Python Boot Camp

karel@e-tunity.com

Contents

About This Boot Camp

Python

- In this course you will need Python 3. There are subtle differences with Python 2, so get the latest version.
- Normally save your files with the extension *.py*, that's the standard for Python scripts or modules.
- Invoke the interpreter with (of course) *python myfile.py*.
- You can also make self-start-scripts:
  ```
  #!/usr/bin/env python
    print("Hello World!")
  ```
  Unless you're on Windows of course. Then you can't.
- Make sure that you can add Python modules as we go along. E.g., you'll need *tornado.web*. Either pre-install it, or make sure that during the course you can add modules using your favorite package manager.
- Whenever in need for information, try *pydoc*. It reports both on simple functions (*pydoc print*) and on packages (*pydoc lxml.etree*).
- Get a good editor that has a standard indentation support (4 spaces per indent recommended), color highlighting, parentheses-matching, etc..

Prerequisites

For this bootcamp you need some programming experience in a high-level language. C/C++, Java, Perl etc. suffice. In your programming skills you must be familiar with complex data types (arrays, hashes, maps), flow control (if/while/for), regular expressions, functions, classes (constructors, destructors, base classes, inheritance). For the last exercise you must have basic knowledge of the HTTP protocol (server/client) and JSON representation.

Oh, and you must know your way around your favorite operating system. This document neither explains how a command line interface works, nor why you need it.

About This Document

Types, Variables and Operators

Scalars

Variables must be initialized before usage[1]:

```
nr = 10                        # declare + initialize
print("Number is:", nr)        # use
```

"Using" a variable (as r-value) without prior assignment is an error:

```
myvar = myothervar             # error if myothervar was not
                               # an l-value previously
```

Scalars are type-free:

```
nr = 10                        # implicitly an integer
name = "Karel"                 # a string, 'Karel' is also ok
amount = 13.45                 # implicitly a float
```

Integer numbers can be specified in decimal notation as e.g. *10*, or hexadecimal as e.g. *0xa*, or octal as e.g. *0o12*. The octal notation somewhat deviates from other languages; instead of a leading zero, it needs a leading zero and a lower-case o[2].

```
print(0xa)                     # 10
print(0o12)                    # 10
```

Once Python determines the type of a variable, automatic conversion is not done (in contrast to e.g., Perl where *12 + "whatever"* gives 12, because *whatever* as number is zero):

```
nr = 10                        # number
name = "Karel"                 # string
print(nr + name)               # BANG! (TypeError: unsupported
                               # operand type(s))
```

You can type-cast using *str(), float()* and *int():*

```
nr_as_string = str(nr)
rounded_whole_amount = int(amount + 0.5)
```

[1] Time to bootstrap. The statement *print(...)* below shows some output on screen. Function invocation are explained later, but here's a first example. The *print()* statement expects one or more variables, separated by commas. Their value is shown on screen, followed by a newline.
[2] This is since Python 3. In Python 2, only a leading zero was required.

The type of a variable can always be determined using the built-in function *type()* which tests for identity (not equality!):

```
type(nr)                        # int
type(name)                      # str
type(nr_as_string)              # str
```

Boolean types have values *True* or *False*. There is also a value *None* which is like the *undef* of Perl. Python also supports multiple assignments:

```
a,b,c = 12,13,14                # assign 3 variables at once
```

This is by the way very handy in swapping operations:

```
a,b = b,a                       # swap a and b
```

Lists, Sets and Tuples and Dictionaries

Python has the following simple 'compound data types':

- Lists, sets and tuples are array-variants
- Dictionaries are what other languages call hashes or maps.

Lists

Lists in Python are what other languages call arrays. They are denoted by [and]:

```
provinces = [ "Groningen", "Friesland", "Overijssel" ]
someprov = provinces[1]         # "Friesland"
```

The number of items in a list is returned by *len(mylist)*. Indexing starts at 0 and ranges up to but not including *len(mylist)*. Slices of arrays are made using the square brackets, with *[from:to]*. The returned slice has all elements that start at index *from* up until but not including the element at index *to*. The *from* and *to* specification can be left out and defaults to 0 and *len(mylist)*. Furthermore, slicing supports a "step" specification, as in *[from:to:step]*. The step can be negative, in which case the slice 'counts backwards', which is a great way to reverse a list:

```
mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(mylist[2:5])              # [3,4,5]
print(mylist[::2])              # [1,3,5,7,9]
print(mylist[::-1])             # [10,9,8, ... 1]
```

The function *range(min,max,step))* expands to an array and is a quick way of making consecutive numbers.[3] The *min* and *step* parameters are optional and default to 0 and 1. The *step* can be negative to count backwards:

```
nrs = range(10)                 # [0,1,2, ... 9]
```

[3] Technically *range* is not a function but a class that returns a *range* object. An iterator method yields the values. So if you try *print(range(1,10))* then you don't see a list *[1,2, ...]* but you see the descriptive text *range object*. Python 2.x had also a more optimized *xrange()* that you might encounter in documents, but in Python 3 that's incorporated in *range()*.

```
nrs = range(1, 20)                # [1,2,3, ... 19]
nrs = range(0, 100, 10)           # 0..99, step 10, so stops at 90
nrs = range(10, 0, -1)            # [10,9,8, ... 1]
```

Another handy operator when constructing arrays is the * (multiplication):

```
arr = [1] * 30;                   # [ 1,1,1,1 ... (30 times) ]
print('-' * 80)                   # line of 80 hyphens
```

Yet another handy list-function is *enumerate(array)*. It iterates over an array and returns a tuple of the index and value[4]:

```
for i in enumerate(provinces):
    print(i)                      # (0,Groningen)
                                  # (1,Friesland)
                                  # (2,Overijssel)
```

The two tuple items can also be caught in two separate variables:

```
for i,p in enumerate(provinces):
    print(i,p)                    # 0 Groningen, etc.
```

To test whether an element is in an array, use the operator *in:*

```
print('Groningen' in provinces) # True
print('Limburg' in provinces)   # False
```

Sets

A set is a non-ordered, de-duplicated array. When the set is created, the ordering is lost. Sets also support operators like "in" (is something in a set?).

```
provinces = [ "Groningen", "Friesland",
              "Overijssel", "Groningen" ]
prvs = set(provinces)
"Groningen" in prvs               # True and only once in set
```

Tuples

Tuples are lists, but immutable (such as a function sees its parameter list). Tuples are denoted by (and):

```
row = (1, 2, 3, 4, 5)
row[0] = 12                       # ERROR, can't change a tuple
```

A tuple can be converted to a list using either a *for* loop to build up a new list, or by using the magic keyword *list*:

```
arr = list(row)
```

[4] Tuples are explained just below.

```
arr[0] = 12                     # No error
print(arr)                      # [ 12, 2, 3, 4, 5 ]
```

Other andy list-functions are:

- *array.append(whatever)* - appends as new element
- *array.pop()* - removes last element and returns it
- *array.pop(12)* - 12'th element
- *array.remove(element)* - removes an item, at whichever index it may be

Dictionaries

Dictionaries are what Python calls hashes or maps. They are denoted by { and }, keys are separated from values using a colon, and key/value combo's are separated by commas:

```
capitals = { "Groningen"  : "Groningen",
             "Friesland"  : "Leeuwarden",
             "Overijssel" : "Zwolle" }
```

Alternatively, the built-in function *dict()* can be used to construct a dictionary:

```
capitals = dict(Groningen='Groningen', Friesland='Leeuwarden',
                Overijssel='Zwolle')
```

This form looks suspiciously similar to keyword arguments to a function (see below).

Indexing is then done using [and]:

```
city = capitals["Friesland"]    # "Leeuwarden"
```

The keys in a dictionary can always be retrieved using the method *.keys()* which returns an array.

```
capitals.keys()                 # [ 'Groningen', 'Friesland',
                                #   'Overijssel' ]
```

New entries are added to existing dictionaries by simple indexing:

```
capitals['Gelderland'] = 'Arnhem'
```

Existing entries are deleted using the keyword *del*:

```
del capitals['Gelderland']
```

Another nice function is *zip(arr_a, arr_b)*. It creates a zip object that associates the first element of *arr_a* with the first element of *arr_b*, the second of *arr_a* with the second of *arr_b* and so on. The resulting zip object can be converted to a dictionary using *dict()*. The above dictionary *capitals* could also be constructed as follows:

```
c = [ 'Groningen', 'Leeuwarden', 'Zwolle' ]
p = [ 'Groningen', 'Friesland',  'Overijssel' ]
capitals = dict(zip(c,p))
```

Nested (recursive) arrays and dictionaries

Arrays and dictionaries can of course contain as values again dictionaries or arrays. This is in contrast to e.g. Perl where arrays or hashes can contain *references*, but not the full large data sets. Below are two entries from a hypothetical book webshop:

```
books = { 'The Hitchhikers Guide to the Galaxy' :
          { 'author'      : 'Douglas Adams',
            'formats'     :
            { 'ebook'     : { 'price'    : 12.30,
                              'shipping' :  0.00  },
              'hardcopy'  : { 'price'    : 20.00,
                              'shipping' :  2.50  },
              'paperback' : { 'price'    : 15.75,
                              'shipping' :  1.50 } } },
          'The Three Musketeers' :
          { 'author'      : 'Alexandre Dumas',
            'formats'     :
            { 'ebook'     : { 'price'    : 19.90,
                              'shipping' :  0.00  },
              'hardcopy'  : { 'price'    : 38.00,
                              'shipping' :  3.00  },
              'paperback' : { 'price'    : 24.10,
                              'shipping' :  2.00 } } } }
```

When is What Copied

A word of warning. Python won't copy arrays or dictionaries when using the assignment operator. Instead, a new variable is created which is a *reference* to the existing variable. For example:

```
a = [1, 2, 3]
b = a
b[1] = 12
print(a)                        # suddenly [1, 12, 3]
```

Now consider the books dictionary above and the below assignment[5]:

```
other = books
other['The Three Musketeers']['formats'] \
     ['hardcopy']['price'] = 99.99
print(books['The Three Musketeers']['formats']['hardcopy'] \
           ['price'])           # suddenly 99.99
```

[5] Note also the backslash. It continues a statement over the next line.

If you need a true copy (instead of a reference), then use the following trick. Compound data (lists, dictionaries) are actually objects that have a *copy()* method[6], which, as expected returns a copy:

```
a = [1, 2, 3]
b = a.copy()
b[1] = 12
print(a)                        # still [1, 2, 3]
```

For recursive data, where elements are arrays or dictionaries themselves, there's a Python module[7] *copy*. It implements an important function *deepcopy()* which makes a true copy of its argument, and of all embedded sub-structures:

```
import copy
other = copy.deepcopy(books)
other['The Three Musketeers']['formats']['hardcopy']['price'] \
    = 99.99
print(books['The Three Musketeers']['formats'] \
     ['hardcopy']['price'])     # still 38.00
```

String Goodies

Strings behave just like arrays. You can index them (*mystring[3]* is the fourth character), and slice them (*mystring[4:5]* is one character, from index 4 up to but not including index 5).

The operator % is used with strings to format a-la printf():

```
s = "My name is %s" % "Karel"
print(s)
```

Formatting multiple arguments into a string is done with a tuple:

```
s = "The first three primes are %d, %d and %d" % (2, 3, 5)
print(s)
```

Inside the string, the usual printf-like notation is used: %f for a float, %s for a string, %3.3d for a zero-padded integer over 3 positions, etc.

Three consecutive double quotes are used to start a "here-document" that can span several lines. (The term "here-document" comes from the shell script language, Perl and so on. It denotes exactly what it says, a piece of text in the middle of the script. It's usually denoted by <<*EOF* in shell-scripts.)

[6] Classes, objects and methods are explained later; here'a a quick preview: *object.method()* is how the call is coded. In this case, the object is the array or dictionary, and *copy()* is the method.
[7] See the chapters below for information on *import*, using modules and calling functions.

```
s = """
Copyright (c) e-tunity 2014ff. All rights reserved.
For information, contact info@e0tunity.com
"""
```

Other handy string functions are:

- `mystring.lower():` returns a lowercase version of *mystring*. Similarly, there is also `mystring.upper()`
- `ord(ch):` Returns the ASCII value of character *ch*. Similarly there is also `chr(val)` which returns the string (single character) for ASCII value *val*.
- For other useful string functions, run *pydoc str*.

Python also supports a "raw-string" format, where escape-sequences like \t and \n are not interpreted. Raw strings start with *r*:

```
myvar = r'Hello\nWorld'
print(myvar)          # literally Hello, backslash, n, World
```

Finally, there is a generic 'buffer' format, denoted by *b'......'* Buffers are generic data blobs, e.g., returned by HTTP clients[8]. In order to convert a buffer to a string, they need to be decoded using an encoding standard, e.g., utf-8.

Exercise

Write a program that counts how many a's, b's, c's etc. there are in the string *The quick brown fox jumped over the lazy dog*. Case-fold the characters before counting; i.e., count the *T* in *The* as a *t*, just like a lower-case *t*. The program displays the characters (a-z) and their counts. E.g., it should show that there are four o's.

Operators

Python knows the usual suspects such as +, -, /, *. Some remarks:

- + is also the string-concatenation operator
- ** is a shorthand for power raising: *2 ** 3* is *math.pow(2, 3)*
- Logical operators are *and*, *or, not*
- The comparison operators ==*, !=, <,* <= etc. also work on strings.
- % is both the modulo-operator and the string-formatting operator.
- The usual bithandling operators exist: ~ << >>
- Compound assignment operators exist: +=, -= etc..
- Python does not have the increment or decrement operators. Instead, use *myvar += 1.*
- An operator that is new for non-Python folks is probably the truncation operator //. It performs a division and returns an integer value: *15 // 2* equals *7.*

[8] See the last chapter.

Flow Control

Instead of BEGIN and END or { and }, Python uses a colon : for separation, and an indentation level to indicate what "block" belongs together. Parentheses around logical expressions are not necessary. A statement separator or terminator (such as the usual semicolon in C/C++/Java/Perl) is not necessary.

If-statement

```
if x > 5:
     statement_1
     statement_2
     statement_3
next_statement
```

When a single statement is used following the if (as opposed to a block), then a newline plus indentation is not necessary. Strictly speaking, the indentation is only necessary to form a block of two or more statements. E.g.:

```
if x == 7: print("x has the value seven")
```

If-else ladders use the keyword "elif", and the optional final "else":

```
if weather == "raining":
     print("Bring an umbrella, and maybe your jacket too")
elif weather == "windy":
     print("Bring a jacket, but not your umbrella")
else:
     print("Figure it out yourself")
```

Loops

Python has the usual flow control statements: *if*, *while*, *for*. The latter is often used with the operators *in* and the *range()* object:

```
for i in range(1, 10):
     print(i)
for i in range(0, len(provinces)):
     print("At index", i, "is province name", provinces[i])
```

The initial starting value defaults to zero; so if you need 10 times an ‘a’, you can just say:

```
for i in range(10): print("a")
```

Break and Continue

Python has the usual controls to force loop execution or to break out of it:

```
# Prints 1, 2, 4, 5, 6, then stops
i = 0
while True:
    i = i + 1
    if i == 3:
        continue
    print(i)
    if (i == 6):
        break
```

Looping over a list or dictionary

The for statement is typically used with "in" to loop over an array. To loop over a dictionary, simply have unpack to get at the keys:

```
provinces = [ "Groningen", "Friesland", "Overijssel" ]
for pr in provinces:
     print("There is a province called", pr)

capitals = { "Groningen"  : "Groningen",
             "Friesland"  : "Leeuwarden",
             "Overijssel" : "Zwolle" }
for pr in capitals.keys():
     print("The capital of", pr, "is", capitals[pr])
```

Exercise

Write a program that implements the Sieve of Eratosthenes to find primes up to 1.000.000. Use an array of 1.000.001 values of *True* or *False* to indicate whether a number is still in the number set (that way, the index can climb up to 1.000.000, to test even 1.000.000 for prime-ness).

The sieve works as follows. Represent positive numbers using a 'number line': an array, where each index is the number in question. Use booleans to indicate whether the number is still on the line. Initially, the line would look as follows (plus signs indicate a *True* value, indices are below the line, numbers 0 and 1 won't be used):

```
+++++ +++++ +++++ +++++ +++++ +++++ +++++ +++++ +++++ +++++ +++++
0     5    10    15    20    25    30    35    40    45    50
```

Now start at index 2 and proceed as follows:

- If the number under the index is *False*, then it's not a prime.
- If the number under the index is *True*, then it's a prime. You can print it. Next all multiples of the index are set to *False*.
- After this, move on to the next index.

E.g., after checking number 2 (which is a prime), the line looks as follows:

```
++++- +-+-+ -+-+- +-+-+ -+-+- +-+-+ -+-+- +-+-+ -+-+- +-+-+ -+-+-
0     5     10    15    20    25    30    35    40    45    50
```

The next index to check is 3, which is again on the line, so it's a prime. After removing the multiples of 3, the line is:

```
++++- +-+-- -+-+- --+-+ ---+- +---+ -+--- +-+-- -+-+- --+-+ ---+-
0     5     10    15    20    25    30    35    40    45    50
```

The sieve can be expressed in 6 lines of code. Maybe even less.

Functions

Defining a Function

A function is defined using the keyword *def*, then the function name, followed by optional parameters in parentheses:

```
def myfirstfunction():
     print("Hello World")
def greetperson(name):
     print("Welcome", name)
```

The shortest form of a do-nothing function just has one statement *pass* which is the "no-operation" statement in Python:

```
def silly():
    pass
```

Global vs. Local Variables

Unless specified otherwise, variables will be local in a function. In the following snippet, *nr* will be a local variable in *myfun()*:

```
nr = 50
def myfun():
    nr = 51

print(nr)       # 50
myfun()
print(nr)       # still 50
```

In order to address a global variable, use the keyword *global*:

```
nr = 50
def myfun():
    global nr
    nr = 51

print(nr)       # 50
myfun()
print(nr)       # 51
```

Exercise

Given the above-shown list of books in a bookshop:

```
books = { 'The Hitchhikers Guide to the Galaxy' :
```

```
{ 'author' : 'Douglas Adams',
  'formats' :
  { 'ebook' : { 'price' : 12.30,
                'shipping' : 0.00 },
    'hardcopy' : { 'price' : 20.00,
                   'shipping' : 2.50 },
    'paperback' : { 'price' : 15.75,
                    'shipping' : 1.50 } } },
'The Three Musketeers' :
{ 'author' : 'Alexandre Dumas',
  'formats' :
  { 'ebook' : { 'price' : 19.90,
                'shipping' : 0.00 },
    'hardcopy' : { 'price' : 38.00,
                   'shipping' : 3.00 },
    'paperback' : { 'price' : 24.10,
                    'shipping' : 2.00 } } } }
```

Write a function *cheapest_paperback()* that examines the list and prints the title and price of the cheapest paperback book in the shop. The price must be the total consumer price; i.e., cost plus shipping. With the above shop we expect to see The Hichhikers Guide to the Galaxy for a price of 17.25. Obviously your code must be able to handle larger arrays than just the two-element one above. But you can depend on the dictionary tags being author, formats and so on.

Default Parameters

Python understands "default parameters" that may or may not be given:

```
def countdown(start = 10):
     for nr in range(start, 0, -1):
          print(nr)
     print("Boom!")
countdown()                      # 10, 9, 8, ... boom
countdown(5)                     # short fuse
```

Variable Argument List

A variable argument list is stated as **varlist* in the function's parameter list (note the asterisk). Inside the function, varlist is treated as a tuple:

```
def sayhello(*names):
     print("There are", len(names), "persons here.")
     for n in names:
          print("Hello,", n)
sayhello("Fred", "George", "Hermione")
```

Keyword Argument List

Functions can also get "label=value" arguments. Often these are used as options. The caller writes this as a "label=value" form. In the function definition, keyword argument lists are denoted by ***kw* which stands for "keywords" (note the double asterisk). Inside the function body, keyword arguments are analyzed as if they were a dictionary. The below example uses both a variable argument list and a keyword argument list.

```
def operation(*nrs, **kw):
    if kw['mode'] == 'sum':
        sum = 0
        for nr in nrs:
            sum = sum + nr
        return sum
    if kw['mode'] == 'mult':
        mul = 1
        for nr in nrs:
            mul = mul * nr
        return mul
    print("error, specify either mode=sum or mode=mult")
    return None


print("Sum mode:",
      operation(1, 2, 3, 4, mode='sum'))              # 10
print("Multiplication mode:",
      operation(1, 2, 3, 4, mode='mult')              # 24
```

The keyword arguments are in fact a dictionary, but without the surrounding *dict()* statement.

Passing parameters to sub-functions

When passing variable argument lists to subfunctions, then the asterisk must be repeated in the call. Also, when passing keyword arguments, the double asterisk must be repeated:

```
def myfun(arg1, arg2, *args, **kw):
    otherfun(*args, **kw)
```

Returning from a Function

The return statement is of course a typical flow control statement in a function: at that point, the function stops (and some value may be returned to the caller). The syntax is, as expected: *return* (which doesn't return anything) or *return val* (which returns a value).

In Python more than one value can be returned. Here is an example of a function that returns a boolean and some other value:

```
def myfun():
    ...
    return True, 12

success,number = myfun()
if not success:
    ....
```

Return vs. Yield

As an example, consider this function that returns an array of odd numbers with a default array size of 100.

```
# Return array of odd numbers (1, 3, 5, etc)
def oddnrs_array(sz = 100):
    ret = []
    counter = 1
    while len(ret) < sz:
        ret.append(counter)
        counter += 2
    return ret

# Prints odd numbers, stop when 50 is reached.
for nr in oddnrs_array():
     print(nr)
     if nr > 50:
          break
```

The obvious disadvantage is that the entire array is built and passed to the caller (memory-heavy). For that, Python has a "yield" statement, which makes the function where it is used a *producer* function. The *consumer* calls the function as if it would return an array; while it only returns one element at a time. Technically, the function is a placeholder for an *iterator* that is used in a for-statement.[9]

```
# Iterator that yields odd numbers 1, 3, 5, etc.
def oddnrs_iter():
    counter = 1
    while True:
        yield counter
        counter += 2

# Print odd numbers, stop when 50 is reached.
```

[9] Even more technically, the function doesn't exist. The code of the function is a *generator* that pastes the function's code at the calling points. Or something like that. Ask Guido van Rossum if you want to know the gory details; he implemented this in Python 2.4.

```
for nr in oddnrs_iter():
    print(nr)
    if nr > 50:
        break
```

A summary of differences between a function that returns an array or one that yields elements:

- A yielding function returns elements at a time, it is memory-light.
- A function that returns an array will need to have a max array size limit (you can't build an infinitely large array). A yielding function doesn't need a limit; the caller decides when to stop consuming.
- A function that returns an array, makes an assignment possible to an other array: `my_odd_nrs = oddnrs_array()`
- A function that yields elements doesn't allow such assignments; the function produces single values. When necessary, the caller has to construct the array:
  ```
  # Get the first 10 odd numbers
  my_odd_nrs = []
  for i in range(10):
        my_odd_nrs.append(oddnrs_iter())
  ```

The two approaches can be very well mixed. If you need to write a function that returns an array, simply write an underlying producer, and use it inside the function that builds and returns the array. That way, a coder can choose for themselves what to use, the iterator or the array version.

```
# Odd numbers as iterator
def oddnrs_iter():
    counter = 1
    while True:
        yield counter
        counter += 2

# Odd numbers as array, uses the iterator to build
# its return value
def oddnrs_array(sz = 100):
    ret = []
    for nr in oddnrs_iter():
        ret.append(nr)
        if len(ret) >= sz:
            return ret
```

Functional Tools for Lists

Python has three functions for lists that require your code to do their work:

- *filter(function, list):* For each element *el* of the list, the *function(el)* is invoked. It has to report whether the element is eligible to be filtered. The return value of *filter()* is a new array of - of course - filtered items.
- *map(function, list):* For each element the *function(el)* is invoked. The return value of *map()* is a list of all thus obtained values.
- The mapping function can be called with more than one list. In that case, the function is called with two parameters, *el1* and *el2* from the resp. the first and second list.
- *functools.reduce(function, list):* Here the function is called the first time with the first two elements of the list as arguments. The function must return a value. The second time, the function is called with the arguments: result of the first call, and third element. Again it must return a value. This is repeated over the entire array. The return value of *reduce()* is thus computed.[10]

An example is shown below.

```
import functools                      # for functools.reduce

nrs = range(1,21)                     # 1,2,3,4,...

def isodd(x): return x % 2 == 1
for i in filter(isodd, nrs):
    print('filtered:', i)             # 1,3,5,7,...

def addhundred(x): return x + 100
for i in map(addhundred, nrs):
    print('mapped:', i)               # 101,102,...

def sumelem(x, y): return x + y
sum = functools.reduce(sumelem, nrs)
print('reduced:', sum)                # 210
```

Lambda Functions

Anonymous Lambda functions are denoted by the keyword *lambda*. For example, recall the *map()* example above:

```
def addhundred(x): return x + 100
for i in map(addhundred, nrs):
    print('mapped:', i)               # 101,102,...
```

The disadvantage here is that a name *addhundred()* must be defined, which leads to namespace pollution etc.. The example could be rewritten as follows:

[10] As of Python 3, *reduce()* is in module *functools*. You will need an import to make it work. More on importing modules later.

```
for i in map(lambda nr: nr + 100, nrs):
    print('mapped:', i)
```

The syntax of a Lambda-function is *lambda symbol: expression*. The expression is what the anonymous function returns given the input *symbol*.

Lambda functions can be assigned to a variable. The function call is then via the variable:

```
threemore = lambda x: x + 3
print(threemore(7))                 # 10
```

Or a Lambda function can have multiple arguments:

```
addtwonumbers = lambda x,y: x + y
print(addtwonumbers(3, 4))          # 7
```

The fact that Lambda functions can be assigned to a variable means that they can also be the return value of a normal function. In that case, the function is a *generator*:

```
def make_addnumber(x):
    return lambda nr: nr + x
fourmore = make_addnumber(4)        # new function
print(fourmore(14))                 # 18
```

Generators and Decorators

Imagine that you have a function to compute the factorial number of *n*, called *fact(n)*. The function is defined as follows:

- *fact(1)* is 1
- *fact(2)* is 2
- *fact(3)* is 3 times *fact(2)*
- *fact(4)* is 4 times *fact(3)*
- etc..

Someone wrote that function for you, and it reads:

```
def fact(n):
    if n < 3:
        return n
    return n * fact(n - 1)
```

Now you want to use this function in your own code, but you have two special requirements:

- The argument must be an *int*. There's no sense in computing the factorial of a string, float, etc..
- The argument must be a positive number.

You could of course write your own function that satisfies these requirements (or raises an exception when the argument is wrong) and you could call it *myfact()*. But there's another way. Using generators and decorators.

Wrapping a function using a generator

Here is a generator function that returns a function wrapper which checks that the type of an argument is an *int:*

```
def needs_int(fn):
    def wrapper(arg):
        if type(arg) != int:
            raise Exception('function needs int argument')
        return fn(arg)
    return wrapper
```

Similarly here's a generator that returns a wrapper which checks that the argument is a positive number:

```
def needs_positive_nr(fn):
    def wrapper(arg):
        if arg < 1:
            raise Exception('function needs positive ' +
                               'number  as argument')
        return fn(arg)
    return wrapper
```

The simplistic usage of the above two wrappers and the predefined function *fact()* could be:

```
wrapped_1 = needs_int(fact)
wrapped_2 = needs_positive_nr(wrapped_1)
print(wrapped_2(12.5))          # BOOM not an int
print(wrapped_2(-3))            # BOOM not a positive number
print(wrapped_2(12))            # Yup 479001600
```

However, instead of introducing new names, the symbol *fact* can also be redefined:

```
fact = needs_int(fact)
fact = needs_positive_nr(fact)
```

after which you can use it with all the glorious new features:

```
print(fact(12.5))               # BOOM not an int
print(fact(-3))                 # BOOM not a positive number
print(fact(12))                 # Yup 479001600
```

Decorators

The approach of re-defining a function symbol through a wrapper is so common-place that Python offers the syntax *@wrappergenerator* as an alternative to *myfun=wrappergenerator(myfun)*. The statement *@wrappergenerator* occurs on a separate line just prior to the function definition and can be repeated for as many wrapper generators as necessary:

```
@needs_int
@needs_positive_nr
def fact(n):
    if n < 3:
        return n
    return n * fact(n - 1)
print(fact(12.5))                # BOOM not an int
print(fact(-3))                  # BOOM not a positive number
print(fact(12))                  # Yup 479001600
```

Exercise

Imagine you have a function *fib(n)* that returns the n'th Fibonacci number. This number is defined as follows:

- *fib(0)* is 0
- *fib(1)* is 1
- *fib(2)* is *fib(1)* + *fib(0)*
- *fib(3)* is *fib(2)* + *fib(1)*
- etc..

The function code is:

```
def fib(n):
    if n in (0, 1):
        return n
    return fib(n - 1) + fib(n - 2)
```

Here is the exercise.

1. Use the function to compute the 40th Fibonacci number.
2. How long does this take? Why? (Hint: add *print()* statements to see what's going on.)
3. Write a wrapper generator called *cached()* that caches previously obtained results in a local dictionary.[11] Use the wrapper on the above function *fib()*.
4. How long does it take now to compute the 40th Fibonacci number?

[11]This technique is officially called *memoizing*. It's especially useful on functions that need previously obtained results for new results: typically recursive or iterative functions. It's also very useful for functions that need to run long queries.

Simple Input and Output

We have already met the function *print()*. This is acutally a built-in language construct. The function takes an arbitrary number of arguments, prints them on stdout separated by spaces, and prints a newline at the end.[12]

You can suppress newline printing by specifying a different terminator:

```
print('Hello', 'World')          # will be followed by a newline
print('Hello', 'World', end='')  # won't be followed by a newline
```

Similarly, the default space separator can be redefined:

```
print('Hello', 'World', sep='-') # Hello-World
```

Standard Streams and write()

To handle the three standard streams directly (*stdout*, *stderr* and *stdin*), use the module *sys*:

```
import sys
```

After this, *sys.stdout* and *sys.stderr* are available for writing and *sys.stdin* for reading.

To send output to e.g. *sys.stdout*, you can use the *write()* method. This accepts just one argument, a string, so you're likely to use %-formatting:

```
sys.stdout.write('Hello World\n' +
                 'Another line\n' +
                 'A number with 3 digits precision: %.3f\n' \
                     % (10.0 / 3))
```

Note that the *write()* method takes only one argument, so you must concatenate separate strings using the plus sign onto one parameter. The usual escape sequences are available: \n, \t etc.

Alternatively, function *print()* supports the *file* keyword that also can be used:

```
print('Hello World\n',
      'Another line\n',
      'A number with 3 digits precision: %.3f\n' % (10.0 / 3),
      file=sys.stdout)               # or sys.stderr, or an open
file
```

Exercises

1. For the C geeks. Write a function *printf()* that has at least one argument, a format string. Optionally next arguments can be present, which get expanded according

[12] Python 2.x uses the form *print "arguments",* without parentheses, because *print* is a language construct, like *if* etc. Python 3.x requires parentheses in order to make the statement look like a function invocation; even though *print* is still built-in.

to *%s*, *%d* etc. in the format string. E.g.:

```
printf('Hello ')
printf('World!\n')                  # Hello World! <newline>

printf('%d plus %d equals %d\n',
       12, 13, 25)                  # 12 plus 13 equals 25
<newline>
```

2. Write a program that displays the Fibonacci series until a number larger than 100 is reached. The series runs as follows: 1, 2, 3, 5, 8, etc. Each element is the sum of the previous two. The series is initialized with 1 and 2.
    a. Don't use the recursive variant described above. Instead, write an iterative version.
    b. First write the program in such a way that there is a function fibo() which returns the next Fibonacci number (so it returns one integer). The state can be kept in two global variables.
    c. Now rewrite the program in such a way that there is a function fibo() which is a producer: it yields the next Fibonacci number. The state must kept in local variables; there may be no pollution outside of the function.

3. Write a pi estimator in the following manner.
    - The program runs a loop a large number of times, say 100.000, 500.000, 1.000.000 or more. The loop count should be a variable at the top of the program.
    - Per loop, the program chooses a random point in the square between coordinates (-1,-1) and (1,1). Then the program determines whether this point is inside the unit circle (which is a circle with its centre at (0,0) and having radius 1). Using this information, pi is estimated as follows:
        - Given the *x* and *y* coordinate, the distance of the point *(x,y)* to the origin (0,0) is computed.
          If the distance is less than or equal to 1.0, then a counter, the "number of hits within the unit circle" is increased.
        - After all loops have been done, the ratio of the hits within the unit circle and the total of hits in the square is computed. This ratio is an estimate of 1/4 pi.
        - Finally, the estimate of pi is displayed.

    - Hints: you'll need the module *random*, with one of the random functions. E.g., *random.random()* returns a random value between 0 and 1.
      You'll also need the module *math*. There is a function *math.sqrt()* and *math.hypot()*.

   How many loop iterations do you need to estimate pi up to a precision of 3 digits?

Files

The above *sys.stdout* and *sys.stderr* are of course files, just as an open network socket [13]. You can open your own files for reading, writing and appending using *open()*. Here are some examples of files being read:

[13] See the appropriately named module *socket*.

```
f = open('/etc/bashrc', 'r')                # open file
print(f.read())                             # read & print file
f.close()                                   # close

f = open('/etc/bashrc', 'r')                # open file
line = f.readline()                         # read & print lines
while line:                                 # until no more info
    print('Line:', line, end='')            # note: line already
    line = f.readline()                     # has \n in it
f.close()                                   # close
```

Writing to a different file can done via opening it, and supplying a *file*=... keyword to *print():*

```
f = open('myfile.txt', 'w')                 # w=write, a=append
print('Hello World', file=f)
```

Alternatively an open file will support the *write()* method:

```
f = open('myfile.txt', 'w')
w.write('Hello World\n')
```

Binary Mode

Files that are opened in moded *r, w* or *a* are suitable to read or write strings (or string representations). If you need to write or read a buffer of bytes, open the file in *rb* or *wb*.

Regular Expressions

Just like all modern languages, Python of course has (Perl-like) regular expressions. These are contained in the module *re*. After *import re* you have the following most important functions:

- *re.search(pattern, string):* True if the pattern can be found in the string
- *re.split(pattern, string):* Splits the string by the pattern.
- *re.findall(pattern, string):* Finds all occurrences of the pattern in the string, returns an array of the matched substrings.
- *re.finditer(pattern, string):* Similar to *findall*, but returns "match objects" with information on the match start, match end and matched substring.
- *re.sub(pattern, replacement, string):* Calls *function* with a match-object for every occurrence of the pattern. The replacement may contain *\1*, *\2* etc. for matches of groups in the pattern.
- *re.sub(pattern, function, string):* Similar to the above, but the *function* is called to modify the string.
- *re.compile(pattern):* Compiles the pattern and returns a "regex program". The regex program can be re-used; this is handy in loops where you don't want Python to compile the same pattern over and over again.

Simple examples

The most simple search:

```
import re

haystack = """Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor incididunt ut labore
et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate
velit esse cillum dolore eu fugiat nulla pariatur. Excepteur
sint occaecat cupidatat non proident, sunt in culpa qui
officia deserunt mollit anim id est laborum."""

if re.search(r'ipsum', haystack):
    print('ipsum occurs in the haystack')
```

Note how the pattern in *r'ipsum'* is stated as a "raw" string. You should always do this, to avoid multiple string interpretations. E.g., *r'\n'* specifies a newline; otherwise, you'd have to type *'\\n'*. This is due to the fact that the string must be passed to the underlying module which triggers a second interpretation.

Following the "haystack" parameter, search- or match options are given. The most important ones are:

- *re.IGNORECASE* or *r.I:* case-insensitive
- *re.MULTILINE* or *re.M:* when working with multi-line strings
- Multiple flags are OR-ed: *re.I* | *re.M*

For example:

```
if re.search(r'SUNT', haystack, re.I):
    print('SUNT occurs in the haystack without regard to casing')
```

To find all occurrences of a substring, use *re.findall()*. This function returns an array of all matches. The following snippet searches for substrings that begin with a space, then two any characters, then a lower or uppercase 't':

```
for r in re.findall(r'( ..t)+', haystack, re.M | re.I):
    print(r, 'occurs in the haystack')
```

Pre-compiling Regular Expressions

Pre-compiling regular expressions into a "regex program" is handy when the expression is used multiple times, e.g. in a match. The following snippet splits the haystack into lines, and then tries to find *r'i...m'* (this will be found twice, in the first land fifth line). When speed is relevant, this search-expression can be pre-compiled:

```
prog = re.compile(r'i...m')
for line in re.split(r'\n', haystack):
    if prog.search(line):
        print('i...m occurs in:', line)
```

Using a pre-compiled expression (a regex-program) is very similar to the standard usage:

- Instead of *re.search(pattern, string)* you use *prog.search(string)*
- Instead of *re.findall(pattern, string)* you use *prog.findall(string)*
- Instead of *re.split(pattern, string)* you use *prog.split(string)*
- etc..

The Match Object

Some functions, such as *re.finditer(),* return match objects. A match object *m* has the following useful properties:

- *m.group(0)* is the first matched substring of the pattern where *(...)* was used - so this is a-la $1 in Perl. For example: given string *Karel Kubat* and pattern *r'Karel (.*)'*, the first group will contain *Kubat*. Another example: given pattern *r'(\w) (\w)',* the first group *m.group(0)* will contain *Karel* and the second which is *m.group(1)* will contain *Kubat*.
- *m.start()* is the start index where the match was found
- *m.end()* is the end index.

Example:

```
for r in re.finditer(r'( ..t)+', haystack, re.M | re.I):
    print(r.group(0), 'occurs in the haystack at',
        r.start(), 'until', r.end())
```

Matching and Replacing

Using *re.sub()* you can replace substrings. The parameters are:

- The pattern to match, which should be expressed as *r'.....'*. Groups in the pattern must be enclosed by *( )*, they are assigned to *\1, \2* etc.
- The replacement for matches. Groups in the pattern can be referred to using their ordering *\1, \2* etc. The replacement pattern is also given as a raw string in the format *r'....'* in order to protect *\1* and so on.
- The string to search in
- Flags, e.g. *re.I* or *re.M*

For example:

```
newstring = re.sub(r'(.*) (s..) (.*)', r'\1 XYZZY \3',
                   haystack, re.M | re.I)
print(newstring)
# Output:
# Lorem ipsum dolor XYZZY amet, consectetur adipisicing
# elit, XYZZY do eiusmod tempor incididunt ut labore et dolore
# magna ... etc ...
```

Providing a Custom Processing Function

Method *re.sub()* allows you to provide your own processing when a match occurs. The parameters to *re.sub()* are:

- The pattern
- The custom processing function
- The string to search in
- Again, optional flags such as *re.I*

The processing function is called with a match object, where *m.group(0)* is the complete matched string. E.g, in the case of matching with flag *re.M* (mutliline), this will be the line where the match occurred. Then, *m.group(1)* is the first matched *(...)* of the pattern, and so on:

```
nchanged = 1
def changer(m):
    global nchanged
    print(m.groups())
    newstr = m.group(1) + ' XYZZY-' + str(nchanged) +
             ' ' + m.group(3)
    nchanged += 1
    return newstr

newstring = re.sub(r'(.*) (s..) (.*)', changer,
                   haystack, re.M | re.I)
print(newstring)

def changer(m):
    return m.group(1) + ' XYZZY ' + m.group(3)
newstring = re.sub(r'(.*) (s..) (.*)', changer,
                   haystack, re.M | re.I)
print(newstring)
# Output:
# Lorem ipsum dolor XYZZY-1 amet, consectetur adipisicing
# elit, XYZZY-2 do eiusmod tempor incididunt ut labore et dolore
# magna ... etc ...
```

Exceptions

Trying and catching

Keywords *try* and *except* surround blocks of code where exceptions may occur:

```
try:
     result = x / y
except Exception:
     # Probably a type error, or division by zero
     print("Division didn't work out")
```

Exceptions have their own type and string representation. In the above example, the exception may actually be a *ZeroDivisionError*, which is a specific type of an *Exception*. To print what the error is, the exception is caught as a variable, and the *str()* function is used to report what it is:

```
try:
     result = x / y
except Exception as e:
     print("Division didn't work out, error is:", str(e))
```

It is possible to differentiate by exception type. The type *Exception* is a generic; and given the operation(s), there may be sub-types. For example:

```
try:
     result = x / y
except ZeroDivisionError as e:
     print("You cannot divide by zero,",
           " the universe would explode")
     print("Error description:", str(e))
except TypeError as e:
     print("You cannot mix number and non-number ",
            "types in divisions")
     print("Error description:", str(e))
```

Final Cleanups

If you have code that needs to run in both the ‘happy flow’ (no exceptions) *and* in the exceptional flow, then you could of course duplicate the code in the *try* block and in the *except* block. But as always, copy/paste is evil. Therefore there’s the *finally* block, which always runs and is ideal for e.g. cleanups. In the below code *all done* is always printed:

```
try:
    a = b / c
    print('got a:', a)
except Exception as e:
    print('exception:', str(e))
finally:
    print('all done')
```

Raising

Your own code can raise ("throw") exceptions using the keyword *raise*. E.g.:

```
def pay_salary(name, amount):
    if amount < 0:
        raise Exception("Trying to pay a negative amount " +
                            str(amount) + " to employee " + name)

pay_salary("Fred", 12)          # ok
pay_salary("George", -3)        # boom
```

Passing and Re-raising

Sometimes you may want to ignore an exception. In that case you need the *except* part, with a *pass* statement. Alternatively you might want to perform special actions and throw the exception anyway. For example, assume that you want to fetch an external resource (from some website or whatever) and retry three times:

```
for try_nr in range(1, 4):
    try:
        result = fetch_resource()
        process_result(result)
        break
    except Exception as e:
        if try_nr < 3:
            print('Fetching the resource failed')
            print('Try %d, reason %s' % (try_nr, str(e)))
            pass                # ignore this exception
        else:
            print('Fetching the resource failed 3 times.)
            print('Giving up, reason: %s' % str(e))
            raise e             # rethrow to caller
```

Organizing Your Code

Writing Python Modules

Normally you will put your functions in a separate file, e.g. *mylib.py*. Such a library may be imported in other programs using *import mylib*. Documenting and organizing code can be done as follows:

1. Document the functions using a here-document string just below the *def* line:
   ```
   def hello():
           """ Prints 'Hello World' to standard out. """
           print("Hello World")
   def greet(name):
            """ Prints a greeting in the format
                Hello, NAME, nice to see you.
                The name is the function argument. """
            print("Hello, %s, nice to see you" % name)
   ```

2. At the top of the file, put general module information, also as a here-document string:
   ```
   """ This module implements two greeting functions. """
   ```

3. That way, *pydoc /path/to/mylib.py* will show the contained functions in *mylib.py* and the information about them. When *mylib.py* is in the default search path, then *pydoc mylib* will work too.

4. Add unit-testing code to your module as follows. When the module is invoked directly, e.g. as *python mylib.py*, then Python makes sure that the built in variable __*name*__ has the value __*main*__. You can test for that:

   ```
   if __name__ == "__main__":
           # Invoked as 'python mylib.py'. Run some unit tests.
           greet('Waldo)
           hello()
   ```

Importing Modules

Modules are imported using the *import* statement. There are alternatives:

```
# 1. Import all from mylib into the namespace mylib, hence
# functions will be mylib.hello() and mylib.greet()
import mylib
mylib.hello()
mylib.greet('Loesje')


# 2. Import into our own namespace
```

```
from mylib import hello, greet
hello()
greet('Opa')

# 3. Import under a different name
from mylib import hello as h, greet as g
h()                                    # mylib's hello() function
g('Oma')                               # mylib's greet() function
```

Classes

Functions in Python don't have "call by reference", just "call by name". They can of course return values, but what if they need to manipulate data in a more complex manner?

For that, Python uses classes:

- A class is a package, held in its own file *class.py*
- It has data members
- It has function members (methods). All methods must have a first argument *self* in the definition.
- Data and methods are by default "public" (available to the caller), unless their names start with a single underscore. Then, Python automatically suppresses exporting the names.
- Classes may implement special methods, such as:
    - *__init__()* : the constructor
    - *__repr__()* : representation wrapper, called when an object is cast to a string as in *str(obj)*
    - *__eq__()* : equality testing, called during comparison as in e.g. *if obj1 == obj2*. Similarly there is *__le__(), __lt__(),* etc..
    - *__del__(), __enter__()* and *__exit__():* Lifecycle-related, see destructors below.

Writing a Class Definition

Below is a trivial class Person (held in *person.py*) that allows storage of a first and last name. There is a method *full()* that returns the first name, followed by space, followed by the last name. Special methods are the constructor, string serializer, and equality tester. Note also the self-documenting features and the simple unit tests at the bottom.

```
""" Class Person

Allows storage of a first and last name. For example:

 p = Person('John', 'Doe')      # constructor with first/last name
 p.first = 'James'              # first name
 p.last = 'Cameron'             # last name
 fullname = p.full()            # both as a string
 print(str(p))                  # string casting
 if p == otherperson: ...       # equality testing by first+last
"""

class Person:
    def __init__(self, first, last):
        """ Constructor. Expects two arguments,
            first and last name. """
        self.first = first
        self.last  = last

    def full(self):
        """ Returns the first and last name separated
            by a space."""
        return self.first + ' ' + self.last

    def __repr__(self):
        """ str() representation of a Person """
        return 'firstname=' + self.first + \
               ', lastname=' + self.last

    def __eq__(self, other):
        """ Equality tester """
        return self.first == other.first and \
               self.last == other.last

if __name__ == '__main__':
    p = Person('Ronald', 'Reagan')
    q = Person('Donald', 'Duck')
    r = Person('Ronald', 'Reagan')

    print("Donald as string:", str(q))

    if p == q:
        print(p.full(), 'equals', q.full())
    else:
        print(p.full(), 'is not equal to', q.full())

    if p == r:
        print(p.full(), 'equals', r.full())
    else:
        print(p.full(), 'is not equal to', r.full())
```

Using a Class Module

Using a class module is similar to any module:

- Importing: *from modulefile import classname*
- Defining a variable: *obj = classname(optional-parameters)*
- Accessing object data: *obj.dat = 12*
- Calling methods: *obj.method(optional-parameters)*

For example, with the above Person class:

```
from person import Person
p = Person('Klaas', 'Vaak')
print(p.full())
```

Wiring a Destructor

The basic destructor in Python is called *__del__()*. It goes off as soon as an object goes out of scope.[14] Here's an example:

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last  = last
        print('Instantiated a Person object for %s %s' % \
              (first, last))
    def __del__(self):
        print('Object for person %s %s ceased to exist' % \
              (self.first, self.last))
def tester():
    a = Person('Goede, de', 'Sint')
def main():
    b = Person('Klaas', 'Vaak')
    tester()
    c = Person('Mark', 'Rutte')
main()
# Output:
# Instantiated a Person object for Klaas Vaak
# Instantiated a Person object for Goede, de Sint
# Object for person Goede, de Sint ceased to exist
# Instantiated a Person object for Mark Rutte
# Object for person Klaas Vaak ceased to exist
# Object for person Mark Rutte ceased to exist
```

[14] And not when the runtimer decides that it's about time to send along the garbage collector, as some primitive languages do.

Wiring a destructor into a class where a global variable must be accessed, is slightly more work. You will need two additional functions:

- *__enter__(self):* code that gets executed when an instance from the package is instantiated;
- *__exit__(self, type, value, traceback):* code that gets executed when an instance from the package is removed. See the Python documentation for the purpose of *type, value* and *traceback* (these are somewhat advanced topics).

Therefore, Python basically offers two approaches for the creation of destructors:

- *__del__(self)*, and
- *__exit__(self, ...)* in combination with *__enter__(self)*

The reason for having these two approaches is the following. Method *__del__(self)* can only access variables bound to an instance of the class; i.e., instance-variables. In contrast, *__exit__(self, ...)* can also access class-bound variables. So, as soon as your class uses a global variable (one that applies to all instances of that class), you're forced to use the *__enter__* / *__exit__* variant.

For example:

```
_census = 0
class Person:
     def __init__(self, first, last):
          global _census            # class variable _census
          _census += 1
          self.first = first        # object vars self.first
          self.last  = last         # and self.last
     def __enter__(self):
          return self
     def __exit__(self, type, value, traceback):
          global _census
          _census -= 1
     def persons(self):
          global _census
          return _census


def temp_person():
     t = person('Opa', 'Snor')
     print('Inside temp_person: there are now', t.persons(),
           'around')
     # person t goes out of scope here, the destructor
     # fires and _census will be decreased by 1


p = Person('Klaas', 'Vaak');
```

```
q = Person('Meneer', 'De Uil');
print("There are now", p.persons(), "persons around.")     # 2
temp_person()                                               # 3
print("There are now", p.persons(), "persons around.")     # 2
```

The with Statement and its Scope Control

General sequences such as the above in the form

x = whatever()

x.do_something()

x.do_some_other_thing()

x.finish()

have the potential pitfall that *x.finish()* is forgotten. In the input/output example above, we had:

```
f = open('/etc/bashrc', 'r')
print(f.read())
f.close()
```

Forgetting *f.close()* would leave the file open and possibly cause file descriptor exhaustion, even though *f.close()* is an integral part of the file destructor. The problem with the above sequence is that the destructor of the file object doesn't have to fire for some time.

For this, Python as the *with* statement. The above example can be rewritten as follows, which makes absolutely sure that the destructor goes off:

```
with open('/etc/bashrc', 'r') as f:
    print(f.read())
    # here f goes out of scope, so that the destructor fires,
    # and the file is closed
```

Static Variables in Functions

Everything's an object, even plain functions. If you ever need to create a static local variable (i.e., one that doesn't change between calls), then you can do so using the bulit-in function *hasattr(object, name)* that returns *True* when an object has the attribute *name*.

E.g., here is a function that mis-uses a global variable to emulate a static local variable. The purpose of the function is to return the number of times it was called in the program. The global variable of course pollutes the global namespace:

```
ncalled = 0                           # uggh.. global variable
def howmany():                        # needed to hold state
    global ncalled
    ncalled += 1
    return ncalled
print('howmany:', howmany())         # 1
print('howmany:', howmany())         # 2
print('howmany:', howmany())         # 3
```

Here is an improved version with a truly local static variable. At the first invocation, *hasattr(howmuch, 'callings')* is *False*. Hence, *howmuch.callings* is instantiated to the value zero. On subsequent invocations, *hasattr(howmuch, 'callings')* is *True*.

```
def howmuch():
    if not hasattr(howmuch, 'callings'):
        howmuch.callings = 0
    howmuch.callings += 1
    return howmuch.callings
print('howmuch:', howmuch())         # 1
print('howmuch:', howmuch())         # 2
print('howmuch:', howmuch())         # 3
```

Inheritance

Inheritance (even multiple inheritance) is defined at the *class* specification. Following the classname, all base classes are stated, separated by spaces.

The constructor can specifically call *Baseclass.__init__(self, optional-args)* to invoke the superclass-constructor:

```
class Employee(Person):
    def __init__(self, first, last, salary):
        Person.__init__(self, first, last)
        self.salary = salary
    def earns(self):
        return self.salary
```

In the example class *Employee* the member function *__eq__()* should probably also be overridden, in order to correctly match persons with the same name but different salaries. That's of course left to the reader.

Method Overloading and Virtual Methods

Overloading methods in derived classes is no problem at all. Python is a dynamic language, which means that all bindings are determined at run time anyway. This emulates virtual dispatching.

```
class Base():
    def a(self):
        print('method a() of Base class')
        self.b()
    def b(self):
        print('method b() of Base class')
class Derived(Base):
    def b(self):
        print('method b() of Derived class')

print('Base test', '-' * 60)
x = Base()
x.a()
print('Derived test', '-' * 60)
y = Derived()
y.a()
# Output:
# Base test ----------------------------------------------------
# method a() of Base class
# method b() of Base class
# Derived test -------------------------------------------------
# method a() of Base class
# method b() of Derived class
```

Python has no concept of pure virtual functions, though this can be kind-of emulated using *hasattr()*. If you suspect in a baseclass that a derived class *might* have a method *b()*, then you can determine its existence and call it if you want. The gain when compared to a 'do-nothing' method *b()* in the base class is however absent:

```
class Base():
    def a(self):
        print('method a() of Base class')
        if hasattr(self, 'b'):
            self.b()
class Derived(Base):
    def b(self):
        print('method b() of Derived class')

print('Base test', '-' * 60)
x = Base()
x.a()
print('Derived test', '-' * 60)
y = Derived()
y.a()
# Output:
```

```
# Base test ---------------------------------------------------
# method a() of Base class
# Derived test ------------------------------------------------
# method a() of Base class
# method b() of Derived class
```

Custom Exceptions

Exceptions are of course also instances. The base class is *Exception* and you can subclass it to make specific exception types.

The most simple definition is a class, based on *Exception*, with no members at all (just a *pass* statement to satisfy the prototype):

```
class PersonException(Exception):
    pass
```

The exception is used as e.g.:

```
class Person:
    def __init__(self, first, last):
        if first == '': raise PersonException('empty first name')
        if last == '':  raise PersonException('empty last name')
       # ... etc
```

Exercise

In this exercise we will write write a solver for the 8-queens problem, which is: How do you place 8 queens on a chess board, without any one being able to take another?

First, write a class *Board* to support the chess board given this problem. Save it in a file *board.py*. The interface is defined as follows:

```
NAME
    board - Representation of a chess board where queens may be placed.

CLASSES
    builtins.object
        Board

    class Board(builtins.object)
     |  Methods defined here:
     |
     |  __init__(self, size=8)
     |      Initializes the board as a matrix, default 8x8. A size
     |      may be given, e.g.:
```

```
 |        b = Board()        # 8x8
 |        b = Board(10)      # 10x10
 |      The actual size after construction can be retrieved as b.size
 |
 |  __repr__(self)
 |      Returns a string representation of the board. Positions occupied
 |      by queens are shown as asterisk, empty positions as a period.
 |
 |  allowed(self, x, y)
 |      Returns True if the position at (x,y) is free to receive a
 |      queen, else False.
 |
 |  occupied(self, x, y)
 |      Returns True if the position at (x,y) is occupied by a queen,
 |      else False.
 |
 |  place(self, x, y)
 |      Places a queen at (x,y). Caller is responsible for checking
 |      that this move is allowed.
 |
 |  remove(self, x, y)
 |      Removes a queen at (x,y).
```

Next write the solver as follows:

- The solver imports class *Board* from *board.py* and instantiates a default board of 8x8.
- It maintains a last-used X position for each column on the board. The initial values are -1, meaning: the queen is off the board (value 0 would mean: on the field of the column, 1 would mean: on the second field, etc.). For example:
  ```
  b = Board()            # board with default size
  xpos = [-1] * b.size   # [ -1, -1 ... ], b.size elements
  ```
- Write a function *solve(y)* which solves for column *y*. The function is wired as follows:
  - Fields are inspected from *xpos[y] + 1* to the board size *b.size*
  - If a queen is allowed on the given field, then it is placed there and *xpos[y]* is updated with the coordinate.
  - If the last column (which is *b.size - 1*) was thus filled, then the function may return *True*.
  - Else, the function calls itself recursively to fill the next coordinate; i.e., the call is *solve(y + 1)*. If the outcome of that call is *True* then we're done and we may ourselves now return *True*.
  - If solving at *y+1* didn't work, then the queen at our position is removed, *xpos[y]* is reset to -1 (so the queen is placed off the board), and the next position is tried in the same fashion.
  - When all positions, up to *b.size-1* have been tried, then there is no solution given the configuration of the previously placed queens. The queen at the current *y* is removed from the board, and *False* is returned.
- The top level call to the solver is given below.

```
if solve(0):
      print('Solution found:')
      print(b)
else:
      print('No solution...')
```

Concurrent Processing

Concurrent processing means that (sub)tasks are processed in parallel, and that their results are evaluated once they become available. Python supports of course threading, but also *process pool execution* which is a higher abstraction. Process pool execution can mean forks, asynchronous IO, or threading - whichever is available and most appropriate.

Threading

You can wire your own threads using the module *threading*. The module supports two modes: functional (you ask *threading* to run a function for you in a separeate thread), or OO-style (you derive a subclass and implement the actions to take in a method).

Functional API

Your function that is run as a thread must accept the following parameters:

- Optional arguments **args*
- Optional keyword arguments ***kw*

Here is an example of such a function that uses only **args*:

```
import time
def countdown(*args):
    top = args[0]
    bottom = args[1]
    for i in range(top, bottom - 1, -1):
        print('Countdown at', i)
        time.sleep(1)
    print('BOOM')
```

After *import threading* the class *threading.Thread* is available. Its constructor can be run with the arguments *target=myfun, args=(a,b,c), kwargs={...}*:

```
import threading
threading.Thread(target=countdown, args=(7,1)).start()
threading.Thread(target=countdown, args=(3,1)).start()
```

By default *threading.Thread* will wait for all threads to finish before stopping the program. You can explicitly wait for a thread to finish using *.join()*:

```
mythread = threading.Thread(target=countdown, args=(3,0)).start()
mythread.join()
```

OO-style API

Alternatively, you can subclass *threading.Thread* and implement your action code in the function *run()*.

threading.Thread is the base class from which thread-able tasks can be derived. A subclass:

- Must implement the constructor *__init__()* where the super-constructor is called;
- Must implement the method *run()* where the tasks actions are performed;
- Is started using *mytask.start()*
- Can be waited for using *mytask.join()*
- Data member *mythread.name* is an assigned identifier, unless overruled in the constructor
- Etc., see the docs. for e.g. *threading.Lock* (mutexes) and *threading.Event* (inter-thread communications)

Below is a simple threading example. There are two tasks; both print some stuff and sleep for 1 second between the prints. The task constructors expect an integer *max* to control how many loops will be iterated. The main program instantiates the tasks, starts them, and waits for them to finish.

```
import threading
import time

# Prints loop 0, loop 1 etc. until MAX.
# Sleeps 1 second between prints.
class Task1(threading.Thread):
    def __init__(self, max):
        threading.Thread.__init__(self)
        self._max = max
    def run(self):
        for i in range(0, self._max):
            print(self.name, 'loop', str(i))
            time.sleep(1)

# Prints countdown MAX, countdown MAX -1 etc. until 1.
# Sleeps 1 second between prints.
class Task2(threading.Thread):
    def __init__(self, max):
        threading.Thread.__init__(self)
        self._max = max
    def run(self):
        for i in range(self._max, 0, -1):
            print(self.name, 'countdown', str(i))
            time.sleep(1)
```

```
# Define and start two threads.
a = Task1(5)
a.start()
b = Task2(6)
b.start()

# Wait for them to finish.
a.join()
b.join()
print('done')
```

Exercise

Write a threaded class *Primetest* with the following properties:

- It is instantiated using e.g. *a = Primetest(1234567)*. The parameter is a number to test for prime-ness.
- The testing is started using *a.start()*.
- After execution, the result of prime-ness is seen in the data member *a.isprime* which is *True* or *False*.
- The tested number is available as data member *a.number*.

Test the program using the following snippet:

```
import threading

class Primetest(threading.Thread):
    # Your magic goes here

a = Primetest(999983)
b = Primetest(999999)
a.start()
b.start()
a.join()
b.join()

for t in (a, b):
    if t.isprime:
        print(t.number, 'is a prime')
    else:
        print(t.number, 'is NOT a prime')
```

Process Pool Execution

Module *concurrent.futures* introduces the concept of a "future" to Python. A future is simply an object that will become available at a future time, and at that time *future.result()* will yield something meaningful.

A future is obtained after submitting a task to the process pool. This is done using an *executor*. For every task, an executor is created from the pool, and a task is submitted using *executor.submit(function, args)*. The executor will return a future which contains the scheduling of *function(args)*. Once that is finished, the future's *result()* can be obtained (which is what *function(args)* returns).

Whether the execuctor's task runs parallel to other tasks usually depends on the system-specific implementation. Usually the number of concurrent executor tasks is determined by the number of cores on your system (or in other words: tasks are assigned to as many threads as there are cores).

Here is a trivial example.

```
from concurrent.futures import ProcessPoolExecutor
import time

def myfun(nr):
    print('Myfun: called with number', nr,
          'and now taking a nap')
    time.sleep(3)
    print('Myfun: woke up and giving you back your number', nr)
    return nr

executor = ProcessPoolExecutor()
future = executor.submit(myfun, 12)
print('Future result is:', future.result())
```

The example is simplistic, because

1. Only one task is run, in an otherwise linear program. Starting a parallel process and then waiting for it while doing nothing won't save you any time.
2. The "wait for finish" occurs in the call *future.result()*. This will block until the future has finished running. In a real life situation, you'll want to check the future state before calling *result()*, or at least wait until a few futures are finished and then start pulling results.

Most often you'll have a list of futures (pending tasks, running in parallel) and want to wait until they have finished. Method *as_completed()* does that. It iterates over a list of futures, and returns one by one the finished future objects.

In order to illustrate this:

```
from concurrent.futures import ProcessPoolExecutor, as_completed
import time

def myfun(nr):
    print('Myfun: called with number', nr,
          'and now taking a nap')
    time.sleep(3)
    print('Myfun: woke up and giving you back your number', nr)
    return nr

futures = []                                  # list of underway
tasks
for nr in range(20, 25):
    executor = ProcessPoolExecutor()
    future = executor.submit(myfun, nr)
    futures.append(future)                    # add to list of tasks

for future in as_completed(futures):          # what’s finished
    print('Future result is:', future.result())
```

The statements around the *for* loop could of course be written in a more compact way by eliminating locally used variables:

```
futures = []
for nr in range(20, 25):
     futures.append(ProcessPoolExecutor.submit(myfun, nr))
```

Exercise

In this exercise write a program that factorizes numbers in parallel. The factors of a given number *n* are the prime numbers between 1 and *n* (exclusive) by which *n* can be divided without a remainder. E.g., the factors of 14 are 2 and 7. The factors of 105 are 3, 5 and 7. Note that the factors of 315 (which is 3 * 105) are also just 3, 5 and 7 - and not 3, 3, 5 and 7.

First write a function *factorize()* that receives one argument, the number to factorize. Test it:

```
print(factorize(105))                         # [3, 5, 7]
print(factorize(315))                         # [3, 5, 7]
```

Next, for comparison, try the following function *factorize_pool()*. It receives a list of numbers to factorize as arguments, and returns a dictionary of results. The keys in the returned dictionary are the factorized numbers. The values (per key) is a list of the factors:

```
def factorize_pool(*nrs):
    ret = {}
    for n in nrs:
        ret[n] = factorize(n)
    return ret
print(factorize_pool(14, 105, 305))
# {
#   14: [2, 7],
#  105: [3, 5, 7],
#  315: [3, 5, 7]
# }
```

Try running *factorize_pool(10, 99999989, 99999991, 99999993)*. How long (in seconds) does it run?

Next, rewrite the function *factorize_pool()* so that:

- For every number-crunching task an *executor* is obtained
- A parallel task is submitted using *executor.submit(),* with the arguments (a) the function to call, which is *factorize()*, (b) the argument to that function, the number
- The thus obtained future is appended to a list of running tasks
- Once the tasks have finished (hint: use *as_completed()*), the return dictionary is constructed and returned.
- Check that the return value is identical to the non-parallel version given above.

How long does the parallel version run?

HTTP and JSON

There are many frameworks for HTTP servers: from *twisted* (a network event server, which includes the HTTP protocol too), up until *django* (a production-ripe HTTP server that implements goodies for app development such as MVC, ORM).

Tornado HTTP Server

A nice and simple middle class is *tornado*. This is a full-fledged HTTP framework, but without e.g. *django's* enforced directory structure and the such. A *tornado*-based HTTP server is built as follows:

- First, the application entry points are defined. An entry point definition is a regular expression to match a request URI, and a subclass of *RequestHandler* to handle hits.
- The thus created application is instructed to listen to a given port.
- Finally the IO loop is started:

```
from tornado.ioloop import IOLoop
from tornado.web import RequestHandler, Application, url

# RequestHandler-derived classes are defined for
# specific situations:
class WelcomeHandler(RequestHandler):
    def get(self):
        self.write('Hello there.')
class UploadHandler(RequestHandler):
    def post(self):
        ...

# GETs  to /index.html will be handled by the WelcomeHandler
# POSTs to /upload/... will be handled by the UploadHandler
Application( [ url(r'^/index.html$', WelcomeHandler),
               url(r'^/upload/',     UploadHandler),
             ] ).listen(8888)

# Event loop starts.
IOLoop.current().start()
```

Inside a *RequestHandler*-derived class, the following goodies are available:

- *def get(self), post(self), put(self), delete(self), head(self):* These methods can be defined to implement what the server should do during a GET, POST, PUT, DELETE or HEAD request.
- *self.write(data)*: Writes data to the client. The data can be a string (in which case

it's assumed to be text/html), or it can be a dictionary (in which case it is sent as JSON with content-type text/json).

- *self.get_argument(name)*: returns the value of the named argument in GET style request
- *self.get_body_argument(name):* returns the value of the argument in a POST style request
- ... and many others.

Exercise: JSON API Server

In this exercise we will implement a JSON-style API to a service that allows one to list staff members, to modify them, to delete, or to add. The primitives are given in the following class *Staff*. The class internally simply has a dictionary of numeric ID's, each leading to a dictionary that represents a staff member (*f* for first name, *m* for middle name, *l* for last name, *e* for e-mail address).

Save this into a file *staff.py*.

```
import copy

# Initialization of staff (would normally come out of a db)
staff = {
    1: { 'f': 'Harry', 'l': 'Kodden',
         'e': 'harry@e-tunity.com' },
    2: { 'f': 'Peter', 'l': 'Bosch',
         'e': 'peter@e-tunity.com' },
    3: { 'f': 'Jelly', 'l': 'Vaartjes',
         'e': 'jelly@e-tunity.com' },
    4: { 'f': 'Emiel', 'm': 'van', 'l': 'Rooijen',
         'e': 'emiel@e-tunity.com' },
    5: { 'f': 'Riemer', 'm': 'van der', 'l': 'Kleij',
         'e': 'riemer@e-tunity.com' },
    6: { 'f': 'Martijn', 'm': 'van der', 'l': 'Molen',
         'e': 'martijn@e-tunity.com' },
    7: { 'f': 'Karel', 'l': 'Kubat',
         'e': 'karel@e-tunity.com' }
    }

# Representation of the staff
class Staff:

    # Constructor, loads initial staff
    def __init__(self):
        global staff
        self.staff = staff

    # List everyone
    def list_all(self):
        return copy.deepcopy(self.staff)

    # Return staff member given an ID
    def get_entry(self,id):
        if not id in self.staff:
            raise Exception('no such id ' + str(id))
        return copy.deepcopy(self.staff[id])

    # Add new member
    def add_member(self, member):
        maxid = -1
        for i in self.staff:
            id = int(i)
            if id > maxid:
                maxid = id
        maxid += 1
        self.staff[maxid] = member
        return maxid

    # Update existing member
    def update_member(self, id, member):
        if not id in self.staff:
            raise Exception('no such id ' + str(id))
        self.staff[id] = member

    # Remove existing member
    def remove_member(self, id):
        if not id in self.staff:
            raise Exception('no such id ' + str(id))
        del self.staff[id]
```

Note that the above methods that return a dictionary, actually return a deep copy (and not a reference). That way, the caller may modify the obtained value as they like, without reflecting the object's own data *self.staff*.

Next, implement a RESTful[15] API in the Tornado HTTP server. As for the RESTful properties:

1. GET calls don't modify, they request a listing. For example, a GET call on URI */staff* will list all staff members, a call on */staff/1* will list staff member with ID 1 etc.
2. POST calls create new nodes when used on a URI that identifies a 'collection'. For example, a POST call on URI */staff* will create a new member under a new ID. The call must require the CGI variables *f*, *l* and *e* (first, last, e-mail), a middle name *m* is optional. This call must return the just-created ID to the caller using JSON output (see below).
3. POST[16] calls on a URI that's not a collection but an end-node, replace the node. For example, a POST call on URI */staff/1* replaces the data of the staff member with ID 1. The CGI variables *f*, *l* and *e* are required, *m* is optional.
4. DELETE calls remove nodes. For example, a DELETE on URI */staff/1* deletes the entry with ID 1.

As for the inputs, the data may be passed as form variables (CGI-encoding). REST isn't strict here; one can use CGI encoding, or anything else (XML in the body, JSON in the body, etc.).

As for the outputs, the web services must return a JSON document which is a literal representation of what's in class Staff (ergo, a dictionary in JSON format). Sometimes this is the full staff dictionary (as in the URI */staff*), sometimes only what's pointed to by a given ID (as in the URI */staff/3*). After the creation of a new staff member, the API must return a key *id* with the value of the newly created ID.

The output must also contain a key *result*, having normally the value *ok*. When an error is seen, *result* must be set to *error*, and a new key *reason* must state the reason of failure.

Here are some guidelines for this exercise.

1. Implement the web API in one file, e.g., *RESTapi.py*. In this file, include *staff.py* as an abstraction of the data layer. After importing, use a global variabe *staff = Staff()* that's manipulated using *Staff*'s methods *list_all(), get_entry(),* etc..
2. The API has just two relevant URI's. Implement them in two classes. The switch-URI's are:
   a. */staff,* which works on a staff member collection (supporting GET for listing and POST for creating)
   b. */staff/NUMBER*, which works on a singular staff member (supporting GET for listing, POST for modifying, DELETE for deleting).
3. Both classes of the above point will need to decode form data and check that the relevant fields are there (*f, l, e* and optional *m,* but no other). If not an error must

[15] REST stands for Representational State Transfer which is a fancy word for Remote Procedure Call. From time to time we need new names for old stuff.

[16] Some REST-evangelists argue that this should be a PUT instead of a POST. Some REST-programmers argue that POST's are more ubiquitous in client API's and therefore are just as acceptable.

be returned. Of course, you don't want to type this code twice or copy/paste it. You just want to code this once, in a helper method called e.g. *staff_form_data()*. What's the best way of having two classes share a method?

4. Some methods from *staff.py* may throw exceptions. Make sure that the RESTful API returns the right answer, which is a dictionary {*"result":"error", "description":"TEXT"*}. The right exception text must be in the *description* key.

Tornado HTTP Client

Tornado of course also implements an HTTP client class:

- This requires *HTTPClient* and *HTTPRequest* from *tornado.httpclient*.
- Firing an HTTP request and receiving a response involves the following steps:
    - An HTTP client is instantiated using *cl = HTTPClient()*
    - A request is created, using *rq = HTTPRequest(....)* See the below bullet for more information.
    - The request is sent, a response is received: *resp = cl.fetch(rq)*
    - The response can be examined for status codes etc..
    - The body in the response is *resp.body*. This is a byte-blob. To convert it to e.g. a string (assuming UTF-8 encoding) is done via *resp.body.decode('utf-8')*
- The creation of an *HTTPRequest* involves calling the constructor with at least 1 string argument: the URL. Optionally keywords *method* and *body* can be given to override the defaults *GET* and *None*.
- When POST-ing CGI-encoded variables, *urllib.parse.urlencode()* (an import from *urllib*) can be used to create the body as a sting. E.g, variables *min* and *max* with the values *20* and *30*, can be encoded into a body using *urllib.parse.urlencode({ 'min':20, 'max':30 }).*

Here is an example of a GET and a POST:

```
from tornado.httpclient import HTTPClient, HTTPRequest
import urllib

# Simple GET example, no POST-ed data
cl = HTTPClient()
rq = HTTPRequest('http://www.kubat.nl')
resp = cl.fetch(rq)
body = resp.body.decode('utf-8')
print(body)

# POST example
vars = { 'email':'karel@kubat.nl', 'password':'secret' }
cl = HTTPClient()
encbody = urllib.parse.urlencode(vars)
rq = HTTPRequest('http://www.kubat.nl/login', method='POST',
                 body=encbody)
```

```
resp = cl.fetch(rq)
body = resp.body.decode('utf-8')
print(body)
```

Exercise: JSON API Client

Given the above JSON API server, now implement a client for the staff API. The client can be implemented as a series of stand-alone functions (or, if you want to goldplate it, wrap it in a class). The API must offer at least:

- *listall()* - returns a dictionary of all ID's and underlying staff members
- *list(3)* - returns a dictionary of the staff member at the given ID, here 3
- *add(f='Adam'', l='Boon', e='adam@gmail.com'):* adds a staff member. Keyword *m* for middle-name must also be supported. The returned dictionary must of course contain the server's output, which is the ID at which the staff member was added.
- *modify(3, f='Adam', l='Boon', e='adam@gmail.com'):* Modifies an existing user at the given ID. Keyword *m* for middle-name must also be supported.
- *delete(3)*: Deletes the user at the given ID.

After each call to the server, the client API must check that the returned result is *ok*. Else the appropriate error message must be displayed.