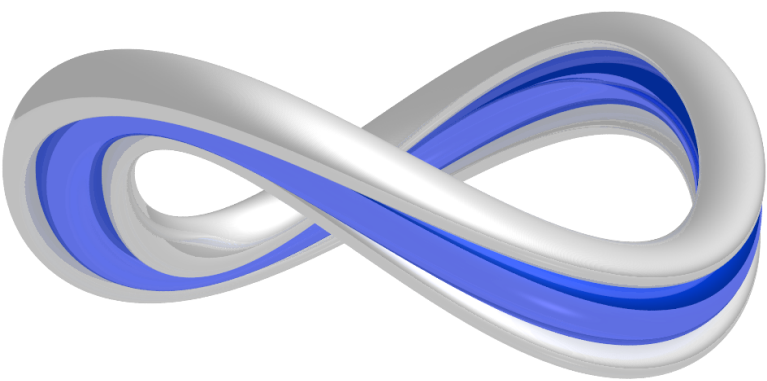
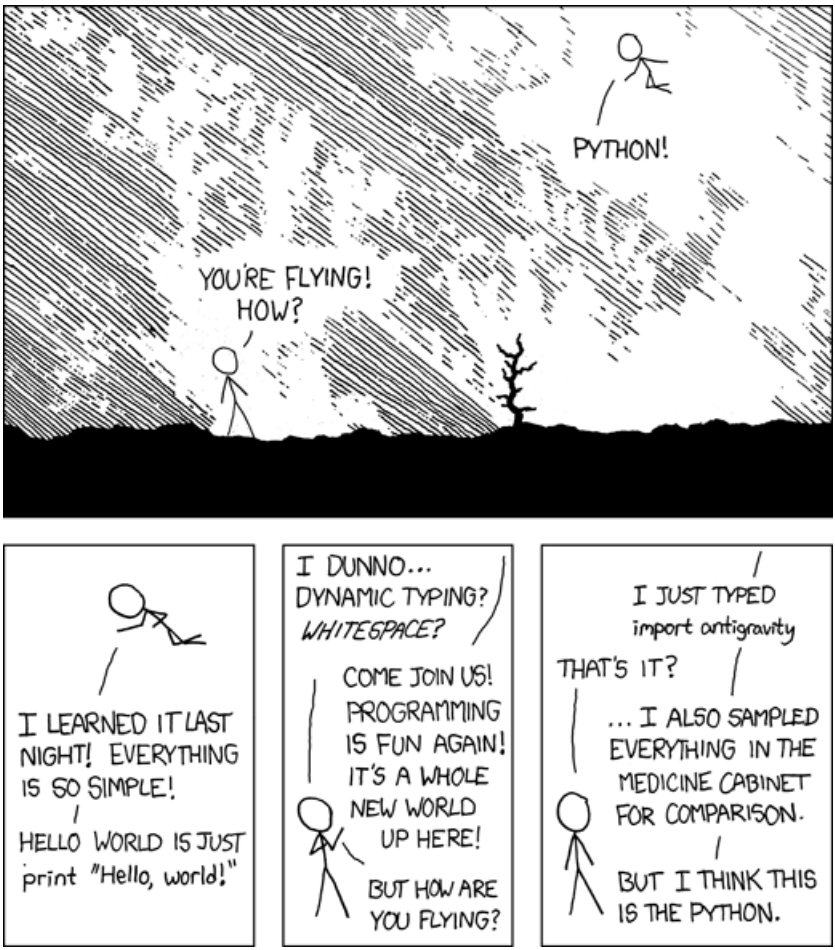
# Python Primer

Ruben S. Andrist

Santa Fe Institute






Comic strip from xkcd, `http://xkcd.com/353/`

## Introduction

Python is a scripting language that is designed to be intuitive and easy to learn, yet both powerful and diverse in its applications. With lots of built in functionality, it can be used for small scripts, data analysis tools as well as larger programming projects.

## Prerequisites

Visit the python website, `http://python.org` for details on how to install python on your particular platform (or use the built-in package manager). On Linux and MacOS, python is generally preinstalled, though not necessarily in the desired version. On Windows you should try the "pythonXY" distribution, `http://code.google.com/p/pythonxy/`, which bundles several useful scientific tools. The tutorial itself uses python 2.7 and IDLE, python's integrated development environment. However, up to some subtle differences, it should be compatible with other versions and your preferred editor or shell.

## Notation

```
>>> #this is a interactive python shell example
>>> print "showing both input and output"
showing both input and output
>>>
```

**Python source file** `code/intro.py`

```
1 # This is python code from a file, which you can
2 # find in the collection of source files
3 text = "You can download the files, just go to:"
4 url  = "http://www.andrist.org/teaching/python"
```

**Caution**: Common pitfalls and caveats are indicated like this.

**Hint**: Useful pointers and remarks are shown with a light bulb.

## License

## Zen of Python

Python has a built-in "Easter egg" that highlights some of the guidelines to writing good – or as it is called *pythonic* – code. Most of them are aimed at keeping your code simple, more accessible and less error prone:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

## Further Reading

Several well written and more thorough introductions to the python language are available online or as a book. If you are interested in further reading, I recommend the following sources:

- Official tutorial: `http://docs.python.org/tutorial/index.html`
- Python Style Guide: `http://www.python.org/dev/peps/pep-0008/`
- Dive Into Python 2: `http://www.diveintopython.net/toc/index.html`
- Dive Into Python 3: `http://getpython3.com/diveintopython3/`
- SciPy Lecture Notes: `http://scipy-lectures.github.com/`

# Contents

# Chapter 1

# Getting Started

Python is an interpreted (as opposed to compiled) language, which means that it is not translated to native machine code (like `C++`) but rather to an intermediate "byte code" which is then interpreted by a virtual machine as it is run. While I will not bore you with technical details, this implementation decision makes python slightly slower than, for instance, `C++`, but also much more flexible. A very useful side effect is that python can be run as an interactive shell: In the shell you can type python statements and immediately see their effect on the screen.

Let's try this with a few simple print statements. When starting IDLE on Windows, you should get a shell window automatically (it can be identified by the command prompt `>>>`). On Unix systems you can invoke the interactive python shell in a terminal with the command `python` or `python2.7` (if you have multiple versions installed). Test it by typing the following commands:

```
>>> print 'Hello world'
Hello world
>>> print 'The answer is', 42, '!'
The answer is 42 !
>>>
```

You can print just one item or you can provide multiple things to print, separated by a comma: In the third line we are printing two strings and a number. In the output they are separated with a space.

**Note**: Testing things in the shell is simple and fun. And if you ever get stuck, you can hit `Ctrl-C` to abort the current instruction and you can leave the interactive shell by typing `Ctrl-D`.

### `print` in python3

**Caution**: The print statement was *replaced* by a built-in `print()` function in python3. In this case you have to enclose the list of items in parentheses and the output formatting will differ slightly:

```
>>> # printing in python3.0+
>>> print('Brave', 'new', 'world')
Brave new world
>>>
```

You can start using the new print statement even in python2. Because this change has been planned for a while now, it is already included as a future package for importing: `from __future__ import print_function`. After this statement, print behaves as in python3.

### Storing a program in a file

Of course, python is not only an interactive script language: The commands that you type in the shell can also be written in a file for later (re)use. The convention is to call the file `something.py` and it will be translated to the byte-code file `something.pyc` for execution. You can safely delete any `.pyc` file but it will be recreated upon execution. In IDLE, you can create a new file with the Menu entry `File > New` and run it by hitting `F5`. On Unix compatible systems, you can run it in the shell with `$ python something.py`.

**Note**: On Unix compatible systems it is customary to write the interpreter `#!/usr/bin/env python` as the she-bang (first line) to allow direct execution via `./something.py` in the shell.

Make sure you can run code from a file using the following example:

**Python source file** `code/start/hello.py`

```
1 #!/usr/bin/env python
2
3 # let's say hello to the world:
4 print "Hello, I'm a python file!"
5
6 # run with $ python hello.py or by hitting F5
```

**Caution**: When choosing the name of your file, try to avoid the names of built in modules as this will cause problems when trying to import said modules later.

## 1.1 Assignments

Python variables do not need to be declared explicitly before using them. The declaration happens automatically when you assign a value to a variable. The equal sign "=" is used to assign values:

**variable assignment**

```
>>> age = 42
>>> name = 'Arthur'
>>> print name, age
Arthur 42
>>>
```

It is important to note that a variable *must* be assigned before using it, otherwise the interpreter will throw an exception. This is python's way of complaining that you did something wrong or ambiguous:

**variable accesss**

```
>>> print z
Traceback (most recent call last):
  ...
    print z
NameError: name 'z' is not defined
>>> z = 42
>>> print z
42
>>>
```

In some cases, it might be necessary to get rid of a variable that had previously been assigned a value (and thus been declared). It can be removed again with the `del` statement:

**deleting a variable**

```
>>> z = 42
>>> print z
42
>>> del z
>>> print z
Traceback (most recent call last):
  ...
    print z
NameError: name 'z' is not defined
>>>
```

### Native datatypes

Python has many native datatypes. Here are the important ones:

- Booleans are either `True` or `False`.
- Numbers can be `int`egers, `float`s or `complex` numbers (python uses a trailing `j` or `J` to denote the imaginary unit $\texttt{j}^2 = -1$).
- Strings are sequences of characters. They can be enclosed with single `'` or double quotes `"`; multiline strings are enclosed with `"""`.
- Bytes and byte arrays are used to store binary data.

Collections are discussed in chapter 3.

### Dynamic vs. static typing

Typing of variables is *dynamic* (as opposed to static) in python. This means that you don't need to specify the type of a variable before using it and the type can change in the course of executing your program. In fact, you rarely ever see type names in python unless you manually verify them using `type(...)`:

**dynamical typing in python**

```
>>> x = "hello"
>>> type(x)
<type 'str'>
>>> x = 42
>>> type(x)
<type 'int'>
>>> x += 13j
>>> type(x)
<type 'complex'>
>>>
```

Whenever a variable is assigned a new value, its type is dynamically changed to the type of the assigned value. For instance, when we add an imaginary part to the number in line 7, the type is automatically adjusted to `complex` to store this information.

### Strong vs. weak typing

Even though the typing is dynamic in python, it is also *strong* (as opposed to weak). This means that python will not implicitly convert the type of a variable during manipulations. It requires an explicit conversion:

**strong typing in python**

```
>>> x = "hello"
>>> print x + " world"
hello world
>>> x = 42
>>> print "answer = "+x
Traceback (most recent call last):
  ...
    print "answer = "+x
TypeError: cannot concatenate 'str' and 'int' objects
>>> print "answer = "+str(x)
answer = 42
>>>
```

The strings in the first print statement are concatenated with the `+` operator. Trying the same feat with a string and a number fails because python does not know how to add these two types and refuses to implicitly convert one or the other. To concatenate the two types, we need to first convert our number to a string. The translation is achieved by denoting the desired type to convert to. For instance, `str(x)` uses the integer type's `__str__()` function to convert it to a string.

### User input

A frequent necessity for conversion arises when dealing with user input. In python 2.x, text input can be achieved with the function `raw_input(prompt)`, which prints a prompt to the screen and returns the text entered by the user as a string:

**user input as string**

```
>>> name = raw_input("What is your name? ")
What is your name? Arthur
>>> print "Hello", name
Hello Arthur
>>>
```

This works great for string input, but for other data types we might need to convert (cast) the string to another type:

**user input as integer**

```
>>> age = raw_input("How old are you? ")
How old are you? 42
>>> print "Next year you will be", int(age)+1
Next year you will be 43
>>>
```

The explicit conversion to an integer, `int(age)`, allows adding one to the age entered by the user. Of course this will throw an exception if the user input cannot be translated to a number.

**Caution**: The `raw_input()` function was *renamed* to `input()` in python3. However, there is also a (different) `input()` function in python2.x. You should keep this in mind if your version differs from the tutorial.

## 1.2 Introspection

Python provides a number of very useful tools for *introspection*. These tools allow "examining" a variable, function or module in more detail in order to determine its type, properties and functionality. We have already encountered the `type(...)` command to inspect the type of a variable.

### `dir(...)`

The `dir()` command (without arguments) shows the objects in the current scope. If you just started the shell, this will typically only contain a few special variables with leading/trailing underlines. These are the built-in objects that are always there. Then, as soon as you create new variables, in this case `x = 42`, these also show up.

**using `dir()` to show all objects in this scope**

```
>>> dir()
['__builtins__', '__name__']
>>> x = 42
>>> dir()
['__builtins__', '__name__', 'x']
>>>
```

When applied to an object, `dir()` shows its properties and functionality. `dir()` is therefore also useful as an interactive aid for working with modu-

les and variables. For instance, we can apply `dir()` to an integer variable `x` to see what can be done with an integer:

**using `dir()` to show all properties of an object**

```
>>> type(x)
<type 'int'>
>>> dir(x)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__',
    ↪ '__delattr__', '__div__', '__divmod__', '__doc__', '__float__',
    ↪ '__floordiv__', '__format__', '__getattribute__', '__getnewargs__',
    ↪ '__hash__', '__hex__', '__index__', '__init__', '__int__', '__invert__',
    ↪ '__long__', '__lshift__', '__mod__', '__mul__', '__neg__', '__new__',
    ↪ '__nonzero__', '__oct__', '__or__', '__pos__', '__pow__', '__radd__',
    ↪ '__rand__', '__rdiv__', '__rdivmod__', '__reduce__', '__reduce_ex__',
    ↪ '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
    ↪ '__ror__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
    ↪ '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
    ↪ '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
    ↪ 'bit_length', 'conjugate', 'denominator', 'imag', 'numerator', 'real']
>>>
```

The above output shows that numbers have a lot of built-in functionality (several of the lines with underscores represent mathematical operators) and even some properties we can read directly, such as its `real` and `imag`inary parts. The latter vanishes of course for integers but is provided to ensure a common interface with the `complex` datatype.

### `help(...)`

The `help()` function provides a verbose explanation of built-in functions as well as modules and packages that you can use:

```
>>> help()
<some basic hints and pointers to tutorials ...>
>>> help(complex)
<detailed explanation of the complex data type ...>
>>> help('try')
<explanation of the try & except construct ...>
```

In the third use, it was necessary to give the keyword `'try'` as a string because it is a statement and not an object (i.e. it does not exist to python and would cause an exception).

**Note**: In the interactive shell, calling `help()` can switch your prompt to a help prompt, indicated by `help>` at the input line. There you can search the documentation directly. You can leave this mode again with `Ctrl-D`.

## 1.3 Importing modules

Python comes with "batteries included": A lot of useful functionality is already built into the standard distribution in the form of packages and modules. You can make use of these with the `import` statement, using one of the following notations:

```
>>> import os
>>> dir
<built-in function dir>
>>> dir()
['__builtins__', '__name__', '__package__', 'os']
>>> os.sys.platform
'darwin'
>>>
```

As you can verify with the `dir()` function, the import command creates a new module object with the name "`os`" in the current scope. You can then access the functions within the module with `os.functionname()`. This is the preferred way of importing modules as it does not clutter the current scope beyond the single variable that is created. Alternatively, it is also possible to selectively import only some functions from within the module. These are then placed directly in the current scope and can be accessed without writing the module name first. Optionally, you can rename imports with `import as`.

```
>>> from math import sqrt as wurzel, pow
>>> dir()
['__builtins__', '__name__', '__package__', 'os', 'pow', 'wurzel']
>>> wurzel(42)
6.48074069840786
>>> pow(2,3)
8
>>>
```

For the truly lazy, it is also possible to include all functions within a module and import them into the current scope using `from package import *`. While this saves the programmer from typing a few extra letters, it is not advisable as it imports several unused functions and clutters the current scope unnecessarily.

The following example shows that importing `*` from `math` populates the current namespace with several objects (mostly mathematical functions), whereas `import math` would add only a single object:

```
>>> from math import *
>>> dir()
['__builtins__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh',
    ↪ 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees',
    ↪ 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
    ↪ 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp',
    ↪ 'lgamma', 'log', 'log10', 'log1p', 'modf', 'os', 'pi', 'pow', 'radians',
    ↪ 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> sin(pi)
1.2246467991473532e-16
>>> sin(pi/2)
1.0
>>>
```

**Caution**: The os module should not be imported with `from os import *`, because the `os.open()` function would shadow the built-in function `open()` which is needed for file access.

## 1.4 Exercises

1.1 **Spring cleaning**: Use `dir()` to see what variables you have created in the current session. Use the `del` instruction to clean up all except the built-in ones (those with double underlines). Now the session is good as new! spring_cleaning.py

1.2 **Number of digits**: Write a program that converts a number to a string and then prints the number of digits (by looking at the string length with the `len()` function). number_of_digits.py

1.3 **Very long number**: What is the longest number you can produce on the screen by typing just 5 characters (followed by ENTER) into the interpreter? (What counts is the number of digits printed.) very_long_number.py

1.4 **Hello programmer**: Write a program that asks the user for her name, then greets her with a personalized message. You can use `raw_input()` to read what the user types and `+` for string concatenation. hello_programmer.py

1.5 **Who am I?** Import the `platform` module and use introspection to find out how the operating system can be determined with this module. who_am_i.py

1.6 **Now**: Import the `datetime.datetime` module and try to print the current date and time to the screen (formatting does not matter for now). now.py

# Chapter 2

# Building Blocks

Many programming languages use either curly braces `{...}` or surrounding keywords (`if/then/endif`, `while/do/done`, etc.) to group several statements into a *code block*. This is typically necessary whenever the statements in that block are part of a loop, conditional or other flow control structure. In addition, programmers are also expected (but not required) to use indentation to visually indicate this grouping of statements for better readability.

Because this is redundant, python has chosen a different route by *requiring* the programmer to use *logically correct indentation* for all code blocks. For example, to make the next four statements part of a code block, you simply indent these four lines, then resume without indentation. As a result, whitespace at the beginning of the line has syntactic meaning and python code is only functional when it is formatted in a readable way. How great is that? Any working python code snippet is also properly indented!

While whitespace with meaning seems scary at first, it is really more "visual structure with meaning", because only the indentation (leading whitespace on each line) matters. The following sections will illustrate how this grouping mechanism is used to form conditional statements, loops and (in the following chapters) functions and classes.

**Caution**: You should not mix tabs and spaces in your indentation (and as of python3 it is forbidden). IDLE and most editors can be configured to use one or the other exclusively. The de facto standard is to use four spaces.

## 2.1 Conditionals

As an example, let us consider how conditionals are formed in python. Keep in mind that python does not use {...} to enclose code blocks, only a colon to start the indented block.

```
# a typical politician:
personal_opinion = "YES"
public_approval = 0.42
bribe_for_no = 5000

if bribe_for_no > 10000:
    vote = "NO"
elif public_approval < 0.45:
    vote = "NO"
elif public_approval > 0.55:
    vote = "YES"
else:
    vote = personal_opinion
```

The four conditional statements `if-elif-elif-else` are indented the same, therefore they are in the same code block. For these chained conditionals, the first block whose condition evaluates to `True` is executed (or the `else` block, if none of the conditions are matched).

**Note**: The conditions for `if` and `elif` do not need to be enclosed in parentheses (contrary to C++ or Java). Also, python does *not* allow assignments within a comparison: Only `==` (equality test) is allowed, while `=` (assignment) is a syntax error.

### Short version

To enhance readability in certain situations, python does allow the following abbreviation to be used for short conditionals:

```
>>> x = 42
>>> if x > 20: x = 13
...
>>>
```

A short, single instruction that is executed conditionally may be written on the same line as the condition, i.e. directly after the colon ":". In this case, the following line *must not* be indented to close the block. In the interactive shell, you have to undo the indentation and press Enter twice to end the indented block (as illustrated above).

### Ternary operator

An assignment that implements a conditional choice typically takes four lines, but for short values/conditions this can be written with the ternary operator:

```
>>> if x > 20:
...     x = 42
... else:
...     x = 13
...
>>> # can be shortened to:
>>> x = 42 if x > 20 else 13
```

The functionality corresponds to the `( condition ? a : b )` operator found in many programming languages, but this notation aims for better readability: "take value `a if condition` holds, otherwise take value `b`".

## 2.2 Loops

Loops are used to perform repetitive tasks in programs. The most prominent example is the counting loop that iterates over a range of integers. In python syntax, this is written as:

```
>>> # counting from 0 to 9
>>> for i in range(10):
...     print i,
...
0 1 2 3 4 5 6 7 8 9
>>> # counting from 3 to 12
>>> for i in range(3,13):
...     print i,
...
3 4 5 6 7 8 9 10 11 12
>>> # counting from 0 to 18 in steps of 2
>>> for i in range(0,20,2):
...     print i,
...
0 2 4 6 8 10 12 14 16 18
>>>
```

The range function `range(a,b,c)` starts counting from `a` and counts up to (but not including) `b`, with a step size of `c`. The default step size is 1.

**Note**: The trailing comma after the print instruction has the effect that no newline is printed (to show the numbers on the same line). In python3, you can add the keyword argument `end=''` to the print function to achieve the same result.

### While

An alternative loop is the `while` loop, which is repeated as long as the given condition holds at the beginning of each iteration:

**Python source file** code/blocks/while.py

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

answer = -1
while answer != 42:
    answer = int(raw_input("What is the answer? "))
print "correct, the answer is 42!"
```

As for conditional statements, the condition does not need to be enclosed in parentheses. Make sure to check that your condition can actually be `false`, otherwise you are stuck in an endless loop!

### Continue, break and else

As in other languages, you can use the statement `continue` to jump to the top of the loop (i.e., for the next iteration) and the `break` statement to exit the current innermost loop.

A peculiarity of python is that it also allows an `else` block which directly follows after a loop. This `else` block is executed only if the loop terminates regularly (due to end of the `range` or mismatch for the `while` condition), but is *ignored* if the loop was exited manually with `break`.

**Python source file** code/blocks/forelse.py

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

for i in range(20):
    if i==13:
        break
else:
    # this will not be printed because
    # the loop is exited with break
    print "end of first loop reached"

for i in range(20):
    if i==23:
        break
else:
    # this will be printed because the
    # loop reaches the end
    print "end of second loop reached"
```

## 2.3 Try & except

You will notice that your program "crashes" frequently during development – whenever an exception arises, this halts the entire program:

```
Errors should never pass silently.
Unless explicitly silenced.
```

While annoying at first, this is a *good* thing because it allows you to correct the error right then and there rather than having to dig through lots of code to find it later. However, as line 14 of Python Zen already hints, there are situations where we might expect exceptions and we want to specify how to handle them. Frequent examples are testing for variable existence and handling user input: In each case an exception will be thrown, which we can handle explicitly with the statements `try` & `except`:

**Python source file** `code/blocks/tryexcept.py`

```
1  #!/usr/bin/env python
2
3  x = None
4  while type(x) != int:
5      try:
6          x = int(raw_input("Type a number: "))
7      except ValueError:
8          print "That is not a number, try again"
9
10 print "The number is", x
```

This example indicates how `ValueErrors` caused by non-translatable user input can be handled with a try block. This program will keep asking for a number input until it receives a string that can be converted without error.

**Note**: For now, we only consider the built-in exceptions [see `dir(__builtins__)`]. More information on how to derive your own exceptions will be provided in the section on python `class`es.

### Finally

The keyword `finally` is not intended for exception handling but for cleanup purposes: If an exception causes the try block to be terminated prematurely, there might be some instructions that need to be carried out nonetheless for the program to continue running. This typically includes cleanup of the mess caused by the error and/or providing a return statement for the current function.

Consider the following example of handling a “name undefined” exception:

**Python source file** `code/blocks/tryfinally.py`

```
1  #!/usr/bin/env python
2
3  try:
4      print x
5  except NameError:
6      print 'x does not exist'
7  except:
8      print 'unknown error occurred'
9      raise
10 finally:
11     print 'lets forget about that ...'
12
13 print 'but at least we are still running!'
```

The exception handler on line 5 has the matching type, it will be executed and the error is considered to have been noticed and handled. For completeness, line 7 shows the syntax for a generic exception handler: It has no particular type specified and *catches all error types* (you probably already guessed that it is not very pythonic to claim you can fix any error). We have the option to re-raise the error (line 9) in case we have *not* handled it (this allows other `except` blocks at a higher level to handle it). An exception whose type matches no `except` block (or which is reraised while handled) and reaches the top level of the program will cause it to halt and display the exception on the screen.

This example illustrates the different purposes of except and finally: If an exception arises during the try block, its type is compared to all the except blocks and it is handled by the first matching one. The finally statement is run after exception handling has been carried out successfully. It is a way for the programmer to perform the necessary cleanup for the preceeding `try` block which was aborted prematurely due to the exception.

## 2.4 Exercises

2.1 **Calculator**: Write a program that asks for two numbers and prints their sum, difference, product and ratio. You can use the `raw_input()` instruction to read user input, but make sure to convert the input string to an integer: `int(string_var)`. calculator.py

2.2 **Primality**: Ask the user to input a number and write a loop to test whether the number is prime. You will need to use the modulo-operator `x%y` that calculates the remainder of the integer division `x/y`. If the remainder is zero, `x` is divisible by `y`. primality.py

2.3 **Prime suspects**: Write a program that lists the prime numbers from 1 to 100. prime_suspects.py

2.4 **Guessing game**: Your program should choose a random number with `random.randint(0,100)` and then have you guess this number while providing feedback on whether your guess is too small or large. guessing_game.py

2.5 **Reversed roles**: Play the guessing game with reversed positions: You are choosing the number and providing feedback; the program must guess. reversed_roles.py

# Chapter 3

# Collections

Python programs make extensive use of data collections or *iterables*, notably `list`, `tuple` and `dict`. We have already (unknowingly) encountered lists in the section on loops: The `range()` function actually generates a `list` over which the `for` loop iterates:

```
>>> print range(1,6)
[1, 2, 3, 4, 5]
>>> for x in range(1,6):
...     print str(x)+" sheep"
...
1 sheep
2 sheep
3 sheep
4 sheep
5 sheep
>>>
```

The range function has three arguments: start=0, stop and step=1:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(2,7)
[2, 3, 4, 5, 6]
>>> range(0,10,2)
[0, 2, 4, 6, 8]
>>>
```

It provides a shorthand to use `for x in [iterable]` as counting loops, even though the construct really works with any iterable object.

## 3.1 list

Python has a built-in `list` type, which is written as comma separated values within square brackets `[...]`. Lists are similar to arrays in other programming languages in that their data can be accessed with (zero-indexed) indices in square brackets:

```
>>> colors = ['red', 'blue', 'green']
>>> print colors
['red', 'blue', 'green']
>>> type(colors)
<type 'list'>
>>> print colors[0]
red
>>> print colors[2]
green
>>> print len(colors)
3
>>>
```

### List iteration

As we have seen in chapter 2, the counting loop in python iterates over lists generated by the `range()` function. The same construct can also be employed to iterate any collection: `for var in list` looks at each element in the list:

```
>>> numbers = [12, 16, 10, 4]
>>> total = 0
>>> for var in numbers:
...     total += var
>>> print total
42
>>>
```

**Caution**: Do not add elements to or remove them from the list while you are iterating it. It is not clearly defined what happends if you manipulate the part you are currently iterating.

Python provides a convenient way to test if an element appears in a collection using the syntax `value in list` (takes linear time in the list size):

```
>>> fruits = ['apple','banana','cherry']
>>> 'apple' in fruits
True
>>> 'strawberry' in fruits
False
```

### Iteration tricks

In case you need to iterate two lists at the same time, you can use the `zip` function to combine them into one list which has two elements at each position:

**Python source file** code/collections/zip.py

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

names = ['arthur', 'ford', 'zaphod']
numbers = ['555-1132', '555-0174', '555-3142']
for name, number in zip(names,numbers):
    print "call",name,"at",number

# prints:
# call arthur at 555-1132
# call ford at 555-0174
# call zaphod at 555-3142
```

Similarly, if you need to know both the position and value of each element in the list, the function `enumerate` is your friend:

**Python source file** code/collections/enumerate.py

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

fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
    print fruit,"is at position",index

# prints
# apple is at position 0
# banana is at position 1
# cherry is at position 2
```

### List methods

The `list` type provides a lot of functionality out of the box. After defining them with square brackets, they can be appended to using the `+` operator or repeated several times by multiplying them with a number.

```
>>> L = [1,4,2,3]
>>> print L+L
[1, 4, 2, 3, 1, 4, 2, 3]
>>> print L*3
[1, 4, 2, 3, 1, 4, 2, 3, 1, 4, 2, 3]
>>>
```

Besides these binary operators, a list also provides several member functions for manipulation. For instance:

**operations on a list**

```
>>> L = [1,4,2,3]
>>> L.reverse()
>>> print L
[3, 2, 4, 1]
>>> L.sort()
>>> print L
[1, 2, 3, 4]
>>>
```

Some of the more frequently used member functions include the following:

```
list.append(elem) # adds a single element to the end of the list.
list.count(elem)  # total number of occurrences of elem in list
list.insert(index, elem) # inserts the element at the given index
list.index(elem)  # find the index of the given elem
list.pop(index)   # remove and return the element at the given index
list.remove(elem) # find & remove the first occurrence of the elem
list.reverse()    # reverse the order of all elements in list
```

**Caution**: The above methods *do not* return the modified list, they just modify the original list.

In addition, these standalone functions can be applied to any list to determine its length, as well as the smallest, largest and total value:

```
len(L)  # length of L
min(L)  # smallest item of L
max(L)  # largest item of L
sum(L)  # total value of items in L
```

Note that these operations (except for `len`) take time linear in the size of the list because it is *not a structured container*. The contents of the list are stored exactly in the order given and a function looking for the largest element needs to look at all of them to find it.

### Slicing

Another useful functionality of lists is *slicing*. We have already seen that the square brackets of a list can be used to access individual elements in the list by position. By providing a slicing instruction, `start:stop:step`, the same brackets can be used to select a sublist rather than just a single element.

The following snippet illustrates several slicing variants:

**list slicing**

```
>>> L = range(10)
>>> # from the 5th element to the end
>>> L[5:]
[5, 6, 7, 8, 9]
>>> # from start up to, but not including, the 4th element
>>> L[:4]
[0, 1, 2, 3]
>>> # from the 3rd last up to, not including, the last
>>> L[-3:-1]
[7, 8]
>>> L[::2]
[0, 2, 4, 6, 8]
>>>
```

The subsets addressed via slicing can not only be read but also altered and removed using the assignment and `del` statements, respectively:

**list slicing**

```
>>> L = range(10)
>>> L[3:-3] = [42]
>>> print L
[0, 1, 2, 42, 7, 8, 9]
>>> del L[1::2]
>>> print L
[0, 2, 7, 9]
>>>
```

The assignment in the second line replaces the entire sublist with a smaller list containing only the number 42. As a result, the numbers `4, 5, 6` are missing in line 4 – replaced by the slicing/assignment operation. The next line uses a slicing/del statement to remove the subsequence starting at 1 with a step of 2. As a result, all elements which had an odd index – `1, 42, 8` – are gone.

### Shallow vs. deep copies

By default, python creates references for mutable objects (such as lists) in assignments or when calling a function (for efficiency reasons). This means that mutating the copy has an effect on the original and vice-versa.

**Note**: Assigning a new value to a reference *is not a mutation*. It changes the variable to reference of different object while leaving the previously referenced object unchanged! A mutation is changing a property or elements contained within the referenced object.

Typical copy attempts are `copy = list(original)` or `copy = original[:]` (the latter uses slicing to get a view of the entire original list). Both of them seem to work at first, but they only make *shallow copies*. For a list, a shallow copy means that the created list itself is a new object but the elements it contains are references to the elements in the original list. This is irrelevant when the elements are immutable (such as integers or strings), but makes a difference when they can be changed (i.e., lists nested in other lists):

**Python source file** code/collections/shallow.py

```
1  #!/usr/bin/env python
2
3  nested = ["hello", [1, 2, 3]]
4  reference = nested
5  copy = nested[:]
6
7  nested[0] = "world";
8  nested[1][1] = 42;
9
10 print nested     # prints ['world', [1, 42, 3]]
11 print reference  # prints ['world', [1, 42, 3]]
12 print copy       # prints ['hello', [1, 42, 3]]
```

This indicates that an attempt to copy a list using slicing merely fixes the issue at the top level of a potentially deeply nested collection. If you need a *deep copy* (i.e., one that is completely independent of the original), you can use the `deepcopy` function from the `copy` module. It recursively copies the elements of (nested) collections, rather than using references:

**Python source file** code/collections/deep.py

```
1  #!/usr/bin/env python
2  import copy
3
4  nested = ["hello", [1, 2, 3]]
5  deepcopy = copy.deepcopy(nested)
6
7  nested[0] = "world";
8  nested[1][1] = 42;
9
10 print nested     # prints ['world', [1, 42, 3]]
11 print deepcopy   # prints ['hello', [1, 2, 3]]
```

As a result, changing the original object (lines 7-8) has no effect on the independent deep copy we have created in line 5.

### List comprehension*

List comprehension is a short-hand notation for a loop that generates a new, altered sequence from an existing one. The notation uses square brackets, but rather than explicitly listing the elements, we write a recipe how to generate each element from the items found in the first list:

**list comprehension**

```
>>> L = range(5)
>>> [ "Sheep "+str(x) for x in L ]
['Sheep 0', 'Sheep 1', 'Sheep 2', 'Sheep 3', 'Sheep 4']
>>> [ x*x for x in range(20) if x%2==0 ]
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
>>>
```

The second line illustrates the simple use case: The newly generated list should contain *expression* for every *element* found in the list. Here we take each number in `L` and we create a string of the form `"Sheep " + str(x)` for the new list. Similarly, in the fourth line we are generating a new list that will contain the square of each original element. However, in this case the list comprehension also includes a condition for the element to be included in the new list (only even ones). As a result, we get a list of squares of even numbers.

## 3.2 `tuple`

A `tuple` is written with parentheses `(...)` and it is essentially the same as a `list`, except it cannot be altered once created (immutable). This can be seen by comparing the methods these two types of collections allow:

```
>>> [ m for m in dir(list) if m not in dir(tuple) ]
['__delitem__', '__delslice__', '__iadd__', '__imul__', '__reversed__',
     ↪ '__setitem__', '__setslice__', 'append', 'extend', 'insert', 'pop',
     ↪ 'remove', 'reverse', 'sort']
>>>
```

This nifty list comprehension shows all the methods a `list` has but a `tuple` does not. As you can see, this includes all the methods that add, remove or change items in the collection. This subtle differentiation of mutability is necessary because in certain places only immutable objects can be used, namely as a key in a dictionary (see section 3.3).

Because tuples are immutable, the attempt to assign a new value to individual items or appending to a tuple raises an error:

```
>>> T = (1, 2)
>>> T.append(3)
Traceback (most recent call last):
  ...
AttributeError: 'tuple' object has no attribute 'append'
>>>
```

Apart from that, they can be treated the same way as lists. Both iteration and slicing is possible with tuples, as long as the operation does not change their value. If necessary, it is also possible to convert lists into tuples and vice-versa, using casting:

```
>>> myList = range(10)
>>> myTuple = tuple(myList)
>>> print myTuple
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>>
```

Note the parentheses (rather than square brackets) surrounding the output, indicating that it is indeed a `tuple` that is being printed.

**Caution**: Where it is not ambiguous, the parentheses enclosing the tuple may be dropped. A special case is a tuple of length 1: Because the brackets in `(1)` could be interpreted as "grouping", it is necessary to add a trailing comma in this case: `(1,)`:

```
>>> type( () )
<type 'tuple'>
>>> type( (42) )
<type 'int'>
>>> type( (42,) )
<type 'tuple'>
>>>
```

While both `()` and `(42,)` can unambiguously be identified as tuples by the interpreter, the case of `(42)` is less clear. Because parentheses serve the double purpose of grouping elements in expressions, this is interpreted as an integer number grouped by itself, rather than as a tuple with a single integer element. Appending the trailing comma fixes this issue.

### Packing / unpacking in assignments

Another very convenient application of tuples is for multiple assignments via their automatic packing behavior:

```
>>> T = 1, 2
>>> type(T)
<type 'tuple'>
>>> x, y = T
>>> print x
1
>>> print y
2
>>> x, y = y, x
>>> print x,y
2 1
>>>
```

The first line shows tuple packing: Even though there are no parentheses on the right hand side, the values are automatically "packed" into a tuple that is stored in T (i.e., the parentheses are optional where the syntax is unambiguous without them).

The next lines illustrates tuple "unpacking", where the left-hand side consists of a tuple of multiple variables `x, y`. The values on the right-hand side are assigned to the variables in the order they appear. Note that they can be given implicitly (as a tuple valued variable or function) or explicitly (written as a tuple). In particular, this construct also allows for a convenient way of swapping two variables: `x, y = y, x`.

## 3.3 dict

The other workhorse datatype in python is the dictionary, which implements a hash table. It maps unique immutable keys to values with an efficient lookup mechanism. For better understanding, python dictionaries are best thought of as what their real-world counter parts are: language dictionaries or telephone books. They serve the purpose of looking up a *key* (the name of a person or an English word) and finding its translation or corresponding *value*. Just as in the phone book analogy, searching for a `key` (i.e., name) is fast in a dictionary, but finding a value (i.e., phone number) is difficult because the collection is organized by the former.

Consider the following example of a phone book implemented as a dictionary:

```
>>> phonebook = {
...     'Arthur': '555 12 34',
...     'Trillian': '555 56 78',
...     'Ford': '555 99 99',
... }
>>> print phonebook['Ford']
555 99 99
>>> 'Zaphod' in phonebook
False
>>> phonebook['Zaphod'] = '555 55 55'
>>> print phonebook
{'Arthur': '555 12 34', 'Zaphod': '555 55 55', 'Trillian': '555 56 78', 'Ford':
    ↪ '555 99 99'}
>>>
```

When printed, a dictionary can be recognized by the surrounding curly braces. Its contents are written as a set of `key:value`, separated by commas. Value access follows the same syntax as for lists and tuples (i.e., using square brackets), but the key is used as the index rather than a number representing the position. In fact, dictionaries do not remember position or ordering at all: Zaphod, who was added to the phonebook last, shows up somewhere in the middle.

### Valid key types

In the phone book example we used strings as keys, but dictionaries can use any *immutable* object such as strings, numbers or tuples thereof. You can even use different key types in the same dictionary – the only requirement is that the keys must be unique.

### Non-existing keys

Trying to access a key which is not in the dictionary throws a `KeyError`. You can either catch that error or use the keyword "`in`" to check if a particular key is present: `key in dict`. The member method `yourdict.get(key)` returns the corresponding value (if the key is present) or `None` if the key is not found.

### Iterating a dictionary

A for loop on a dictionary iterates over its keys by default. The keys will appear in an arbitrary order (i.e., python provides no guarantee that they will be sorted in any particular way). The method `dict.values()` can be used to

iterate over the set of values instead. There is also a member function `items()` which provides a list of `(key, value)` `tuple`s for combined iteration:

```
>>> print phonebook.keys()
['Arthur', 'Zaphod', 'Trillian', 'Ford']
>>> print phonebook.values()
['555 12 34', '555 55 55', '555 56 78', '555 99 99']
>>> print phonebook.items()
[('Arthur', '555 12 34'), ('Zaphod', '555 55 55'), ('Trillian', '555 56 78'),
    ↪ ('Ford', '555 99 99')]
>>> for key, value in sorted(phonebook.items()):
...     print key, "has the number", value
...
Arthur has the number 555 12 34
Ford has the number 555 99 99
Trillian has the number 555 56 78
Zaphod has the number 555 55 55
>>>
```

The first three print statements indicate how to access keys, values or key-value pairs (items) as a list. The for-loop illustrates several tricks when iterating a dictionary. By using `phonebook.items()`, we have direct access to both the key and the value inside the loop. And because we send this list of items through the `sorted()` function, the resulting output is actually sorted alphabetically by key.

**Note**: When iterating the items in python2.x, you should use the item generator `iteritems()` in place of `items()`. It generates items on the fly rather than precomputing the full list of items in memory.

## 3.4 Exercises

3.1 **Sharity**: Given two lists of numbers, write a list comprehension that only contains the numbers which are in both lists. sharity.py

3.2 **Only you**: Given a list of names, write a list comprehension that copies only names with the letter 'u' to the new list. only_you.py

3.3 **Zippity zip**: Use the `zip()` function to glue two lists of equal length together to form a dictionary. zippity_zip.py

3.4 **Up is down***: Write a program that takes a dictionary and builds the corresponding dictionary for backwards translation. What are the requirements for the keys and values for this to work?

up_is_down.py

3.5 **Many of a kind***: A list can contain different datatypes (strings, numbers, etc.). Test how the sorting function deals with different types.

many_of_a_kind.py

# Chapter 4

# Functions

Functions in python are created with the keyword **def**, followed by the function name and its argument list (enclosed in parentheses). As with the blocks in chapter 2, the statements that form the function body are grouped with indentation. The value of the function is indicated with the **return** statement.

**Python source file** code/functions/basic.py

```
1 def square(x):
2     result = x * x
3     return result
4
5 print square(8) # 64
```

Functions in python are only run when you "call" them, i.e. by adding parentheses containing the necessary parameters. It is also possible to refer to a function by name without calling it: This is used to assign the same function to a different variable or to pass the function itself as an argument:

**Python source file** code/functions/byname.py

```
1  # a simple function to multiply with 2
2  def double(number):
3      return number *2
4
5  # store a copy of the function in 'repeat', then use
6  timestwo = double
7  print timestwo(21) # prints 42
8
9  # map applies a function to items in an interable:
10 L = map(double,range(5))
11 print L # prints [0, 2, 4, 6, 8]
```

## 4.1 Arguments

Python offers multiple ways to pass arguments to a function. For the simplest case, so called "positional arguments", the values passed to the function are stored in the argument variables according to their position. This is how most programming languages handle function arguments:

**Python source file** code/functions/positionalargs.py

```
1 def test(a,b=0,c=0):
2     print a,b,c
3
4 test(1,2,3) # 1 2 3
5 test(3,2,1) # 3 2 1
6 test('4',2) # 4 2 0
7 test('hello') # hello 0 0
```

Note that the values passed are assigned to the arguments in the order each is noted. This form of argument-value association is primarily useful for small helper functions where the number of arguments is small and fixed and the order is clear or irrelevant. A good example is a multiply function, where both arguments are of the same type and the result does not depend on their order. In addition, python allows for default values to be indicated. These are used in case the user passes less values to the function than it has arguments.

**Caution**: Because positional arguments are assigned according to their order, you can only assign default values for some or all arguments starting from the back of the list.

### Keyword arguments

To enhance readability and avoid confusing the argument order, python also allows the user to specify arguments by *keyword* (i.e. by indicating the name of each argument). When identified by their keyword, the arguments can be given in any order and python will associate them correctly:

**Python source file** code/functions/keywordargs.py

```
1 def test(a,b=0,c=0):
2     print a,b,c
3
4 test(a=1, b=2, c=3) # 1 2 3
5 test(c=3, b=2, a=1) # 1 2 3
6 test(b=2, a='4')    # 4 2 0
7 test(a='hello')     # hello 0 0
```

Keyword arguments are often used in situations where a function allows for several optional arguments, out of which only few are used concurrently:

**Python source file** code/functions/optionalargs.py

```
1  def myprint(what, before="", after="", repeat=1, yell=False):
2      text = before + str(what) + after
3      if yell:
4          text = text.upper()
5      print repeat * text
6
7  myprint('hello') # hello
8  myprint('hello', after='!!!') # hello!!!
9  myprint('hello', before='[[ ', after=' ]]') # [[ hello ]]
10 myprint('hello', yell=True ) # HELLO
11 myprint('hello', repeat=3) # hellohellohello
```

Depending on which arguments are provided, the `myprint` function in the previous code snippet will deviate from the default behavior in a different way: By printing a prefix, suffix, all uppercase or repeated message. Those parameters which are not provided as keyword arguments in the function call take their default values instead.

**Note**: If you know the argument names, it is recommended to use keyword arguments to improve readability and avoid mistakes in the argument ordering. This also makes your code less vulnerable to (small) interface changes.

**Caution**: When mixing positional and keyword arguments in a function call, you *must* note all the positional arguments first to avoid ambiguity. Positional arguments are then assigned *first*, followed by keyword arguments:

```
>>> def power(base=2, exp=10):
...     return base**exp
...
>>> power(exp=5,2)
    Traceback (most recent call last): ...
    SyntaxError: non-keyword arg after keyword arg
>>> power(base=2,5)
    Traceback (most recent call last): ...
    SyntaxError: non-keyword arg after keyword arg
>>>
```

### Star Arguments

Similar to the automatic packing and unpacking of `tuple`s seen in chapter 3, there is also a way to pass the items in a `list` or `tuple` to a function as positional arguments. In this case, a preceding star has to indicate the desired *unpacking* (otherwise you would be passing the entire collection as the first argument).

**Python source file** code/functions/starlistargs.py

```
1  def test(a,b=0,c=0):
2      print a,b,c
3
4  args = [1,2,3]
5  test(*args) # 1,2,3
6
7  args = ['4',2]
8  test(*args) # 4 2 0
9
10 args = ['hello']
11 test(*args) # hello 0 0
```

**Caution**: Just as with a regular argument count mismatch, this will throw an error whenever the number of items in the `list` is not appropriate for the function. Use conditions or slicing to avoid this.

Conversely, it is also possible to write a function that accepts any number of arguments by storing them in a list. In this case, we need to request the desired *packing* of positional arguments with a star preceding the parameter:

**Python source file** code/functions/variadic.py

```
1  def product(*numbers):
2      result = 1
3      for x in numbers:
4          result *= x
5      return result
6
7  print product(1,2,3) # 6
8  print product(2,3,5,7,11,13) # 30030
9  print product(*range(1,42))
10 # 3345252661316380710817006205344075166515200000000
```

After packing, the parameter `numbers` in the previous example will be a tuple of all the *positional* arguments passed to the product function. Possible keyword arguments would not be included in this parameter.

### Unpacking dictionaries

Similarly, it is also possible to unpack a dictionary with string keys as the keyword arguments in a function call. This dictionary unpacking uses two stars, **, as the prefix for unpacking:

**Python source file** code/functions/starkeywordargs.py

```
def myprint(what, before="", after="", repeat=1, yell=False):
    text = before + str(what) + after
    if yell:
        text = text.upper()
    print repeat * text

kwargs = {
    what = 'hello',
    after = '!!!',
}
myprint(**kwargs) # hello!!!

kwargs['before'] = '[['
kwargs['after'] = ']]'
myprint(**kwargs) # [[ hello ]]

kwargs['yell'] = True
myprint(**kwargs) # [[ HELLO ]]
```

Likewise, a function which should accept any number of keyword arguments can be written by preceding one parameter with two stars – after packing of the keyword arguments, this parameter will be a dictionary with string keys and argument values (but it will not contain any positional arguments):

**Python source file** code/functions/variadic_keywords.py

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

def print_kw(**keywords):
    for key, value in keywords.iteritems():
        print "received",key,"=>",value

print_kw(test=1,foo='bar')
# received test => 1
# received foo => bar

print_kw(1,test='hello')
#Traceback (most recent call last):
#  File "code/functions/variadic_keywords.py", line 9, in <module>
#    print_kw(1,test='hello')
#TypeError: print_kw() takes exactly 0 arguments (2 given)
```

## 4.2 Docstrings

In chapter 1, we have seen that introspection allows us to investigate built in functions and objects. We can provide the same kind of information for our own functions by enhancing them with so-called *docstrings*. That is simply a string constant at the very beginning of the function, enclosed in triple quotes, `"""`. This string does not serve a programmatical purpose other than to explain what the function does when introspection is used:

**Python source file** code/functions/docstrings.py

```
1  def factorial(n):
2      """Return the factorial of n, an exact integer >= 0.
3
4      The function calculates the factorial n!,
5      where the parameter n is rounded to the next
6      lower or equal integer value.
7
8      """
9      result = 1
10     factor = 2
11     while factor <= n:
12         result *= factor
13         factor += 1
14     return result
15
16 help(factorial)
```

In addition to presenting a list of parameters, the call to `help(factorial)` uses the docstring to provide usage information for the `factorial` function.

## 4.3 `lambda` functions*

Python provides an inline way to declare short, unnamed functions using the following `lambda` function syntax:

```
>>> is_odd = lambda x: x%2==1
>>> is_odd(13)
True
>>>
```

In the above example, the `lambda` function is immediately assigned to a variable, thus giving it a name (we could have written `def is_odd: return x%2==1` just as well). Typically, lambda functions are created on-the-fly to be passed as an argument to another method – for instance as a sorting predicate in a sorting function.

I.e., to sort words by their length rather than alphabetically, we could use:

```
>>> words = ['Time','is','an','illusion','Lunchtime','doubly','so']
>>> sorted(words, key=lambda x: len(x))
['is', 'an', 'so', 'Time', 'doubly', 'illusion', 'Lunchtime']
>>>
```

## 4.4 Generator functions*

Generator functions maintain their state (and current location of the virtual execution pointer) between calls. By using `yield` instead of `return`, the function will resume at that point when called the next time:

**Python source file** code/functions/generator.py

```
1  def fibonacci_gen():
2      x,y = 1,1
3      while True:
4          yield x
5          x,y = y,x+y
6
7  fib = fibonacci_gen()
8
9  print [ fib.next() for x in range(20) ]
10 # [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
11 #  610, 987, 1597, 2584, 4181, 6765]
```

The advantage of generators is that they only calculate the next value of the sequence when needed, rather than precalculating and storing all the values of the sequence in memory.

## 4.5 Decorators*

Functions are objects, therefore they can be passed to a function and also be the return value of a function. A decorator is a wrapper that takes a function and returns an enhanced function that implements additional functionality.

```
def offbyone(original_func):  # decorated: takes original function
    def new_func(x,y):
        return original_func(x, y) + 1
    return new_func  # returns enhanced function

@offbyone  # syntax for decorating: passes add through offbyone
def add(x, y):
    return x + y

print add(1,1) # prints 3 (output from the enhanced function)
```

Consider also the following culinary example which illustrates how decorators enhance the original function by adding additional output:

**Python source file** code/functions/decorators.py

```
1  def bread(func):
2      def enhanced():
3          print "</'''''''''\>"
4          func()
5          print "<\_________/>"
6      return enhanced
7
8  def ingredients(func):
9      def enhanced():
10         print " @tomato@"
11         func()
12         print " ~lettuce~"
13     return enhanced
14
15 def bacon(food=" --bacon--"):
16     print food
17
18 bacon()
19 # --bacon--
20
21 wrapped_bacon = bread(ingredients(bacon))
22 wrapped_bacon()
23 #</'''''''''\>
24 # @tomato@
25 # --bacon--
26 # ~lettuce~
27 #<\_______/>
28
29 # same as above but with decorator syntax
30 @bread
31 @ingredients
32 def sandwich(food=" --bacon--"):
33     print food
34
35 sandwich()
36 #</'''''''''\>
37 # @tomato@
38 # --bacon--
39 # ~lettuce~
40 #<\_______/>
```

The example on line 21 shows how to manually use decorators on existing functions to create an enhanced version of them. The equivalent example on lines 30-32 does the same using the decorator syntax using `@decorator`.

## 4.6 Exercises

4.1 **All the things!**: Write a function that has positional and keyword arguments, some default values and takes an arbitrary number of each. all_the_things.py

4.2 **Mean machine**: Write a function that takes any number of numerical arguments and calculates their average. mean_machine.py

4.3 **Primal instinct**: Wrap your primality test from chapter 2 in a function for easier use, then use it in a list comprehension to generate a prime list. primal_instinct.py

4.4 **Decorative yelling***: Write a decorator that announces every function call by printing to the screen. decorative_yelling.py

4.5 **Generous***: Write a generator function that produces a sequence such as the Fibonacci series (your choice of sequence). generous.py

# Chapter 5

# Strings, input and output

Strings are either enclosed in single or double quotes in python. In either case, you can use the escape character '\' to denote special characters such as line breaks '`\n`', tabs '`\t`' or quotes '`\'`' within strings:

```
>>> 'Hello I\'m a string'
"Hello I'm a string"
>>> "and I am a \"string\" too"
'and I am a "string" too'
>>>
```

**Note**: `print` tries to be smart about how to represent strings such that it needs to use less escape characters by switching from single to double quotes and vice-versa.

For situations where you expect to use many characters that would need escaping, python understands so called *raw strings* which are denoted with a leading '`r`' character before the opening quote. In raw strings, the escape character has no special meaning (except for escaping a quote at the very end). Thus, the string is used exactly as given, i.e. *raw*. This form is frequently used in regular expressions, see section 6.5:

```
>>> print 'I am a s\t\ri\ng with lots of \\\\ '
i am a s
g with lots of \\
>>> print r'I am a s\t\ri\ng with lots of \\\\ '
I am a s\t\ri\ng with lots of \\\\
>>>
```

### Docstrings

Python uses triple quotes to denote multiline strings. Because they are enclosed in triple quotes (three double quotes), individual quotes – single or double – need not be escaped within. They are used as documentation strings (as seen in chapters 4 and **??**) and whenever a long string needs to be included verbatim.

## 5.1 Encoding

To use unicode characters, you need to declare the encoding in the header:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
...
```

Python 2.x knows two different types of strings – standard `str` objects which can only handle ascii characters and Unicode strings that can handle unicode. The latter is indicated by preceding the string literal with the character 'u'. Within these unicode strings, you can directly type non-ascii characters if you have set up the file encoding as described above. Alternatively, you can escape a unicode character with \u2026 where the number is its unicode value.

## 5.2 String formatting

There are multiple methods for string formatting available. The one I find most convenient and usable has the following syntax:

```
>>> '{} is not {}'.format(13,42)
'13 is not 42'
>>> '{1} is not {0}'.format(13,42)
'42 is not 13'
>>> '{first} is not {second}'.format(first=13,second=42)
'13 is not 42'
>>> '{:.3f} is not {:+d}'.format(13,42)
'13.000 is not +42'
>>>
```

Slots where a variable should be printed are indicated in the string with braces {...}. The default behavior is to assign the values to the slots in the order they are given. It is however possible to indicate which positional argument is to be used with a number in each slot. In particular, this also allows reusing the same variable multiple times if necessary. Yet another option is to use keyword arguments and denote the slots with the corresponding keyword.

The last line illustrates some of the additional formatting options:

```
# :d   -- decimal
# :x   -- hexadecimal
# :o   -- octal
# :b   -- binary
# :f   -- float
# :.3f -- float with 3 digits
# :+d  -- decimal with sign always shown
# :>12 -- right aligned in 12 character slot
# :^12 -- center aligned in 12 character slot
# :<12 -- left aligned in 12 character slot
```

## 5.3 File input/output

Reading and writing files is a frequent chore in python, a simple (though not necessarily the fastest) way to achieve this is:

```
>>> # open a file for reading:
>>> fp = open('data.txt', 'r')
>>> for line in fp:
...     print line
...
<contents of the file>
>>>
```

Using the `open()` function with '`r`' (read option) returns a file object that can be iterated over. Each iteration corresponds to one line of the file which can then be handled before stepping to the next line. Similarly, you can also use the `open()` function to get a file object that is writable:

```
>>> # open a file for writing:
>>> fp = open('data.txt', 'w')
>>> fp.write("HELLO\n")
>>> fp.close()
>>>
```

The write method simply writes the exact string it is given to the file. In particular, it does not append a newline character. It is important to close the file to make sure all buffered output is written to the file system. If you need to read files with unicode characters in them, you can use the codecs module, which provides support for reading a Unicode file.

```
>>> import codecs
>>> fp = codecs.open('data.txt', 'rU', 'utf-8')
```

## 5.4 String Methods

The following list shows some of the more frequently used string methods. You can use introspection to get the full list:

```
s.lower(), s.upper() # returns s in a lowercase or uppercase version
s.strip()            # returns s with whitespace removed from start/end
s.find('needle')     # searches for the string needle (returns index)
s.replace('aa','bb') # returns a s with all 'aa' replaced by 'bb'
s.split('delim')     # returns the list of substrings when cutting at delim
s.split()            # default behavior splits on all whitespace chars
'delim'.join(list)   # concatenates list items using delim as glue
pattern.format(...)  # uses string pattern for formatting
```

### Slicing

Of course python strings also allow for individual character access and slicing as we have seen it with lists. The only minor annoyance is that strings are immutable objects and therefore write-protected. You cannot assign characters or substrings with slicing:

```
>>> text = "So long and thanks for all the fish"
>>> text[-4:] = "fun"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

### Bytearray

Like `tuple` and `list`, python knows both an immutable and a mutable text type. The reason is the same as before: Because they are immutable, strings can be used as keys in a dictionary. Our options are to create a new string with slicing or to use the mutable counterpart to string, the bytearray:

```
>>> text = "So long and thanks for all the fish"
>>> text[:-4] + "fun"
'So long and thanks for all the fun'
>>>
>>> mutable = bytearray(text)
>>> mutable[-4:] = "fun"
>>> mutable
bytearray(b'So long and thanks for all the fun')
>>>
```

## 5.5 Exercises

5.1 **Palindrome**: A palindrome is a word that reads the same in either direction. Write a palindrome test using slicing operations.

palindrome.py

5.2 **Full Stop**: Write. a. function. that. makes. the. reader. take. a. mental. pause. after. each. word. she. reads.

full_stop.py

5.3 **Comma, comma, comma**: Read lines from a csv file (comma separated values) and split each line into a list of numbers.

comma_comma_comma.py

5.4 **Number formatting**: Write a program that reads a number in base 10 (from user input) and then prints it in base 16, 10, 8 and 2 with formatting options (the relevant formats are x,d,o,b).

number_formatting.py

5.5 **All-in-one**: Use `str.format()` in conjunction with keyword arguments and dictionary unpacking (`**yourdict` as argument).

all_in_one.py

# Chapter 6

# Built-in Modules

Python embraces the "batteries included" philosophy. This is best seen in the sophisticated, wide-ranging and robust capabilities of its built-in modules. The following pages contain some information to get you started with the ones I frequently use. You can find a more complete list and their detailed documentation online: `http://docs.python.org/2/tutorial/modules.html`

## 6.1 Date and time

The `datetime` module supplies classes for manipulating dates and times:

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2013, 4, 17)
>>> now.strftime("%d. %m. %Y is a %A.")
'17. 04. 2013 is a Wednesday.'
>>> birthday = date(1952, 3, 11)
>>> age = date.today() - birthday
>>> age.days
22317
>>>
```

One of the most important function is `date.strftime` which formats a `datetime` object according to a given format string. In the above example the format displays the date and current day of the week. The full list of modifiers is available online: `http://docs.python.org/2/library/time.html`

The reverse is also possible: the `time` module has a string parsing function where you can specify the expected format of the string:

```
>>> import time
>>> datestr = "Wednesday, 17.04.2013"
>>> when = time.strptime(datestr, "%A, %d.%m.%Y")
>>> print when
time.struct_time(tm_year=2013, tm_mon=4, tm_mday=17, tm_hour=0, tm_min=0,
    ↪ tm_sec=0, tm_wday=2, tm_yday=107, tm_isdst=-1)
```

If you need more advanced date/time parsing, you should give the (external) `dateutil` package a try.

## 6.2 Mathematical functions

The math module gives access to the underlying C library functions for floating point math. You can use `dir()` to get an impression of all the functionality the module provides. Let's try some of these:

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh', 'asin',
    ↪ 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
    ↪ 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
    ↪ 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan',
    ↪ 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow',
    ↪ 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> math.cos(math.pi/4)
0.7071067811865476
>>> math.log(1024, 2)
10.0
>>>
```

In case you need to use some of these functions frequently, consider importing them into the main namespace using a direct import statement. For instance, `from math import atan2` would import `atan2` for use without the `math.` prefix.

**Note**: For large scale numerical calculations, `numpy` and `scipy` provide more efficient array classes and mathematical functions that can be applied to all elements in an array at once (see chapter 7).

## 6.3 Random Numbers

The random module provides a good pseudo random number generator (to be precise, a Mersenne twister) that can be used for random choices, samples, uniform floats and random integers from a range:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'pear'
>>> random.sample(range(100), 10)
[2, 27, 22, 73, 67, 89, 8, 42, 21, 50]
>>> random.random()
0.026535969683863625
>>> random.randrange(6)
1
>>>
```

For reproducible results, use the `random.seed()` function with a seed number.

## 6.4 OS interaction

The following modules are useful to gather information about the host operating system and to interact with the file system: The `platform` module provides some insights into the type of the host machine.

```
>>> import platform
>>> platform.python_version
2.7.3
>>> platform.platform
Darwin-11.4.2-i386-64bit
>>>
```

The `os` module provides functions for (file) system interaction, including:

```
os.getcwd()
os.chdir(...)
os.chmod(path,mode)
os.chown(path, uid, gid)
os.rename(src, dst)
os.remove(path)
os.rmdir(path)
# use walk to traverse the file system:
os.walk(top, topdown=True, onerror=None, followlinks=False)
```

**Caution**: Be sure to use `import os` rather than `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates differently.

### shutil

For simple file and directory management, the shutil module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.txt', 'backup/data.txt')
>>> shutil.move('log.txt', 'oldlogs')
>>>
```

### glob

The glob function from the `glob` module lets you use wildcards to list files in the current directory as you would in the shell. It returns a list of the files matching the description:

```
>>> import glob.glob
>>> glob.glob('*.*')
['primer.pdf', 'code', 'chapters', ...]
>>>
```

## 6.5 Regular Expressions*

Regular expressions are used to describe patterns for complex matching and manipulations of strings. Python provides its interpretation in the `re` module:

```
>>> import re
>>> text = 'So long and thanks for all the fish'
>>> re.match(r'\bf[a-z]*', text) # match checks only beginning
>>> re.search(r'\bf[a-z]*', text) # search checks entire string
<_sre.SRE_Match object at ...>
>>> re.findall(r'\bf[a-z]*', text) # all words with f
['for', 'fish']
>>> re.sub(r'\bf[a-z]*', r'fun', text)
'So long and thanks fun all the fun'
>>>
```

The pattern describes a word boundary '`\b`', followed by the letter '`f`' and any number of characters `[a-z]`. The first query (line 3) tries to match this pattern at the beginning of the string, which fails (no output). The second query (line 4) searches for the pattern anywhere in the string and successfully finds one of the two words starting with an `f`. The result of the third query (line 6) is a list of all words that start with the letter '`f`'. And the last query replaces all the occurrences of the pattern with the word "fun".

**Note**: When only simple capabilities are needed, string methods (such as replace) should be preferred because they are easier to read and debug (and also more efficient).

See `http://docs.python.org/2/howto/regex.html` for more information.

## 6.6 Argument parsing (argparse)*

The `argparse` module provides the functionality to implement a user-friendly command-line tool which accepts several long and/or short options as well as (positional) arguments. The configuration of the argument parser defines which are valid options and takes care of interpreting the contents of `sys.argv` accordingly. As a nice feature, it also automatically generates a help text of all the options the program understands.

**Python source file** `code/modules/arg_parsing.py`

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

import argparse
parser = argparse.ArgumentParser()

parser.add_argument('pos', nargs='+', help='positional arguments (will be
    ↪ printed)')
parser.add_argument('-v', '--verbose', help='more output', action='store_true')
args = parser.parse_args()

if args.verbose:
    print 'The positional arguments are:',args.pos
else:
    print args.pos
```

The example illustrates how to parse positional arguments and optional flags such as a verbosity setting. The help printed by the example script would look as follows:

```
usage: arg_parsing.py [-h] [-v] pos [pos ...]

positional arguments:
  pos            positional arguments (will be printed)

  optional arguments:
    -h, --help     show this help message and exit
    -v, --verbose  more output
```

More details: `https://docs.python.org/dev/library/argparse.html`

## 6.7 Test driven development (unittest)*

TDD builds on the concept of writing down the requirements for your program first, then writing and testing your code against them. The `unittest` module provides the necessary tools to accomplish this in five steps:

1.) Import the `unittest` built-in module.

2.) Create your won class which subclasses the `TestCase` class from the `unittest` module.

3.) (optional) Write member function `setUp()` and/or `tearDown()`, which will be called before/after all test cases.

4.) For each test case, write a function whose name starts with `test_` and which specifies the expected behavior using class methods such as `self.assertTrue`, `self.assertEqual` and `self.assertRaises`

5.) Run `unittest.main()` to see which tests fail and which ones succeed.

The following example shows a class with two unit tests for the built-in sequence functions `sort()` and `choice()`. They verify that sorting really produces the numbers `1...10` in increasing order and that any randomly picked number is indeed in the sequence it was picked from.

**Python source file** `code/modules/test_driven_development.py`

```
#!/usr/bin/env python
import random
import unittest

class TestSequenceFunctions(unittest.TestCase):
    def setUp(self):
        self.seq = range(10)

    def test_sortingworks(self):
        random.shuffle(self.seq)
        self.seq.sort()
        self.assertEqual(self.seq, range(10))

    def test_choicepicksfromsequence(self):
        element = random.choice(self.seq)
        self.assertTrue(element in self.seq)

if __name__ == '__main__':
    unittest.main()
```

Running the code snippet on the previous page produces the following output, indicating that the sequence functions do indeed work as expected.

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

When developing your code, write down a test, then implement the necessary functionality to make it pass successfully. Iterate these two alternating steps until your program does everything you need it to do. You should run all the tests frequently to make sure your changes don't accidentally break the existing functionality. The full documentation of the unittest module is available online at `https://docs.python.org/dev/library/unittest.html`.

## 6.8 Exercises

6.1 **Can't wait**: Use datetime to calculate the time remaining until your next birthday (or some other important event in the future). Notice the type of the result when calculating the difference between two dates.
cant_wait.py

6.2 **Complex pie**: Import the exponential function `exp` and the constant `pi` from `cmath` and verify that $e^{i\pi}$ is indeed -1. Note: The `cmath` module provides mathematical functions which also work for complex numbers. Use the suffix j to indicate an imaginary number.
complex_pie.py

6.3 **Feeling lucky**: Write a program that suggests lotto numbers to pick for the next drawing. Make sure your suggestion has no repeated numbers in them. *Bonus*: Write a unit test which randomly generates 100 suggestions and tests that all numbers are in the correct range and that none are repeated.
feeling_lucky.py

6.4 **Snakes on every plane**: Write a program that finds all the python files on the current file-system level and its subdirectories.
snakes_on_every_plane.py

6.5 **Four-letter words**: Write a regular expression which finds all the four-letter words in a text. *Bonus*: Write a regular expression which finds only the capitalized ones.
four_letter_words.py

# Chapter 7

# Numerical python

The `numpy` package provides tools to manipulate large data sets. The package typically needs to be installed separately (in addition to) your existing python installation (and with the matching version number). The Windows distribution PythonXY bundles them with the installer for convenience.

## 7.1 Arrays

The workhorse datatype of the `numpy` package is the $n$-dimensional array (or `ndarray`). Arrays can be created from python lists or by using one of the various numpy helper functions which provide different initial values:

**Python source file** `code/numpy/create.py`

```
#!/usr/bin/env python
import numpy as np

a = np.array([1,2,3]) # [1 2 3]
f = np.array([1,2,3], dtype=float) # [ 1.  2.  3.]
r = np.arange(5) # [0 1 2 3 4]
z = np.zeros(7) # [ 0.  0.  0.  0.  0.  0.  0.]
o = np.ones(9) # [ 1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

By default the arrays are one-dimensional, as indicated by their shape:

```
>>> z.shape
(7,)
```

It is also possible to convert nested lists/tuples or to reinterpret the contents of one-dimensional arrays as a multi-dimensional one by adjusting the shape:

**Python source file** code/numpy/shape.py

```
#!/usr/bin/env python
import numpy as np

# converted nested lists/tuples
n = np.array([
    (0,1,2,3,4),
    (5,6,7,8,9),
    (10,11,12,13,14),
])
print n

# reshape a given 1-dimensional array
a = np.arange(15)
r = a.reshape(3,5)

# both n and r are now a 2-dimensional array:
# [[ 0  1  2  3  4]
#  [ 5  6  7  8  9]
#  [10 11 12 13 14]]
```

In case you need to generate a bunch of numbers at even intervals covering a given range, you can use the built-in `np.linspace` function:

```
>>> np.linspace(0, 1, 5)
array([ 0., 0.25, 0.5, 0.75, 1.])
```

## Operators

Arithmetic operators and mathematical functions for `ndarrays` are applied piece-wise – including multiplication(!):

**Python source file** code/numpy/operators.py

```
#!/usr/bin/env python
import numpy as np
a, r = np.ones(3), np.random.random(3)

print r          # array([ 0.87831,  0.23283,  0.22790])
print 3*a        # array([3,3,3])
print a*r        # array([ 0.87831,  0.23283,  0.22790])
print a + r/2    # array([ 1.43915,  1.11641,  1.11395])
print np.exp(r) # array([ 2.40683,  1.26217,  1.25596])
```

In case you need the dot product instead, use the `np.dot()` function:

**Python source file** code/numpy/dot.py

```
#!/usr/bin/env python
import numpy as np

a = np.arange(5)
print np.dot(a,a)
# 30 (=1+4+9+16)
```

## Indexing

Indexing for one-dimensional arrays works as for lists/tuples:

**Python source file** code/numpy/index.py

```
#!/usr/bin/env python
import numpy as np

a = np.arange(10)
print a[::-1] # [9 8 7 6 5 4 3 2 1 0]
a[1::2] = -1
print(a) # [ 0 -1  2 -1  4 -1  6 -1  8 -1]
```

For multidimensional arrays, the indices and/or slices are given as a tuple:

**Python source file** code/numpy/ndslice.py

```
#!/usr/bin/env python
import numpy as np

a = np.arange(15).reshape(3,5)
print a
# [[ 0  1  2  3  4]
#  [ 5  6  7  8  9]
#  [10 11 12 13 14]]
print a[1,1] # 6
print a[1,:] #[5 6 7 8 9]
print a[0:2,2:4]
# [[2 3]
#  [7 8]]

b = np.arange(8).reshape(2,2,2)
# these all print the same:
print b[0]
print b[0,]
print b[0,:,:]
print b[0,...]
```

The dots (`...`) represent as many full slices as necessary for the array shape.

## 7.2 Matrices

Numpy also has a specialized `matrix` datatype which builds on ndarray but provides matrix functionality for arithmetic operations (rather than by-element) and that can be used for linear algebra:

**Python source file** `code/numpy/matrices.py`

```
1  #!/usr/bin/env python
2  import numpy as np
3
4  A = np.arange(1,5, dtype=float).reshape(2,2)
5  M = np.matrix(A)
6  print type(M)  # <class 'numpy.matrixlib.defmatrix.matrix'>
7
8  print M.T  # transpose
9  print M.I  # inverse
10
11 X = np.matrix([5.,7.])
12 Y = X.T
13 print Y
14 # [[5.]
15 #  [7.]]
16
17 print M*Y
18 # [[19.]
19 #  [43.]]
20
21 print np.linalg.solve(A, Y)  # solving linear equation
```

For more advanced linear algebraic calculations you should look into the `scipy` package which is also distributed with PythonXY.