exceed 79 characters. This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.

- Name your classes and functions consistently; the convention is to use **CamelCase** for classes and **lower_case_with_underscores** for functions and methods. Always use `self` as the name for the first method argument (see A First Look at Classes for more on classes and methods).
- Don't use fancy encodings if your code is meant to be used in international environments. Plain ASCII works best in any case.