Advanced Programming

Federico Bruzzone

October 11, 2022

Contents

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

```python
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
  ''' Convert a file size to human-readable form. '''
  if size < 0:
    raise ValueError('number must be non-negative')
  multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
  for suffix in SUFFIX[multiple]:
    size /= multiple
    if size < multiple:
      return '{0:.1f} {1}'.format(size, suffix)
    raise ValueError('number too large')

if __name__ == '__main__':
  print(approximate_size(1000000000000, False))
  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.

```python
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

```python
if __name__ == '__main__':
  print(approximate_size(1000000000000, False))
  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:

```
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__**)
- **humanizeåpproximate_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

```
h1 = humanize.approximate_size(9128)
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:

```
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.

```
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree
```

1.2.9 Running scripts

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

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

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

```
>>> type(1)
> <class 'int'>
>>> isinstance(1, int)
> True
>>> 1+1
> 2
>>> 1+1.0
> 2.0
>>>type(2.0)
> <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 [];

```
>>> a_list = ['1', 1, 'a', 'example']
>>> type(a_list)
> <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

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

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

Count the number of occurrences

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

Look for an item position

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

Elements can be removed by

- position

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

- value

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

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

2.2.5 Tuples

Tuples are immutable lists.

```
>>> a_tuple = (3,14, 1, 'c', 3.14)
>>> a_tuple
> (3,14, 1, 'c', 3.14)
>>> type(a_tuple)
> <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.

```
>>> a_tuple = (1, 2)
>>> (a,b) = a_tuple
>>> a
> 1
>>> b
> 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.

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

```
for suffix in SUFFIX[multiple]:
    size /= multiple
    if size < multiple:
      return '{0:.1f} {1}'.format(size, suffix)
    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:

```
n! =
1          if n=0
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.

```
def fact(n):
  return 1 if n<=1 else n*fact(n-1)

if __name__ == __main__:
  for i in [5, 7, 15, 25, 30, 42, 100]:
    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.

```
def fact(n):
    return
        1
        if <=1
        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.

3 Comprehensions

3.1 Playing around with...

3.1.1 Implementing the LS command

```
import os, sys, time, humanize
from start imoprt *

modes = {
  'r': (S_IXUSR,S_IXGRP,S_IXOTH),
  'w': (S_IXUSR,S_IXGRP,S_IXOTH),
  'x': (S_IXUSR,S_IXGRP,S_IXOTH)
}

def format_mode(mode):
  s = 'd' if S_ISDIR(mode) else "-"
  for i in range(3):
    for j in ['r', 'w', 'x']:
      s += j if S_ISDIR(mode) & modes[i][j] else '-'
  return s

def formate_date(date):
  d = time.localtime(date)
  return "{0:4}-{1:02d}-{2:02d} {3:02d}:{4:02d}:{5:02d}".format(
          d.tm_year, d.tm_mon, d.tm_mday, d.tm_hour, d.tm_min, d.tm_sec)

def ls(dir):
  print("List of {0}:".format(dir))
  for file in os.listdir(dir):
    metadata = os.stat(file)
    print("{2} {1:6} {3} {0} ".format(
          file, approximate_size(metadata.st_size, False),
          format_mode(metadata.st_mode), format_date(metadata.st_ntime)))

if __name__ == "__main__": ls(sys.argv[1])
```

3.2 Introduction

Comprehensions are a compact way to transfor a set of data into another

- it applies to mostly all python's structure sype, e.g., lists, sets, dictionaries;
- it is in contrast to list all the elements.

Some basic comprehensions applied to lists, stes and dictionaries respectively

- a list composed of the first ten integers

```
>>> [elem for elem in range (1, 11)]
> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- a set composed of the first ten even integers

```
>>> {elem*2 for elem in range (1, 11)}
> {2, 4, 6, 8, 10, 12, 14, 16, 18, 20}
```

- a dictionary composed of the first ten couples (n, n$\hat{2}$)

```
>>> {elem:elem*2 for elem in range (1, 11)}
> {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

3.3 To filter out elemets of a dataset

Comprehensions can reduce the elements in the dataset after a constrint.

E.g., to select perfect squares out of the first 100 integers

```
>>> [elem for elem in range(1, 101) if(int(elem**.5))**2 == elem]
> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- **range(1,101)** generates a list of the first 100 integers (first extreme included, second excluded);
- the comprehension skims through the list selecting those elemetns whose square of the integral part of its square roots are equal.

E.g., to select the odd numbers out of a tuple

```
>>> {x for c in (1, 22, 31, 23, 10, 11, 11, -1, 34, 76, 778, 10101, 5, 44)
      if x%2 != 0}
> {1, 31, 23, 11, -1, 10101, 5}
```

- note that the second 11 is removed from the set;
- the set does note respect the tuple order (it is not ordered at all).

3.4 To select multiple values

Comprehensions can select multiple values out of the dataset.

E.g., to swap key and value in the dictionary

```
>>> a_dict = {'a': 1, 'b': 2, 'c': 3}
>>> {value:key for key, value in a_dict.items()}
> {1: 'a', 2: 'b', 3: 'c'}
```

Comprehensions can select values out of multiple datasets.

E.g., to merge two sets in a set of couples

```
>>> english = ['a', 'b', 'c'}
>>> greek = [$\alpha$, $\beta$, $\gamma$]
>>> [(english[i], greek[i]) for i in range(0, len(english))]
> [('a', $\alpha$), ('b', $\beta$), ('c', $\gamma$)]
```

E.g., to calculate the cartesian product

```
>>> {(x,y) for x in range(3) for y in range (5)}
> {(0,1),(1,2),(0,0),(2,2),(1,1),(1,4),(0,2),(2,0),
(1,3),(2,3),(2,1),(0,4),(2,4),(0,3),(1,0)}
```

3.5 Comprehensions @ work: prime numbers calculation

Classic approach to the prime numbers calculation

```
def is_prime(x):
  div = 2
  while div <= math.sqrt(x):
    if x%div == 1: return False
    else: div += 1
  return True

if __name == "__main__":
  primes = []
  for i in range(1, 50):
    if is_prime(i): primes.append(i)
  print(primes)
```

The alsorithm again but using comprehensions

```
def is_prime(x):
  div = [elem for elem in range(0, math.sqrt(x)) if x%elem==0]
  return len(div) == 0

if __name == "__main__":
  print([elem for elem in range(1,50) if is_prime(elem)])
```

3.6 Comprehensions @ work: quicksort

```
def quicksort(s):
  if len(s) == 0: return []
  else
    return quicksort([x for x in s[1:] if x < s[0]]) +
    [s[0]] +
    quicksort([s for x in s[1:] if x >= s[0]])
```

```

if __name__ == "__main__":
    print(quicksort([])
    print(quicksort([2, 4, 1, 3, 5, 8, 6, 7,])
    print(quicksort("pineapple")
    print(''.join(quicksort('pineapple)))
```

```
> []
> [1, 2, 3, 4, 5, 6, 7, 8]
> ['a', 'e',, 'e', 'i', 'l', 'n', 'p', 'p', 'p']
> aeeeilnppp
```

4 Functional programming in Python

4.1 Functional programming

4.1.1 Overvirew

What is functional programming?

- Functions are first class (objects).

  That is, everything you can do with "data" can be done with functions themselves (such as passing a function to another function).

- Recursion is used as a primary control structure.

  In some languages, no other "loop" construct exists.

- There is focus on **list processing**.

  Lists are often used with recursion on sub-lists as substitute for loops.

- "Pure" functional languages eschew side-effects.

  This excludes assignments to track the program state.

  This discorages the use of statements in favor of expression evaluations.

Whys

- All these characteristics make for more rapidly developed, shorter, and less bug-prone code.
- A lot easier to prove formal properties of functional languages and programs than of imperative languages and programs.

4.2 Functional programming in Python

4.2.1 map(), filter() & reduce()

Python has functional capability since its first release with new releases just a few syntactical sugar has been added

Basic elements of functional programming in python are:

- **map()**: it applies a function to a sequence.

```
>>> import math
>>> print(list(map(math.sqrt, [x**2 for x in range(1,11)])))
> [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
```

- **filter()**: it extracts from a list those elements which verify the passed function.

```
>>> def odd(x): return (x%2 != 0)
>>> print(list(filter(odd, range(1,30))))
> [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
```

- **reduce()**: it reduces a list to a single element according to the passed funcion.

```
>>> import functools
>>> def sum(x,y): return x+y
>>> print(functools.reduce(sum, range(1000)))
> 499500
```

Note, **map()** and **filter()** return an iterator rather than a list.

4.2.2 Eliminating flow control statements: if

Short-circuit conditional call instead of if

```
def cond(x):
  return (x==1 and 'one') or (x==2 and 'two') or 'other'

def cond2(x):
  return ('one' and x==1) or ('two' and x==2) or 'other'

if __name__ == "__main__":
  for i in range(3):
    print("cond({0}) :- {1}".format(i, cond(i)))
  for i in range(3):
    print("cond2({0}) :- {1}".format(i, cond2(i)))
```

```
> cond(0) :- other
> cond(1) :- one
> cond(2) :- two

> cond2(0) :- other
> cond2(1) :- True
> cond2(2) :- True
```

Doing some abstraction

```
block = lambda s: s
cond = lambda x: (x==1 and block('one')) or
                 (x==2 and block('two')) or
                 block('other')

if __name__ == "__main__":
  print("cond({0}) :- {1}".format(3, cond(3)))
```

```
> cond(3) :- other
```

4.2.3 Do abstraction: Lambda functions

The name lambda comes from lambda-calculus which uses the greek letter lambda to represent a similar comncept.

Lambda is a term used to refer to an **anonymous function**.

- that is, a block of code which can be executed as if it were a function but without a name.

Lambdas can be defined anywhere a legal expression can occur.

A lambda looks like this:

```
lambda "args": "an expr on the argss"
```

Thus the previous **reduce()** example could be rewritten as:

```
>>> import functools
>>> print(functools.reduce(lambda i,j: i+x, range(10000)))
> 499500
```

Alternatively the lambda can assigned to a variable as in:

```
>>> add = lambda i,j: i+j
>>> print(functools.reduce(add, range(10000)))
> 499500
```

4.2.4 Envolving factorial

Traditional implementation

```
def fact(n):
  return 1 if n <= 1 else n*fact(n-1)
```

Short-circuit implementation

```
def ffact(n):
  return (n <= 1 and 1) or n*ffact(n-1)
```

reduce()-based implementation

```
from functools import reduce
def f2fact(p):
  return reduce(lambda n,m: n*m, range(1, p+1))
```

4.2.5 Eliminating flow control statements: sequence

Sequential program flow is typical of imperative programming it basically relies on side-effect (variable assignments)

This is basically in contrast with the functional approach.

In a list processing style we have:

```
# let's create an execution utility function
do_it = lambda f: f()
# let f1, f2, f3 (etc) be functions that perform actions
map(do_it, [f1, f2, f3])
```

- single statements of the sequence are replaced by funcions
- the sequene is realized by mapping an activation function to all the function objects that should compose the sequence.

4.2.6 Eliminating while statements: Echo

Statement-based echo function

```
def echo_IMP():
  while True:
    x = input("FP -- ")
    if x == 'quit': break
    else: print(x)

if __name__ == "__main__": echo_IMP()
```

First step toward a functional solution

- No print
- Utility function for "identity with side-effect" (a monad)

```
def monadic_print(x)
  print(x)
  return x
```

Functional version of the echo function

```
echo_FP =
  lambda : monodic_print(input("FP -- ")=='quit' or echo_FP())

if __name__ == "__main__": echo_FP()
```

4.2.7 Whys

Why? To eliminate the side-effects

- mostly all errors depend on variables that obtain unexpected values.
- functional programs bypass the issue by not assigning values to variables at all.

E.g., To determine the pairs whose product is >25

```
def bigmuls(xs,ys):
  bigmuls = []
  for x in xs:
    for y in ys:
      if x*y > 25:
        bigmuls.append((x,y))
  return bigmuls

if __name__ == "__main__":
print(bigmuls((1,2,3,4),(10,15,3,22)))
```

```
from functools import reduce
import itertools

bigmuls = lambda xs, ys: [x_y for x_y in
                  combine(xs,ys) if x_y[0]*x_y[1] > 25]

combine = lambda xs, ys: itertools.zip_longest(
                        xs*len(ys), dupelms(ys,len(xs)))

dupelms = lambda lst, n: reduce(lambda s, t: s+t,
                                  list(map(lambda l,n=n: [l]*n, lst)))

if __name__ == "__main__":
print(bigmuls([1,2,3,4],[10,15,3,22]))
```

4.2.8 Future of map(), reduce() & filter()

The future of the python's map(), filter(), and reduce is uncertain.

Comprehensions can easily replace map() and filter()

- **map()** can be replaced by

```
>>> import math
>>> [math.sqrt(x**2) for x in range(1,11)]
> [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
```

- **filter()** can be replaced by

```
>>> def odd(x): return (x%2 != 0)
>>> [x for x in range(1,30) if odd(x)]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
```

Guido von Rossum finds the **reduce()** too cryptic and prefers to use more ad hoc functions instead

- **sum()**, **any()** and **all()**

To have moved **reduce()** in a module in Python 3 should render manifest his intent.

5 Closures and generators

5.1 Closures

5.1.1 On a real problem

English, from singular to plural

- if a word ends in S, X or Z, add ES, e.g., fax becomes faxes;
- if a word ends in a noisy H,add ES,e.g.,coachbecomescoaches;
- if it ends in a silent H, just add S, e.g., cheetah becomes cheetahs
- if a word ends in Y thath sound like I, change the T to IES, e.g., vacancy becomes vacancies;
- if the Y is combined with a vowel to sound like something else,just add S,e.g.,day becomes days;
- if all else fails, just add S and hope for the best;

We will design a Python module that automatically pluralizes English nouns

- All these characteristics make for more rapidly developed, shorter, and less bug-prone code.
- A lot easier to prove formal properties of functional languages and programs than of imperative languages and programs.

5.2 Functional programming in Python

5.2.1 map(), filter() & reduce()

Python has functional capability since its first release with new releases just a few syntactical sugar has been added

Basic elements of functional programming in python are:

- **map()**: it applies a function to a sequence.

```
>>> import math
>>> print(list(map(math.sqrt, [x**2 for x in range(1,11)])))
> [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
```

- **filter()**: it extracts from a list those elements which verify the passed function.

```
>>> def odd(x): return (x%2 != 0)
>>> print(list(filter(odd, range(1,30))))
> [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
```

- **reduce()**: it reduces a list to a single element according to the passed funcion.

```
>>> import functools
>>> def sum(x,y): return x+y
>>> print(functools.reduce(sum, range(1000)))
> 499500
```

Note, **map()** and **filter()** return an iterator rather than a list.

5.2.2 Eliminating flow control statements: if

Short-circuit conditional call instead of if

```
def cond(x):
  return (x==1 and 'one') or (x==2 and 'two') or 'other'

```

```
def cond2(x):
  return ('one' and x==1) or ('two' and x==2) or 'other'

if __name__ == "__main__":
  for i in range(3):
    print("cond({0}) :- {1}".format(i, cond(i)))
  for i in range(3):
    print("cond2({0}) :- {1}".format(i, cond2(i)))
```

```
> cond(0) :- other
> cond(1) :- one
> cond(2) :- two

> cond2(0) :- other
> cond2(1) :- True
> cond2(2) :- True
```

Doing some abstraction

```
block = lambda s: s
cond = lambda x: (x==1 and block('one')) or
                 (x==2 and block('two')) or
                 block('other')

if __name__ == "__main__":
  print("cond({0}) :- {1}".format(3, cond(3)))
```

```
> cond(3) :- other
```

5.2.3 Do abstraction: Lambda functions

The name lambda comes from lambda-calculus which uses the greek letter lambda to represent a similar comncept.

Lambda is a term used to refer to an **anonymous function**.

- that is, a block of code which can be executed as if it were a function but without a name.

Lambdas can be defined anywhere a legal expression can occur.

A lambda looks like this:

```
lambda "args": "an expr on the argss"
```

Thus the previous **reduce()** example could be rewritten as:

```
>>> import functools
>>> print(functools.reduce(lambda i,j: i+x, range(10000)))
> 499500
```

Alternatively the lambda can assigned to a variable as in:

```
>>> add = lambda i,j: i+j
>>> print(functools.reduce(add, range(10000)))
> 499500
```

5.2.4 Envolving factorial

Traditional implementation

```
def fact(n):
  return 1 if n <= 1 else n*fact(n-1)
```

Short-circuit implementation

```
def ffact(n):
  return (n <= 1 and 1) or n*ffact(n-1)
```

reduce()-based implementation

```
from functools import reduce
def f2fact(p):
  return reduce(lambda n,m: n*m, range(1, p+1))
```

5.2.5 Eliminating flow control statements: sequence

Sequential program flow is typical of imperative programming it basically relies on side-effect (variable assignments)

This is basically in contrast with the functional approach.

In a list processing style we have:

```
# let's create an execution utility function
do_it = lambda f: f()
# let f1, f2, f3 (etc) be functions that perform actions
map(do_it, [f1, f2, f3])
```

- single statements of the sequence are replaced by funcions
- the sequene is realized by mapping an activation function to all the function objects that should compose the sequence.

5.2.6 Eliminating while statements: Echo

Statement-based echo function

```
def echo_IMP():
  while True:
    x = input("FP — ")
    if x == 'quit': break
    else: print(x)

if __name__ == "__main__": echo_IMP()
```

First step toward a functional solution

- No print
- Utility function for "identity with side-effect" (a monad)

```
def monadic_print(x)
  print(x)
  return x
```

Functional version of the echo function

```
echo_FP =
  lambda : monodic_print(input("FP — ")=='quit' or echo_FP())

if __name__ == "__main__": echo_FP()
```

5.2.7 Whys

Why? To eliminate the side-effects

- mostly all errors depend on variables that obtain unexpected values.
- functional programs bypass the issue by not assigning values to variables at all.

E.g., To determine the pairs whose product is >25

```
def bigmuls(xs, ys):
  bigmuls = []
  for x in xs:
    for y in ys:
      if x*y > 25:
        bigmuls.append((x,y))
  return bigmuls

if __name__ == "__main__":
print(bigmuls((1,2,3,4),(10,15,3,22)))
```

```python
from functools import reduce
import itertools

bigmuls = lambda xs, ys: [x_y for x_y in
            combine(xs,ys) if x_y[0]*x_y[1] > 25]

combine = lambda xs, ys: itertools.zip_longest(
                xs*len(ys), dupelms(ys,len(xs)))

dupelms = lambda lst, n: reduce(lambda s, t: s+t,
                            list(map(lambda l,n=n: [l]*n, lst)))

if __name__ == "__main__":
print(bigmuls([1,2,3,4],[10,15,3,22]))
```

5.2.8 Future of map(), reduce() & filter()

The future of the python's map(), filter(), and reduce is uncertain.

Comprehensions can easily replace map() and filter()

- **map()** can be replaced by

```python
>>> import math
>>> [math.sqrt(x**2) for x in range(1,11)]
> [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
```

- **filter()** can be replaced by

```python
>>> def odd(x): return (x%2 != 0)
>>> [x for x in range(1,30) if odd(x)]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
```

Guido von Rossum finds the **reduce()** too cryptic and prefers to use more ad hoc functions instead

- **sum()**, **any()** and **all()**

To have moved **reduce()** in a module in Python 3 should render manifest his intent.

6 Dynamic typing

6.1 Dynamic typing

6.1.1 Variables, Object and References

As you know, Python is dynamically typed

- that is, there is no need to really explicit it

```
>>> a = 42

```

- Three separate concepts behind that assignment:
  - **variable creation**, python works out names in spite of the (possibile) content
  - **variable types**, no type associated to the variable name, type lives with the object
  - **variable use** the name is replaced by the object when used in an expression

```
>>> a = 42
```

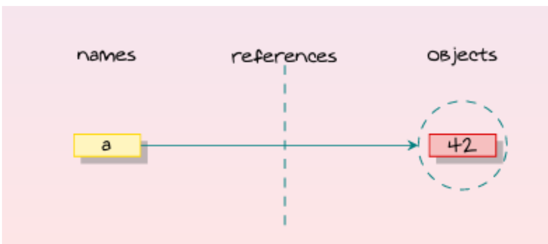
What happens inside?

- Create an object to represent the value 42

  Objects are pieces of allocated memory
- Create the variable a, if it does not exist yet;

  Variables are entries in a system table with spaces for links to objects
- Link the variable a to the new object 42

  References are automatically followed pointers from variables to objectsSS



6.1.2 Types live with objects, not varables

```
>>> a = 42      # it's an integer
>>> a = 'spam'  # now, it's string
>>> s = 3.14    # now, it's a floating point
```

Coming from typed languages programming

This lokks as the type of the name a changes

Of course, this is not true. In python

Name have no types

We simply changed the variable reference to a different object

Objects know what type they have

Each object has an header field that tags it with its type

Because objects know their type, variables don't have to

6.1.3 Object are garbage collected

What happens to the referenced object when the variable is reassinged?

```
>>> a = 42
>>> a = 'spam'   # Reclaim 42 now (unless referenced elsewhere)
>>> s = 3.14     # Reclaim 'spam' now
>>> a = [1,2,3]  # Reclaim 3.14 now
```

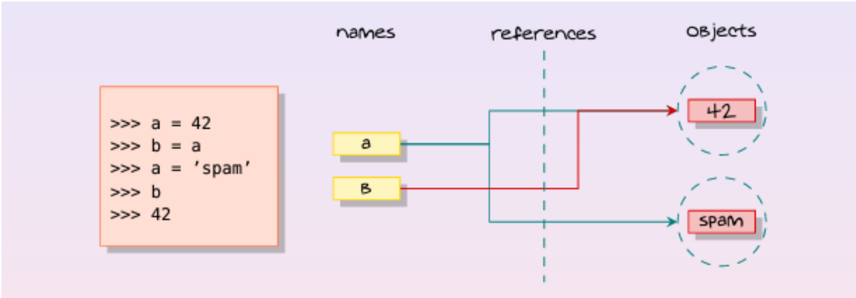
The space held by the referenced object is reclaimed (garbage collected)

If it is not referenced by any other name or object

Automatic garbage collection implies less bookkeeping code

6.1.4 Shared references

What happens when a name changes its reference and the old valueis still referred?



Is this still the same?

```
>>> a = [1,2,3]
>>> b=a
>>> b[1]='spam'
>>> b
[1, 'spam', 3]
>>> a
[1, 'spam', 3]
```

6.1.5 References & Equality

Two ways to check equality:

- == (equality) and item is (object identity)

```
>>> L=[1,2,3]
>>> M=[1,2,3]
>>> N=L
>>> L==M L is M
(True, False)
>>> L==N, L is N
(True, True)
```