The Python Tutorial

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, https://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.

This tutorial introduces the reader informally to the basic concepts and features of the Python language and system. It helps to have a Python interpreter handy for hands-on experience, but all examples are self-contained, so the tutorial can be read off-line as well.

For a description of standard objects and modules, see The Python Standard Library. The Python Language Reference gives a more formal definition of the language. To write extensions in C or C++, read Extending and Embedding the Python Interpreter and Python/C API Reference Manual. There are also several books covering Python in depth.

This tutorial does not attempt to be comprehensive and cover every single feature, or even every commonly used feature. Instead, it introduces many of Python's most noteworthy features, and will give you a good idea of the language's flavor and style. After reading it, you will be able to read and write Python modules and programs, and you will be ready to learn more about the various Python library modules described in The Python Standard Library.

The Glossary is also worth going through.

- 1. Whetting Your Appetite
- 2. Using the Python Interpreter
  - 2.1. Invoking the Interpreter
    - 2.1.1. Argument Passing
    - 2.1.2. Interactive Mode
  - 2.2. The Interpreter and Its Environment
    - 2.2.1. Source Code Encoding
- 3. An Informal Introduction to Python
  - 3.1. Using Python as a Calculator
    - 3.1.1. Numbers
    - 3.1.2. Strings
    - 3.1.3. Lists
  - 3.2. First Steps Towards Programming

- 4. More Control Flow Tools
  - 4.1. `if` Statements
  - 4.2. `for` Statements
  - 4.3. The `range()` Function
  - 4.4. `break` and `continue` Statements, and `else` Clauses on Loops
  - 4.5. `pass` Statements
  - 4.6. Defining Functions
  - 4.7. More on Defining Functions
    - 4.7.1. Default Argument Values
    - 4.7.2. Keyword Arguments
    - 4.7.3. Special parameters
      - 4.7.3.1. Positional-or-Keyword Arguments
      - 4.7.3.2. Positional-Only Parameters
      - 4.7.3.3. Keyword-Only Arguments
      - 4.7.3.4. Function Examples
      - 4.7.3.5. Recap
    - 4.7.4. Arbitrary Argument Lists
    - 4.7.5. Unpacking Argument Lists
    - 4.7.6. Lambda Expressions
    - 4.7.7. Documentation Strings
    - 4.7.8. Function Annotations
  - 4.8. Intermezzo: Coding Style
- 5. Data Structures
  - 5.1. More on Lists
    - 5.1.1. Using Lists as Stacks
    - 5.1.2. Using Lists as Queues
    - 5.1.3. List Comprehensions
    - 5.1.4. Nested List Comprehensions
  - 5.2. The `del` statement
  - 5.3. Tuples and Sequences
  - 5.4. Sets
  - 5.5. Dictionaries
  - 5.6. Looping Techniques
  - 5.7. More on Conditions
  - 5.8. Comparing Sequences and Other Types
- 6. Modules
  - 6.1. More on Modules
    - 6.1.1. Executing modules as scripts
    - 6.1.2. The Module Search Path
    - 6.1.3. “Compiled” Python files
  - 6.2. Standard Modules
  - 6.3. The `dir()` Function
  - 6.4. Packages
    - 6.4.1. Importing * From a Package
    - 6.4.2. Intra-package References
    - 6.4.3. Packages in Multiple Directories
- 7. Input and Output
  - 7.1. Fancier Output Formatting
    - 7.1.1. Formatted String Literals

        - 7.1.2. The String format() Method
        - 7.1.3. Manual String Formatting
        - 7.1.4. Old string formatting
    - 7.2. Reading and Writing Files
        - 7.2.1. Methods of File Objects
        - 7.2.2. Saving structured data with `json`
- 8. Errors and Exceptions
    - 8.1. Syntax Errors
    - 8.2. Exceptions
    - 8.3. Handling Exceptions
    - 8.4. Raising Exceptions
    - 8.5. User-defined Exceptions
    - 8.6. Defining Clean-up Actions
    - 8.7. Predefined Clean-up Actions
- 9. Classes
    - 9.1. A Word About Names and Objects
    - 9.2. Python Scopes and Namespaces
        - 9.2.1. Scopes and Namespaces Example
    - 9.3. A First Look at Classes
        - 9.3.1. Class Definition Syntax
        - 9.3.2. Class Objects
        - 9.3.3. Instance Objects
        - 9.3.4. Method Objects
        - 9.3.5. Class and Instance Variables
    - 9.4. Random Remarks
    - 9.5. Inheritance
        - 9.5.1. Multiple Inheritance
    - 9.6. Private Variables
    - 9.7. Odds and Ends
    - 9.8. Iterators
    - 9.9. Generators
    - 9.10. Generator Expressions
- 10. Brief Tour of the Standard Library
    - 10.1. Operating System Interface
    - 10.2. File Wildcards
    - 10.3. Command Line Arguments
    - 10.4. Error Output Redirection and Program Termination
    - 10.5. String Pattern Matching
    - 10.6. Mathematics
    - 10.7. Internet Access
    - 10.8. Dates and Times
    - 10.9. Data Compression
    - 10.10. Performance Measurement
    - 10.11. Quality Control
    - 10.12. Batteries Included
- 11. Brief Tour of the Standard Library — Part II
    - 11.1. Output Formatting
    - 11.2. Templating
    - 11.3. Working with Binary Data Record Layouts

- 11.4. Multi-threading
- 11.5. Logging
- 11.6. Weak References
- 11.7. Tools for Working with Lists
- 11.8. Decimal Floating Point Arithmetic

- 12. Virtual Environments and Packages
  - 12.1. Introduction
  - 12.2. Creating Virtual Environments
  - 12.3. Managing Packages with pip
- 13. What Now?
- 14. Interactive Input Editing and History Substitution
  - 14.1. Tab Completion and History Editing
  - 14.2. Alternatives to the Interactive Interpreter
- 15. Floating Point Arithmetic: Issues and Limitations
  - 15.1. Representation Error
- 16. Appendix
  - 16.1. Interactive Mode
    - 16.1.1. Error Handling
    - 16.1.2. Executable Python Scripts
    - 16.1.3. The Interactive Startup File
    - 16.1.4. The Customization Modules

1. Whetting Your Appetite

If you do much work on computers, eventually you find that there's some task you'd like to automate. For example, you may wish to perform a search-and-replace over a large number of text files, or rename and rearrange a bunch of photo files in a complicated way. Perhaps you'd like to write a small custom database, or a specialized GUI application, or a simple game.

If you're a professional software developer, you may have to work with several C/C++/Java libraries but find the usual write/compile/test/re-compile cycle is too slow. Perhaps you're writing a test suite for such a library and find writing the testing code a tedious task. Or maybe you've written a program that could use an extension language, and you don't want to design and implement a whole new language for your application.

Python is just the language for you.

You could write a Unix shell script or Windows batch files for some of these tasks, but shell scripts are best at moving around files and changing text data, not well-suited for GUI applications or games. You could write a C/C++/Java program, but it can take a lot of development time to get even a first-draft program. Python is simpler to use, available on Windows, Mac OS X, and Unix operating systems, and will help you get the job done more quickly.

Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than shell scripts or batch files can offer. On the other hand, Python also offers much more error checking than C, and, being a *very-high-level language*, it has high-level data types built in, such as flexible arrays and dictionaries. Because of its more general data types Python is applicable to a much larger problem domain than Awk or even Perl, yet many things are at least as easy in Python as in those languages.

Python allows you to split your program into modules that can be reused in other Python programs. It comes with a large collection of standard modules that you can use as the basis of your programs — or as examples to start learning to program in Python. Some of these modules provide things like file I/O, system calls, sockets, and even interfaces to graphical user interface toolkits like Tk.

Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write throw-away programs, or to test functions during bottom-up program development. It is also a handy desk calculator.

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

- the high-level data types allow you to express complex operations in a single statement;

- statement grouping is done by indentation instead of beginning and ending brackets;
- no variable or argument declarations are necessary.

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

By the way, the language is named after the BBC show “Monty Python’s Flying Circus” and has nothing to do with reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged!

Now that you are all excited about Python, you’ll want to examine it in some more detail. Since the best way to learn a language is to use it, the tutorial invites you to play with the Python interpreter as you read.

In the next chapter, the mechanics of using the interpreter are explained. This is rather mundane information, but essential for trying out the examples shown later.

The rest of the tutorial introduces various features of the Python language and system through examples, beginning with simple expressions, statements and data types, through functions and modules, and finally touching upon advanced concepts like exceptions and user-defined classes.

2. Using the Python Interpreter

2.1. Invoking the Interpreter

The Python interpreter is usually installed as `/usr/local/bin/python3.8` on those machines where it is available; putting `/usr/local/bin` in your Unix shell's search path makes it possible to start it by typing the command:

```
python3.8
```

to the shell. [1] Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

On Windows machines where you have installed Python from the Microsoft Store, the `python3.8` command will be available. If you have the py.exe launcher installed, you can use the `py` command. See Excursus: Setting environment variables for other ways to launch Python.

Typing an end-of-file character (`Control-D` on Unix, `Control-Z` on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following command: `quit()`.

The interpreter's line-editing features include interactive editing, history substitution and code completion on systems that support the GNU Readline library. Perhaps the quickest check to see whether command line editing is supported is typing `Control-P` to the first Python prompt you get. If it beeps, you have command line editing; see Appendix Interactive Input Editing and History Substitution for an introduction to the keys. If nothing appears to happen, or if `^P` is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a *script* from that file.

A second way of starting the interpreter is `python -c command [arg] ...`, which executes the statement(s) in *command*, analogous to the shell's `-c` option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote *command* in its entirety with single quotes.

Some Python modules are also useful as scripts. These can be invoked using `python -m module [arg] ...`, which executes the source file for *module* as if you had spelled out its full name on the command line.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing `-i` before the script.

All command line options are described in Command line and environment.

2.1.1. Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to the `argv` variable in the `sys` module. You can access this list by executing `import sys`. The length of the list is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string. When the script name is given as `'-'` (meaning standard input), `sys.argv[0]` is set to `'-'`. When `-c` *command* is used, `sys.argv[0]` is set to `'-c'`. When `-m` *module* is used, `sys.argv[0]` is set to the full name of the located module. Options found after `-c` *command* or `-m` *module* are not consumed by the Python interpreter's option processing but left in `sys.argv` for the command or module to handle.

2.1.2. Interactive Mode

When commands are read from a tty, the interpreter is said to be in *interactive mode*. In this mode it prompts for the next command with the *primary prompt*, usually three greater-than signs (`>>>`); for continuation lines it prompts with the *secondary prompt*, by default three dots (`...`). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
$ python3.8
Python 3.8 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this `if` statement:

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

For more on interactive mode, see Interactive Mode.

2.2. The Interpreter and Its Environment

2.2.1. Source Code Encoding

By default, Python source files are treated as encoded in UTF-8. In that encoding, characters of most languages in the world can be used simultaneously in string literals, identifiers and comments — although the standard library only uses ASCII characters for identifiers, a convention that any portable code should follow. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.

To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

```
# -*- coding: encoding -*-
```

where *encoding* is one of the valid `codecs` supported by Python.

For example, to declare that Windows-1252 encoding is to be used, the first line of your source code file should be:

```
# -*- coding: cp1252 -*-
```

One exception to the *first line* rule is when the source code starts with a UNIX "shebang" line. In this case, the encoding declaration should be added as the second line of the file. For example:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

Footnotes

[1] On Unix, the Python 3.x interpreter is by default not installed with the executable named `python`, so that it does not conflict with a simultaneously installed Python 2.x executable.

3. An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (>>> and …): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, `#`, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1. Using Python as a Calculator

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, `>>>`. (It shouldn't take long.)

3.1.1. Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` work just like in most other languages (for example, Pascal or C); parentheses (`()`) can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

The integer numbers (e.g. `2`, `4`, `20`) have type `int`, the ones with a fractional part (e.g. `5.0`, `1.6`) have type `float`. We will see more about numeric types later in the tutorial.

Division (/) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use %:

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

With Python, it is possible to use the ** operator to calculate powers [1]:

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

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

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 4 * 3.75 - 1
14.0
```

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

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
```

```
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

This variable should be treated as read-only by the user. Don't explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to `int` and `float`, Python supports other types of numbers, such as `Decimal` and `Fraction`. Python also has built-in support for complex numbers, and uses the `j` or `J` suffix to indicate the imaginary part (e.g. `3+5j`).

3.1.2. Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result [2]. `\` can be used to escape quotes:

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes. The `print()` function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
>>> print('"Isn\'t," they said.')
"Isn't," they said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```

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

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

String literals can span multiple lines. One way is using triple-quotes: """...""" or '''...'''. End of lines are automatically included in the string, but it's possible to prevent this by adding a \ at the end of the line. The following example:

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

produces the following output (note that the initial newline is not included):

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

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

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```
>>> 'Py' 'thon'
'Python'
```

This feature is particularly useful when you want to break long strings:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

This only works with two literals though, not with variables or expressions:

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
  File "<stdin>", line 1
```

```
    prefix 'thon'
                ^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  File "<stdin>", line 1
    ('un' * 3) 'ium'
                   ^
SyntaxError: invalid syntax
```

If you want to concatenate variables or a variable and a literal, use `+`:

```
>>> prefix + 'thon'
'Python'
```

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

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

Note that since -0 is the same as 0, negative indices start from -1.

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

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2]   # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]   # characters from position 4 (included) to the end
'on'
>>> word[-2:]  # characters from the second-last (included) to the end
'on'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0…6 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

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

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python strings cannot be changed — they are immutable. Therefore, assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

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

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

See also:

Text Sequence Type — str
Strings are examples of *sequence types*, and support the common operations supported by such types.

String Methods
Strings support a large number of methods for basic transformations and searching.

Formatted string literals
String literals that have embedded expressions.

Format String Syntax
Information about string formatting with `str.format()`.

printf-style String Formatting
The old formatting operations invoked when strings are the left operand of the `%` operator are described in more detail here.

3.1.3. Lists

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

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in sequence types), lists can be indexed and sliced:

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

Lists also support operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the `append()` *method* (we will see more about methods later):

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

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

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
```

```
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

The built-in function `len()` also applies to lists:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2. First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
0
1
1
2
3
5
8
```

This example introduces several new features.

- The first line contains a *multiple assignment*: the variables `a` and `b` simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.

- The `while` loop executes as long as the condition (here: `a < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: `<` (less than), `>` (greater than), `==` (equal to), `<=` (less than or equal to), `>=` (greater than or equal to) and `!=` (not equal to).
- The *body* of the loop is *indented*: indentation is Python's way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; all decent text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.
- The `print()` function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```
>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536
```

The keyword argument *end* can be used to avoid the newline after the output, or end the output with a different string:

```
>>> a, b = 0, 1
>>> while a < 1000:
...     print(a, end=',')
...     a, b = b, a+b
...
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

Footnotes

[1] Since `**` has higher precedence than `-`, `-3**2` will be interpreted as `-(3**2)` and thus result in `-9`. To avoid this and get `9`, you can use `(-3)**2`.

[2] Unlike other languages, special characters such as `\n` have the same meaning with both single (`'...'`) and double (`"..."`) quotes. The only difference between the two is that within single quotes you don't need to escape `"` (but you have to escape `\'`) and vice versa.