

Everything is an Object

- **Each object x has:**
 - Unique identifier, an integer, returned by `id(x)`; think memory -> FIXED
 - Type, returned by `type(x)` -> FIXED
 - Some content (value) -> may or may not be changed ("mutated")
[a value can itself consist of one or more objects!]
 - Zero or more methods provided by type object to get and/or change content
 - Zero or more names (aliases)
- **Names are NOT properties of an object:**
 - They are stored in separate namespaces
 - Dictionaries of name → object reference (id) pairs

Types are NOT properties of a name

Python is using a pure object model where classes are instances of a meta-class "type" (in Python, the terms "type" and "class" are synonyms). And "type" is the only class which is an instance of itself:

```
>>> type(42)
<class 'int'>
>>> type(int)      # same as type(type(42))
<class 'type'>
>>> type(type)     # same as type(type(type(42)))
<class 'type'>
```

This object model can be useful when we want information about a particular resource in Python. Except for the Python keywords (e.g. if, def, globals), using "type(<name>)" or "dir(<name>)" -or just type the resource name and press enter- will work on pretty much anything.

```
>>> super
<class 'super'>
>>> abs
<built-in function abs>
>>> str
<class 'str'>
>>> dir
<built-in function dir>
```

As we have already mentioned, in some other languages some entities are objects and some are not. In Python, everything is an object – everything is an instance of some class. In earlier versions of Python a distinction was made between built-in types and user-defined classes, but these are now completely indistinguishable. Classes and types are themselves objects, and they are of type type.

Python Identifiers (Names)

Names used to identify things in Python:

- Variable, function, class, module, or other object

Identifier in Python 2:

- Starts with a letter A to Z or a to z or an underscore (`_`)
- Followed by zero or more letters, underscores, and digits (0 to 9)
- No punctuation characters such as `@`, `$` or `%` are allowed
- No limit on length
- Case-sensitive

Python 3 allows Unicode codepoints from outside ASCII range:

- So you could rename the summation function: `Σ = sum`
- Not all codepoints: e.g. not `√`

www.spiraltrain.nl

Python Intro

2

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.

It is also possible to specify a different encoding for source files. In order to do this, put one more special comment line right after the `#!` line to define the source file encoding:

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

With that declaration, everything in the source file will be treated as having the encoding encoding instead of UTF-8. The list of possible encodings can be found in the Python Library Reference, in the section on codecs. For example, if your editor of choice does not support UTF-8 encoded files and insists on using some other encoding, say Windows-1252, you can write:

```
# -*- coding: cp-1252 -*-
```

and still use all characters in the Windows-1252 character set in the source files. The special encoding comment must be in the first or second line within the file.

Python Reserved Words

- May not be used as constant or variable or other identifier name

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield



www.spiraltrain.nl

Python Intro

3

When you use Python interactively, it is frequently handy to have some standard commands executed every time the interpreter is started. You can do this by setting an environment variable named `PYTHONSTARTUP` to the name of a file containing your start-up commands. This is similar to the `.profile` feature of the Unix shells.

This file is only read in interactive sessions, not when Python reads commands from a script, and not when `/dev/tty` is given as the explicit source of commands (which otherwise behaves like an interactive session). It is executed in the same namespace where interactive commands are executed, so that objects that it defines or imports can be used without qualification in the interactive session. You can also change the prompts `sys.ps1` and `sys.ps2` in this file.

If you want to read an additional start-up file from the current directory, you can program this in the global start-up file using code like `if os.path.isfile('.pythonrc.py'):`
`exec(open('.pythonrc.py').read())`.

If you want to use the startup file in a script, you must do this explicitly in the script:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    exec(open(filename).read())
```

Naming conventions

- `_*` Names starting with underscore are treated as “private” and not imported by `from module import *`.
- `__*__` If starts and ends with a double underscore: name of “special method”. Defined by interpreter and its implementation (including standard library).
- `__*` Class-private names: when used within the context of a class definition, re-written to mangled form to help avoid name clashes between “private” attributes of base and derived classes.

Stylistic (PEP8) conventions:

- Use underscores to separate multi-word names: `name_of_some_variable`
- Capitalize classnames: `Some_Class`

www.spiraltrain.nl

Python Intro

4

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.

It is also possible to specify a different encoding for source files. In order to do this, put one more special comment line right after the `#!` line to define the source file encoding:

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

With that declaration, everything in the source file will be treated as having the encoding encoding instead of UTF-8. The list of possible encodings can be found in the Python Library Reference, in the section on codecs. For example, if your editor of choice does not support UTF-8 encoded files and insists on using some other encoding, say Windows-1252, you can write:

```
# -*- coding: cp-1252 -*-
```

and still use all characters in the Windows-1252 character set in the source files. The special encoding comment must be in the first or second line within the file.

Comments in Python

- **Comments starts with:**

- A hash sign (#) that is not inside a string literal
- All characters after the # and up to the physical line end are part of the comment

```
# First comment
print("Hello, Python!") # second comment
```

- **Will produce following result:**

```
Hello, Python!
```

- **Comment may be on same line after statement or expression:**

```
name = "Einstein" # This is again comment
```

- **Comment multiple lines as follows:**

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

www.spiraltrain.nl

Python Intro

5

```
#!/usr/bin/python
# Filename: helloworld.py
print('Hello World')
```

Let us consider the first two lines of the program. These are called *comments* - anything to the right of the # symbol is a comment and is mainly useful as notes for the reader of the program.

Python does not use comments except for the special case of the first line here. It is called the *shebang line* - whenever the first two characters of the source file are #! followed by the location of a program, this tells your Linux/Unix system that this program should be run with this interpreter when you *execute* the program.

Use comments sensibly in your program to explain some important details of your program - this is useful for readers of your program so that they can easily understand what the program is doing. Remember, that person can be yourself after six months!

The comments are followed by a Python *statement* - this just prints the text 'Hello World'. The print is actually an operator and 'Hello World' is referred to as a string.

Lines and Indentation

- **Blocks of code are denoted by line indentation:**
 - No braces for blocks of code for class and function definitions or flow control

- **Number of spaces in the indentation is variable:**

- All statements within the block must be indented the same amount

```
if True:          # Both blocks in first example are fine
    print("True")
else:
    print("False")
```

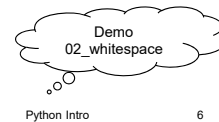
- **Other example:**

```
if True:          # Second line in second block will generate an error
    print("Answer")
    print("True")
else:
    print("Answer")
    print("False")
```

- **print() without new line is written as follows:**

```
print('hello python', end='')
```

www.spiraltrain.nl



Whitespace is important in Python. Actually, whitespace at the beginning of the line is important. This is called indentation. Leading whitespace (spaces and tabs) at the beginning of the logical line is used to determine the indentation level of the logical line, which in turn is used to determine the grouping of statements. This means that statements which go together must have the same indentation. Each such set of statements is called a block. One thing you should remember is how wrong indentation can give rise to errors. For example:

```
i = 5
    print('Value is', i); # Error! Single space at start line
print 'I repeat, the value is', i
```

When you run this, you get the following error:

File "02_whitespace.py", line 4

```
print 'Value is', i # Error! Single space at start line
^
```

SyntaxError: invalid syntax

Notice that there is a single space at the beginning of the second line. The error indicated by Python tells us that the syntax of the program is invalid i.e. the program was not properly written. What this means to you is that you cannot arbitrarily start new blocks of statements (except for the main block which you have been using all along, of course).

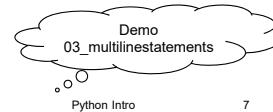
Do not use a mixture of tabs and spaces for the indentation as it does not work across different platforms properly. I strongly recommend that you use a single tab or two or four spaces for each indentation level. Choose any of these three indentation styles. More importantly, choose one and use it consistently i.e. use that indentation style only.

Multi Line Statements

- **Statements in Python typically end with a newline**
- **Python allows the use of the line continuation character (\):**
 - Denotes that the line should continue
- **Statements contained within the [], {}, or () brackets:**
 - No need to use the line continuation character
- **On a single line:**
 - Semicolon (;) allows multiple statements on single line
 - Neither statement should start a new code block

```
total = item_one + \
        item_two + \
        item_three
```

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```



www.spiraltrain.nl

Python Intro

7

A physical line is what you see when you write the program. A logical line is what Python sees as a single statement. Python implicitly assumes that each physical line corresponds to a logical line.

An example of a logical line is a statement like `print('Hello World');`

If this was on a line by itself (as you see it in an editor), then this also corresponds to a physical line. Implicitly, Python encourages the use of a single statement per line which makes code more readable.

An example of writing a logical line spanning many physical lines follows. This is referred to as explicit line joining.

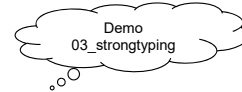
```
s = 'This is a string. \
This continues the string. ';
print(s);
```

This gives the output:

```
This is a string. This continues the string.
```

Python Data Types

- **Data stored in memory can be of many types:**
 - Persons age is stored as a numeric value
 - Persons address is stored as alphanumeric characters
- **A type (or class) determines:**
 - the operations possible on them
 - storage method for each of them
- **Important Python data types:**
 - Numbers
 - String
 - List
 - Tuple
 - Set
 - Dictionary



www.spiraltrain.nl

Variables and Data Types

8

Strong Typing

While Python allows you to be very flexible with your types, you must still be aware of what those types are. Certain operations will require certain types as arguments.

```
>>> 'Day ' + 1
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: cannot concatenate 'str' and 'int' objects
```

This behavior is different from some other loosely-typed languages. If you were to do the same thing in JavaScript, you would get a different result.

```
c:>'Day ' + 1
```

```
Day 1
```

In Python, however, it is possible to change the type of an object through builtin functions.

```
>>> 'Day ' + str(1)
```

```
'Day 1'
```

This type conversion can be a necessity to get the outcome that you want. For example, to force float division instead of integer division. Without an explicit conversion, the division of two integers will result in a third integer.

```
>>> 10 / 3
```

```
3
```

By converting one of the operands to a float, Python will perform float division and give you the result you were looking for (at least within floating point precision).

Variables

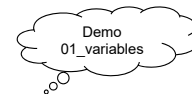
- **Variables (names) are introduced using some form of assignment:**

- They always refer to some object
- Names can be erased (`del name`) but object will persist as long as it "has" other names (aliases)

- **Equal sign = is used to assign values to variables:**

- Operand to the left of the = operator is the name of the variable
- Operand to the right of the = operator is the value stored in the variable

```
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string
```



www.spiraltrain.nl

Variables and Data Types

9

Using just literal constants can soon become boring - we need some way of storing any information and manipulate them as well. This is where *variables* come into the picture. Variables are exactly what they mean - their value can vary i.e. you can store anything using a variable. Variables are just parts of your computer's memory where you store some information. Unlike literal constants, you need some method of accessing these variables and hence you give them names.

Identifier Naming

Variables are examples of identifiers. *Identifiers* are names given to identify *something*. There are some rules you have to follow for naming identifiers:

The first character of the identifier must be a letter of the alphabet (upper or lowercase) or an underscore ('_').

The rest of the identifier name can consist of letters (upper or lowercase), underscores ('_') or digits (0-9).

Identifier names are case-sensitive. For example, `myname` and `myName` are **not** the same. Note the lowercase n in the former and the uppercase N in the latter.

Examples of *valid* identifier names are `i`, `__my_name`, `name_23` and `a1b2_c3`.

Examples of *invalid* identifier names are `2things`, `this is spaced out` and `my-name`.

Variables must be "defined" (assigned a value) before they can be used, or an error will occur.

Python refers to anything used in a program as an *object*. This is meant in the generic sense. Instead of saying 'the *something*', we say 'the *object*'.

Note for Object Oriented Programming users:

Python is strongly object-oriented in the sense that everything is an object including numbers, strings and even functions.

Multiple Assignment

- **Simultaneous assignment of values:**

```
a = b = c = 1
```

- **Integer object is created with the value 1:**

- All three variables are assigned to (refer to) the same memory location

- **Assignment of multiple objects to multiple variables:**

```
a, b, c = 1, 2, "john"
```

- Integer object with value 1 assigned to variable a
- Integer object with value 2 assigned to variable b
- String object with value "john" assigned to variable c



www.spiraltrain.nl

Variables and Data Types

10

Defining:

A variable in Python is defined through assignment. There is no concept of declaring a variable outside of that assignment.

```
>>> ten = 10
>>> ten
10
```

Dynamic Typing

In Python, while the value that a variable points to has a type, the variable itself has no strict type in its definition. You can re-use the same variable to point to an object of a different type. It may be helpful to think of variables as "labels" associated with objects.

```
>>> ten = 10
>>> ten
10
>>> ten = 'ten'
>>> ten
'ten'
```

Note for C/C++ Programmers

Variables are used by just assigning them a value. No declaration or data type definition is needed/used.

Numerical Types

- **Python supports different types to store numeric values:**
 - `int` (signed integers):
 - Integers or ints, are positive or negative whole numbers with no decimal point
 - They are of unlimited size, can also be represented in octal and hexadecimal
 - `float` (floating point real values):
 - Floats, represent real numbers
 - Written with a decimal point dividing the integer and fractional parts
 - `complex` (complex numbers):
 - Have form $a + bJ$, with a and b floats and J (or j) represents the square root of i
 - a is the real part of the number and b is the imaginary part
 - Complex numbers are not used much in Python programming
- **Boolean type is a subtype of the integer type:**
 - Boolean values behave like values 0 and 1
 - When converted to a string in string context `"False"` or `"True"` are returned

www.spiraltrain.nl

Variables and Data Types

11

Literal Constants

An example of a literal constant is a number like 5, 1.23, 9.25e-3 or a string like 'This is a string' or "It's a string!". It is called a literal because it is *literal* - you use its value literally. The number 2 always represents itself and nothing else - it is a constant because its value cannot be changed. Hence, all these are referred to as literal constants.

Numbers

Numbers in Python are of three types - integers, floating point and complex numbers.

Examples of integers are 2 and 111 which are just whole numbers.

Examples of floating point numbers (or *floats* for short) are 3.23 and 52.3E-4. The E notation indicates powers of 10. In this case, 52.3E-4 means $52.3 * 10^{-4}$.

Examples of complex numbers are $(-5+4j)$ and $(2.3 - 4.6j)$

The equal sign (`=`) is used to assign a value to a variable.

Fractions aren't lost when dividing integers.

Note: You might not see exactly the same result; floating point results can differ from one machine to another. We will say more later about controlling the appearance of floating point output. See *Floating Point Arithmetic: Issues and Limitations* in the documentation for a full discussion of some of the subtleties of floating point numbers and their representations.

To do integer division and get an integer result, discarding any fractional result, there is another operator, `//`

Number Examples

int	float	complex
10	0.0	3.14j
-100	15.20	45.j
535633629843	-21.9	9.322e-36j
11 = 0b01011 = 0o13 = 0xB	32.3e18	.876j
0xDEFA BCEC BDAE	-90.	-.6545+0J
0x69	-32.54e100	3e+26J
-0x260	70.2E-12	4.53e-7j

- **Python 2 had separate `int` and `long` types for integer numbers:**
 - In Python 3, there is only one `int` type, which behaves like `long` type in Python 2
- **All integers except 0, have meaning `True`:**
 - Only 0 behaves as `False`



www.spiraltrain.nl

Variables and Data Types

12

Complex numbers are also supported; imaginary numbers are written with a suffix of `j` or `J`. Complex numbers with a nonzero real component are written as `(real+imagj)`, or can be created with the `complex(real, imag)` function.

Complex numbers are always represented as two floating point numbers, the real and imaginary part. To extract these parts from a complex number `z`, use `z.real` and `z.imag`.

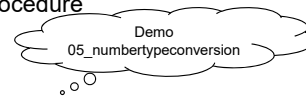
```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

The conversion functions to floating point and integer (`float()`, `int()`) don't work for complex numbers — there is not one correct way to convert a complex number to a real number. Use `abs(z)` to get its magnitude (as a float) or `z.real` to get its real part:

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a) # sqrt(a.real**2 + a.imag**2)
5.0
```

Type conversion: constructing new objects

- **Built-in constructors to perform conversion between data types:**
 - Functions return a new object representing the converted value
- `int(x [,base])`:
 - Converts `x` to a plain integer, `base` specifies the base if `x` is a string
- `complex(x)`:
 - Converts `x` to a complex number with real part `x` and imaginary part zero
- `complex(x, y)`:
 - Converts `x` and `y` to a complex number with real part `x` and imaginary part `y`
 - `x` and `y` are numeric expressions
- `float(x)`:
 - Converts `x` to a floating-point number
- `bool([x])`:
 - Convert value to Boolean using standard truth testing procedure



www.spiraltrain.nl

Variables and Data Types

13

The conversion functions to floating point and integer (`float()`, `int()`) don't work for complex numbers — there is not one correct way to convert a complex number to a real number. Use `abs(z)` to get its magnitude (as a float) or `z.real` to get its real part:

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last): File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a) # sqrt(a.real**2 + a.imag**2)
5.0
```

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.

Strings in Python

- **Python uses quotes to denote string literals:**

- single ('), double (") and triple (" or ''')
- Same type of quote should start and end the string
- Triple quotes can be used to span the string across multiple lines

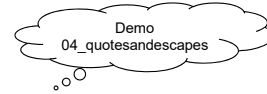
```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph.
                It is made up of multiple sentences."""

paragraph = "This is a paragraph. " "It is made up of multiple sentences."
```

- **Single quote in a single quoted string:**

- Displayed with an escape character \

```
'What \'s your name'
```



www.spiraltrain.nl

Python Intro

14

String literals can be defined with any of single quotes ('), double quotes (") or triple quotes (''' or '''). All give the same result with two important differences. If you quote with single quotes, you do not have to escape double quotes and vice-versa.

If you quote with triple quotes, your string can span multiple lines.

```
>>> 'hello' + " " + '''world'''
'hello world'
```

Suppose, you want to have a string which contains a single quote ('), how will you specify this string? For example, the string is What's your name?. You cannot specify 'What's your name?' because Python will be confused as to where the string starts and ends. So, you will have to specify that this single quote does not indicate the end of the string. This can be done with the help of what is called an *escape sequence*. You specify the single quote as \' - notice the backslash. Now, you can specify the string as 'What\'s your name?'.

Another way of specifying this specific string would be "What's your name?" i.e. using double quotes. Similarly, you have to use an escape sequence for using a double quote itself in a double quoted string. Also, you have to indicate the backslash itself using the escape sequence \\.

What if you wanted to specify a two-line string? One way is to use a triple-quoted string as shown above or you can use an escape sequence for the newline character - \n to indicate the start of a new line. An example is This is the first line\nThis is the second line . Another useful escape sequence to know is the tab - \t. There are many more escape sequences but I have mentioned only the most useful ones here

Python Strings

- **no character type:**
 - Characters are treated as strings of length one, thus also considered a substring
- **immutable data types like numbers:**
 - Changing the value of a string data type results in a newly allocated object
- **strings are sequences, sharing sequence operators with e.g. lists:**
 - slice operator is used to retrieve subsets from sequence:
 - to get the third character: `"abcde"[2]` will return `"c"`
 - to test for membership: `'d' in "abcde"[2]` will return `True`

www.spiraltrain.nl

Variables and Data Types

15

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes or double quotes.

The interpreter prints the result of string operations in the same way as they are typed for input: inside quotes, and with quotes and other funny characters escaped by backslashes, to show the precise value. The string is enclosed in double quotes if the string contains a single quote and no double quotes, else it's enclosed in single quotes. The `print()` function produces a more readable output for such input strings.

If we make the string literal a "raw" string, `\n` sequences are not converted to newlines, but the backslash at the end of the line, and the newline character in the source, are both included in the string as data. Thus, the example:

```
hello = r"This is a rather long string containing\n\
several lines of text much as you would do in C."
print(hello)
```

would print:

```
This is a rather long string containing\n\
several lines of text much as you would do in C.
```

Strings can be subscripted (indexed); like in C, the first character of a string has subscript (index) 0. There is no separate character type; a character is simply a string of size one. As in the Icon programming language, substrings can be specified with the *slice notation*: two indices separated by a colon.

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

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

String Formatting

Operator `%` when used with strings is the string format operator:

```
print("My name is %s and weight is %d kg!" % ('Albert', 55))
```

Result:

```
My name is Albert and weight is 55 kg!
```

In Python 3 new style of formatting available with `format()`:

- Allows complex variable substitutions and value formatting

```
s = '{0}, {1}, {2}'.format('a', 'b', 'c')
```

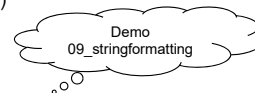
```
print(s) => 'a, b, c'
```

```
s = '{}', {}, {}'.format('a', 'b', 'c')
```

```
print(s) => 'a, b, c'
```

```
s = '{2}, {1}, {0}'.format('a', 'b', 'c')
```

```
print(s) => 'c, b, a'
```



www.spiraltrain.nl

Variables and Data Types

16

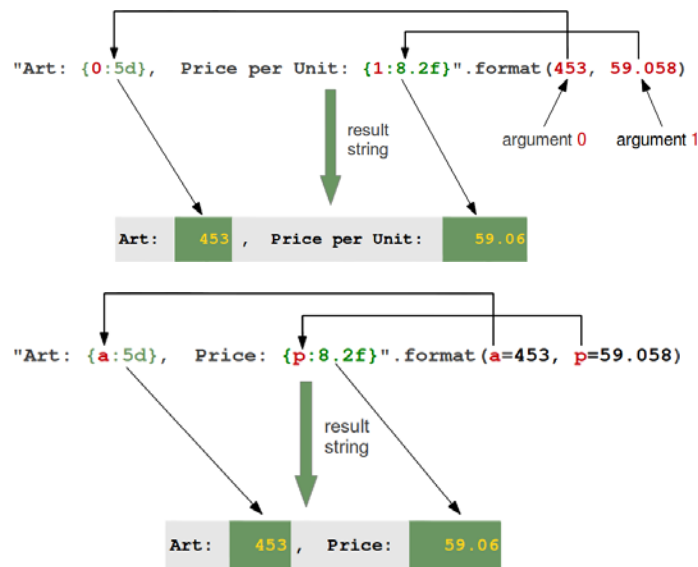
String objects have one unique built-in operation: the `%` operator (modulo). This is also known as the string formatting or interpolation operator. Given format `%` values (where format is a string), `%` conversion specifications in format are replaced with zero or more elements of values. The effect is similar to the using `sprintf()` in the C language.

If format requires a single argument, values may be a single non-tuple object. Otherwise, values must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

- 1) The `'%'` character, which marks the start of the specifier.
- 2) Mapping key (optional), consisting of a parenthesised sequence of characters (for example, (somenam)).
- 3) Conversion flags (optional), which affect the result of some conversion types.
- 4) Minimum field width (optional). If specified as an `'*'` (asterisk), the actual width is read from the next element of the tuple in values, and the object to convert comes after the minimum field width and optional precision.
- 5) Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'` (an asterisk), the actual width is read from the next element of the tuple in values, and the value to convert comes after the precision.
- 6) Length modifier (optional).
- 7) Conversion type.

Format Method



www.spiraltrain.nl

Variables and Data Types

17

The built-in string class provides the ability to do complex variable substitutions and value formatting via the `format()` method.

The `Formatter` class in the `string` module allows you to create and customize your own string formatting behaviors using the same implementation as the built-in `format()` method.

Convert a value to a “formatted” representation, as controlled by `format_spec`. The interpretation of `format_spec` will depend on the type of the value argument, however there is a standard formatting syntax that is used by most built-in types: Format Specification Mini-Language.

“Format specifications” are used within replacement fields contained within a format string to define how individual values are presented. They can also be passed directly to the built-in `format()` function. Each formattable type may define how the format specification is to be interpreted.

Format strings contain “replacement fields” surrounded by curly braces `{}`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: `{{` and `}}`. The grammar for a replacement field is as follows:

```
replacement_field ::= "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name        ::= arg_name ("." attribute_name | "[" element_index "]")*
arg_name          ::= [identifier | integer]
attribute_name    ::= identifier
element_index     ::= integer | index_string
index_string      ::= <any source character except "]"> +
conversion        ::= "r" | "s" | "a"
format_spec       ::= <described in the next section>
```

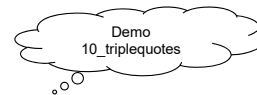
Triple Quotes

- **Allow strings to span multiple lines:**
 - Including verbatim NEWLINES, TABs, and any other special characters
- **Syntax consists of three consecutive single or double quotes:**

```
para_str = """this string is made up of several lines and non-printable
characters such as TAB ( \t ) and they will show up that way when displayed.
NEWLINES within the string, whether explicitly given like this within the
brackets [ \n ], or just a NEWLINE
within the variable assignment will also show up."""
print(para_str)
```

- **Result:**

```
this string is made up of several lines and non-printable
characters such as TAB (    ) and they will show up that way when
displayed. NEWLINES within the string, whether explicitly given like
this within the brackets [
], or just a NEWLINE
within the variable assignment will also show up.
```



www.spiraltrain.nl

Variables and Data Types

18

```
str.find(sub[, start[, end]]):
```

Return the lowest index in the string where substring sub is found, such that sub is contained in the slice s[start:end]. Optional arguments start and end are interpreted as in slice notation. Return -1 if sub is not found. The `find()` method should be used only if you need to know the position of sub. To check if sub is a substring or not, use the `in` operator:

```
>>> 'Py' in 'Python'
```

```
True
```

```
str.replace(old, new[, count]):
```

Return a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

```
str.startswith(prefix[, start[, end]])
```

Return True if string starts with the prefix, otherwise return False. prefix can also be a tuple of prefixes to look for. With optional start, test string beginning at that position. With optional end, stop comparing string at that position.

```
str.swapcase():
```

Return a copy of the string with uppercase characters converted to lowercase and vice versa.

```
str.upper():
```

Return a copy of the string converted to uppercase.

```
str.partition(sep):
```

Split the string at the first occurrence of sep, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

Raw and Unicode Strings

- **Raw String don't treat backslash as a special character:**
 - Every character in a raw string stays the way it was
- **In a normal String the backslash can be escaped:**

```
print('C:\\nowhere')
```
- **Result:** C:\nowhere
- **A raw string is made with the r in front of the String:**

```
print(r'C:\\nowhere')
```
- **Result:** C:\\nowhere
- **Since Python 3 all strings are stored as Unicode internally:**
 - In Unicode strings consist of (32-bit) code points
 - Allows more characters, including characters from most languages in the world
- **unicodedata module:**
 - Provides access to Unicode Character Database (UCD)
 - UCD contains character properties for all Unicode characters

www.spiraltrain.nl

Variables and Data Types

19

Starting with Python 3.0 all strings support Unicode (<http://www.unicode.org/>) .

Unicode has the advantage of providing one ordinal for every character in every script used in modern and ancient texts. Previously, there were only 256 possible ordinals for script characters. Texts were typically bound to a code page which mapped the ordinals to script characters. This lead to very much confusion especially with respect to internationalization (usually written as i18n — 'i' + 18 characters + 'n') of software. Unicode solves these problems by defining one code page for all scripts.

If you want to include special characters in a string, you can do so by using the Python Unicode-Escape encoding. The following example shows how:

```
>>> 'Hello\u0020World !'
'Hello World !'
```

The escape sequence `\u0020` indicates to insert the Unicode character with the ordinal value 0x0020 (the space character) at the given position.

Other characters are interpreted by using their respective ordinal values directly as Unicode ordinals. If you have literal strings in the standard Latin-1 encoding that is used in many Western countries, you will find it convenient that the lower 256 characters of Unicode are the same as the 256 characters of Latin-1.

Apart from these standard encodings, Python provides a whole set of other ways of creating Unicode strings on the basis of a known encoding.

To convert a string into a sequence of bytes using a specific encoding, string objects provide an `encode()` method that takes one argument, the name of the encoding. Lowercase names for encodings are preferred.

```
>>> "Äpfel".encode('utf-8')
b'\xc3\x84pfel'
```

Sequences

- **Most basic data structure in Python is the sequence:**
 - Each element of a sequence is assigned a number - its position or index
 - First index is zero, the second index is one, and so forth
 - Python distinguishes sequences on the basis of mutability
- **Immutable sequences:**
 - Contain objects that cannot change once created:
 - Objects may references to other objects and these other objects may be changed
 - Examples are Strings (Unicode items), Tuples, Bytes (bytes items)
- **Mutable sequences:**
 - Can be changed after they are created
 - Examples are Lists and Byte Arrays
- **Operations that work on all sequences types:**
 - indexing, slicing, adding, multiplying, and checking for membership
- **Python has built-in functions for:**
 - Finding the length of a sequence and its largest and smallest elements

www.spiraltrain.nl

Data Structures

20

Python recognises six sequence types: strings, Unicode strings, lists, tuples, buffers, and xrange objects.

String literals are written in single or double quotes. Unicode strings are much like strings, but are specified in the syntax using a preceding "u" character (e.g., u'cat', u"dog"). Lists are constructed with square brackets, separating items with commas. Tuples are constructed by the comma operator (without square brackets), with or without enclosing parentheses, but an empty tuple must have the enclosing parentheses, such as a, b, c or (). A single item tuple must have a trailing comma (e.g., '(d,)').

Buffer objects are not directly supported by Python syntax, but can be created by calling the builtin function `buffer()`. Concatenation and repetition are not supported, however.

Xrange objects are created using the `xrange()` function. They do not support slicing, concatenation or repetition. Using `in`, `not in`, `min()` or `max()` on them is inefficient.

Most sequence types support the following operations. The `in` and `not in` operations have the same priorities as the comparison operations. The `+` and `*` operators have the same priority as the corresponding numeric operations.

Below are the sequence operations sorted in ascending priority (s and t are sequences of the same type; n, y and j are integers).

`x in s`: True if an item of s is equal to x, else False. When s is a string (of either plain or Unicode format), the `in` and `not in` operations act like a substring test.

`x not in s`: False if an item of s is equal to x, else True. When s is a string (of either plain or Unicode format), the `in` and `not in` operations act like a substring test.

`s + t`: the concatenation of s and t. This type of concatenation is implementation dependent. A more reliable and consistent means for joining strings is to use `str.join()`. (6)

Lists

- **Lists are mutable sequences and similar to arrays in C:**
 - List contains items separated by commas and enclosed within square brackets []
 - Difference is that list items can be of different data type
- **Values are accessed using the slice operator [] and [:] :**
 - Indexes start at 0, then upwards and with end-1
 - Lists can be sliced, concatenated and so on
- **Lists respond to the + and * operators much like strings:**
 - Meaning is concatenation and repetition
 - Except that result is a new list, not a string
- **Lists respond to all sequence operations usable on strings:**

```
lst = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
print(lst[0])      # Prints first element of the list
print(lst[1:3])    # Prints elements starting from 2nd to 4th
print(lst * 2)     # Prints list two times
print(lst)        # Prints complete list
```

www.spiraltrain.nl

Data Structures

21

`s * n , n * s`: `n` shallow copies of `s` concatenated. Values of `n` that are less than zero are treated as zero.

`s[y]`: `y`'th item of `s`, origin 0. If `i` or `j` is negative, the index is oriented to the end of the string.

`s[y:j]`: slice of `s` from `y` to `j`. If `i` or `j` is negative, the index is oriented to the end of the string.

`s[y:j:k]`: slice of `s` from `y` to `j` with step `k`. If `i` or `j` is negative, the index is oriented to the end of the string.

`len(s)`: length of `s`

`min(s)`: smallest item of `s`

`max(s)`: largest item of `s`

One of the great advantages of Python as a programming language is the ease with which it allows you to manipulate containers. Containers (or collections) are an integral part of the language and, as you'll see, built in to the core of the language's syntax. As a result, thinking in a Pythonic manner means thinking about containers.

The first container type that we will look at is the list. A list represents an ordered, mutable collection of objects. You can mix and match any type of object in a list, add to it and remove from it at will.

To create an empty list, you can use empty square brackets or use the `list()` function with no arguments.

```
>>> l = []
```

```
>>> l
```

```
[]
```

Accessing and Updating Lists

- **Update single or multiple elements of lists:**
 - Slice list on the left-hand side of the assignment operator

```
l = ['physics', 'chemistry', 1997, 2000]
print("Value available at index 2: ")
print(l[2])    => 1997
l[2] = 2001
print("New value available at index 2: ")
print(l[2])    => 2001
```
- **Elements can be added to a list with `append()`:**

```
l = [1, 2, 3]
l.append(4)
l.append(5)
print(l)      => [1, 2, 3, 4, 5]
```

Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends. For example:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")    # Terry arrives
>>> queue.append("Graham")  # Graham arrives
>>> queue.popleft()         # First to arrive now leaves
'Eric'
>>> queue.popleft()         # Second to arrive now leaves
'John'
>>> queue                  # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

Multidimensional Lists

- **Multidimensional lists created by using other lists as elements**

```

multilist = [['a', 'b', 'c'], [1, 2, 3, 4]]

# first dimension
print(multilist[0][0])
print(multilist[0][1])
print(multilist[0][2])

# second dimension
print(multilist[1][0])
print(multilist[1][1])
print(multilist[1][2])
print(multilist[1][3])

# print(multilist[1][4]) # IndexError: list index out of range

```

Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```

>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]

```

Delete List Elements

- **Remove a list element with `del` or `remove()`**
- **Use `del` statement if you know index of element to remove:**

```
l = ['physics', 'chemistry', 1997, 2000]
del l[2]
print("After deleting value at index 2: ")
print(l)    => ['physics', 'chemistry', 2000]
```

- **Use `remove()` if want to remove element by value:**

```
l = ['physics', 'chemistry', 1997, 2000]
l.remove('physics')
print(l)    => ['chemistry', 1997, 2000]
```

- **Delete from a multidimensional list:**

```
multilist = [['a', 'b', 'c'], [1, 2, 3, 4]]
del multilist[0][2]
multilist.remove(['a', 'b', 'c'])
```



www.spiraltrain.nl

Data Structures

24

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` can also be used to delete entire variables:

```
>>> del a
```

Referencing the name `a` hereafter is an error (at least until another value is assigned to it).

List Comprehensions

List comprehensions provide a concise way to create lists from sequences. Common applications are to make lists where each element is the result of some operations applied to each member of the sequence, or to create a subsequence of those elements that satisfy a certain condition.

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. If the expression would evaluate to a tuple, it must be parenthesized.

Operations on sequences

Python Expression	Results	Description
<code>len([1,2,3])</code>	3	Length
<code>[1,2,3]+[4,5,6]</code>	<code>[1,2,3,4,5,6]</code>	Concatenation
<code>['go']*3</code>	<code>['go','go','go']</code>	Repetition
<code>3 in [1,2,3]</code>	True	Membership
<code>for x in [1,2,3]: print x</code>	1 2 3	Iteration

- **indexing and slicing:** `lst = ['one','two','three']`

Python Expression	Results	Description
<code>lst[2]</code>	<code>'three'</code>	Offsets start at zero
<code>lst[-2]</code>	<code>'two'</code>	Negative: count from the right
<code>lst[1:]</code>	<code>['two', 'three']</code>	Slicing fetches sections



www.spiraltrain.nl

Data Structures

25

Here we take a list of numbers and return a list of three times each number:

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
```

Now we get a little fancier:

```
>>> [[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

Using the if clause we can filter the stream:

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

A list is a data structure that holds an ordered collection of items i.e. you can store a *sequence* of items in a list. This is easy to imagine if you can think of a shopping list where you have a list of items to buy, except that you probably have each item on a separate line in your shopping list whereas in Python you put commas in between them. A list is an example of usage of objects and classes. When you use a variable `i` and assign a value to it, say integer 5 to it, you can think of it as creating an **object** (instance) `i` of **class** (type) `int`. In fact, you can see `help(int)` to understand this better. A class can also have **methods** i.e. functions defined for use with respect to that class only. You can use these pieces of functionality only when you have an object of that class. For example, Python provides an `append` method for the list class which allows you to add an item to the end of the list. For example, `mylist.append('an item')` will add that string to the list `mylist`. Note the use of dotted notation for accessing methods of the objects.

A class can also have **fields** which are nothing but variables defined for use with respect to that class only. You can use these variables/names only when you have an object of that class. Fields are also accessed by the dotted notation, for example, `mylist.field`.

Tuples

- **Tuple is a sequence of immutable Python objects:**
 - Consist of a number of values separated by commas
 - Tuples are sequences, just like lists, but tuples are faster because read-only
- **Main differences between lists and tuples:**
 - Lists are enclosed in brackets [] and their elements and size can be changed
 - Tuples are optionally enclosed in parentheses () and cannot be updated
- **Create tuple by assigning comma-separated values to variable:**

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = "a", "b", "c", "d "
```
- **Empty tuple is written as two parentheses containing nothing:**

```
tup1 = ()
```
- **For tuple containing a single value include a comma:**
 - Otherwise it will not be recognized as tuple

```
tup1 = (50,)    or: tup1 = 50,
```

www.spiraltrain.nl

Data Structures

26

Tuples are just like lists except that they are immutable like strings i.e. you cannot modify tuples. Tuples are defined by specifying items separated by commas within a pair of parentheses. Tuples are usually used in cases where a statement or a user-defined function can safely assume that the collection of values i.e. the tuple of values used will not change.

```
zoo = ('wolf', 'elephant', 'penguin')
print 'Number of animals in the zoo is', len(zoo)
new_zoo = ('monkey', 'dolphin', zoo)
print 'Number of animals in the new zoo is', len(new_zoo)
print 'All animals in new zoo are', new_zoo
print 'Animals brought from old zoo are', new_zoo[2]
print 'Last animal brought from old zoo is', new_zoo[2][2]
```

Output:

```
$ python using_tuple.py
Number of animals in the zoo is 3
Number of animals in the new zoo is 3
All animals in new zoo are ('monkey', 'dolphin', ('wolf',
'elephant', 'penguin'))
Animals brought from old zoo are ('wolf', 'elephant', 'penguin')
Last animal brought from old zoo is penguin
```

The variable zoo refers to a tuple of items. We see that the len function can be used to get the length of the tuple. This also indicates that a tuple is a sequence as well.

We are now shifting these animals to a new zoo since the old zoo is being closed. Therefore, the new_zoo tuple contains some animals which are already there along with the animals brought over from the old zoo. Back to reality, note that a tuple within a tuple does not lose its identity.

Accessing Values in Tuples

- **All of the general sequence operations can be applied on tuples:**

- Use square brackets for slicing along with index to obtain value, indices start at 0
- Use the + and * operators for concatenation and repetition

```
tup = (1, 2, 3, 4, 5, 6, 7)
tup[0] == 1
tup[1:5] == [2, 3, 4, 5]
```

- **Tuples are immutable and can be thought of as read-only lists:**

- Cannot update them or change values of tuple elements

```
tuple[2] = 1000 # TypeError
```

- **Portions of existing tuples can be used to create new tuples:**

```
tup1 = (12, 34.56)
tup2 = ('abc', 'xyz')
tup3 = tup1 + tup2
tup3 == (12, 34.56, 'abc', 'xyz')
```

www.spiraltrain.nl

Data Structures

27

We can access the items in the tuple by specifying the item's position within a pair of square brackets just like we did for lists. This is called the *indexing* operator. We access the third item in `new_zoo` by specifying `new_zoo[2]` and we access the third item in the third item in the `new_zoo` tuple by specifying `new_zoo[2][2]`. This is pretty simple once you've understood the idiom.

Tuple with 0 or 1 items. An empty tuple is constructed by an empty pair of parentheses such as `myempty = ()`. However, a tuple with a single item is not so simple. You have to specify it using a comma following the first (and only) item so that Python can differentiate between a tuple and a pair of parentheses surrounding the object in an expression i.e. you have to specify `singleton = (2,)` if you mean you want a tuple containing the item 2.

Note for Perl programmers

A list within a list does not lose its identity i.e. lists are not flattened as in Perl. The same applies to a tuple within a tuple, or a tuple within a list, or a list within a tuple, etc. As far as Python is concerned, they are just objects stored using another object, that's all. One of the most common usage of tuples is with the print statement. Here is an example:

```
age = 22; name = 'pipo'
print '%s is %d years old' % (name, age)
print 'Why is %s playing with that python?' % name
```

Output:

```
$ python print_tuple.py
pipo is 22 years old
Why is pipo playing with that python?
```

The print statement can take a string using certain specifications followed by the % symbol followed by a tuple of items matching the specification. The specifications are used to format the output in a certain way. The specification can be like %s for strings and %d for integers. The tuple must have items corresponding to these specifications in the same order.

Delete Tuple Elements

- **Removing individual tuple elements is not possible:**
 - Another tuple with undesired elements discarded must be created
- **Entire tuple can be removed with `del` statement:**

```
tup = ('physics', 'chemistry', 1997, 2000)
print tup => ('physics', 'chemistry', 1997, 2000)
del tup
print("After deleting tup: ")
print(tup)
```

- **Result after deleting tup:**

```
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    print tup;
NameError: name 'tup' is not defined
```

- **Exception since after `del tup` tuple is gone**

www.spiraltrain.nl

Data Structures

28

Tuples Have No Methods

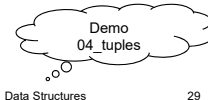
```
>>>t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t.append("new")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> t.remove("z")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> t.index("example")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'index'
>>> "z" in t
True
```

- 1 You can't add elements to a tuple. Tuples have no append or extend method.
- 2 You can't remove elements from a tuple. Tuples have no remove or pop method.
- 3 You can't find elements in a tuple. Tuples have no index method.
- 4 You can, however, use `in` to see if an element exists in the tuple.

Usage of Tuples

- **Tuples have many uses:**
 - Coordinate pairs, records from a database
- **Any set of comma-separated, objects default to tuples:**
 - Written without identifying symbols like brackets or parentheses

```
print('abc', -4.24e93, 'xyz')    => abc -4.24e+93 xyz
x, y = 1, 2
print("Value of x , y: ", x,y) => Value of x , y: 1 2
```
- **Tuples are faster than lists:**
 - It is faster to loop through a tuple than through a list
- **Tuples can make your code safer:**
 - Allows you to “write-protect” data that does not need to be changed
- **A tuple can have lists as items:**
 - The items in the lists are mutable however



www.spiraltrain.nl

Data Structures

29

On output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression).

Tuples have many uses. For example: (x, y) coordinate pairs, employee records from a database, etc. Tuples, like strings, are immutable: it is not possible to assign to the individual items of a tuple (you can simulate much of the same effect with slicing and concatenation, though). It is also possible to create tuples which contain mutable objects, such as lists.

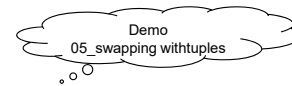
A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

Tuple Functions

- Only functions and methods that do not modify tuple available

Name	Description
<code>len(tuple)</code>	Gives the total length of the tuple
<code>max(tuple)</code>	Returns item from the tuple with max value
<code>min(tuple)</code>	Returns item from the tuple with min value
<code>tuple(seq)</code>	Converts a sequence into tuple
<code>tup.index(i)</code>	index of the first occurrence of <code>i</code> in <code>tup</code>
<code>tup.count(i)</code>	total number of occurrences of <code>i</code> in <code>tup</code>



www.spiraltrain.nl

Data Structures

30

Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of a list.

It makes your code safer if you “write-protect” data that does not need to be changed. Using a tuple instead of a list is like having an implied assert statement that shows this data is constant, and that special thought (and a specific function) is required to override that.

Remember that dictionary keys can be integers, strings, and “a few other types”? Tuples are one of those types. Tuples can be used as keys in a dictionary, but lists can't be used this way. Actually, it's more complicated than that. Dictionary keys must be immutable. Tuples themselves are immutable, but if you have a tuple of lists, that counts as mutable and isn't safe to use as a dictionary key. Only tuples of strings, numbers, or other dictionary-safe tuples can be used as dictionary keys.

Tuples are used in string formatting.

Tuples can be converted into lists, and vice-versa. The built-in tuple function takes a list and returns a tuple with the same elements, and the list function takes a tuple and returns a list. In effect, tuple freezes a list, and list thaws a tuple.

Bytes and Byte Arrays

- **Bytes:**
 - A bytes object is an immutable array
 - Items are 8-bit bytes, represented by integers in the range $0 \leq x < 256$
- **Bytes objects are constructed with:**
 - Bytes literals like `b'abc'` or the built-in function `bytes()`
 - Bytes objects can be decoded to strings via `decode()`
- **Byte Arrays:**
 - Bytearray object is a mutable array
 - Created by built-in `bytearray()` constructor
 - Same interface as immutable bytes objects
- **Extension modules `array` and `collections`:**
 - Provide additional mutable sequence example

Bytes are bytes; characters are an abstraction. An immutable sequence of Unicode characters is called a string. An immutable sequence of numbers-between-0-and-255 is called a bytes object.

To define a bytes object, use the `b` "byte literal" syntax. Each byte within the byte literal can be an ascii character or an encoded hexadecimal number from `\x00` to `\xff` (0–255).

The type of a bytes object is `bytes`.

Just like lists and strings, you can get the length of a bytes object with the built-in `len()` function.

Just like lists and strings, you can use the `+` operator to concatenate bytes objects. The result is a new bytes object.

Concatenating a 5-byte bytes object and a 1-byte bytes object gives you a 6-byte bytes object.

Just like lists and strings, you can use index notation to get individual bytes in a bytes object. The items of a string are strings; the items of a bytes object are integers. Specifically, integers between 0–255.

A bytes object is immutable; you can not assign individual bytes. If you need to change individual bytes, you can either use string slicing and concatenation operators (which work the same as strings), or you can convert the bytes object into a bytearray object.

The link between strings and bytes:

bytes objects have a `decode()` method that takes a character encoding and returns a string, and strings have an `encode()` method that takes a character encoding and returns a bytes object.

Sets

- **Data type representing unordered collection with no duplicates**

- **Sets:**

- Mutable set created by `set()` constructor or by using curly braces `{ }`

```
s = {1, 2}
print(s)    => {1, 2}
```

- Can be modified afterwards by several methods like `add()`

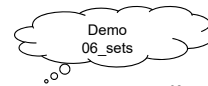
- **Frozen sets:**

- Immutable set created by `frozenset()` constructor

- **Support mathematical operations like union and intersection:**

```
a = set('abracadabra')
b = set('alacazam')
print(a)          # unique letters in a
                  # {'a', 'r', 'b', 'c', 'd'}
print(a - b)      # letters in a but not in b
                  # {'r', 'd', 'b'}
```

www.spiraltrain.nl



32

Python also includes a data type for sets. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the `set()` function can be used to create sets. Note: To create an empty set you have to use `set()`, not `{}`

A set is an unordered “bag” of unique values. A single set can contain values of any immutable datatype. Once you have two sets, you can do standard set operations like union, intersection, and set difference.

First things first. Creating a set is easy.

```
>>> a_set = {1}
>>> a_set
{1}
>>> type(a_set)
<class 'set'>
>>> a_set = {1, 2}
>>> a_set
{1, 2}
```

To create a set with one value, put the value in curly brackets `{}`.

Sets are actually implemented as classes, but don't worry about that for now.

To create a set with multiple values, separate the values with commas and wrap it all up with curly brackets.

Dictionary Characteristics

- **Mutable container type:**
 - Can store any number of Python objects, including other container types

```
d = {'Alice': '2341', 18: [1,8], 'Cecil': 3258}
```
- **In literal representation key is separated from value by a colon :**
 - Items (key-value pairs) are separated by commas
 - Notice curly braces
 - Empty is written with just two curly braces: { }
- **Keys are unique within a dictionary while values need not be**
- **Values of a dictionary can be of any type**
- **Keys must be immutable objects such as strings, numbers or tuples**
- **Items are unordered**

Mutable unordered set...:

A dictionary is a mutable unordered set of key:value pairs. Its values can contain references to any type of object.

In other languages, similar data structures are often called associative arrays or hash tables.

Not a sequence:

Unlike the string, list, and tuple, a dictionary is not a sequence. The sequences are indexed by a range of ordinal numbers. Hence, they are ordered.

Indexed by keys, not numbers:

Dictionaries are indexed by keys. According to the Python Tutorial, a key can be "any non-mutable type." Since strings and numbers are not mutable, you can always use a string or a number as a key in a dictionary.

What about a tuple as a key?:

You can use a tuple as a key if all of the items contained in the tuple are immutable. Hence a tuple to be used as a key can contain strings, numbers, and other tuples containing references to immutable objects.

What about a list as a key?:

You cannot use a list as a key because a list is mutable. That is to say, its contents can be modified using its append() method.

Accessing Values in Dictionary

- **Use square brackets along with the key to obtain its value:**

```
d = {'Name': 'Albert', 'Age': 55, 'Class': 'First'}
print("d['Name']: ", d['Name']) => d['Name']:  Albert
print("d['Age']: ", d['Age'])   => d['Age']:   55
```

- **Access data item with key not in dictionary gives error:**

```
d = {'Name': 'Albert', 'Age': 55, 'Class': 'First'}
print("looking up 'Robert': ", d['Robert'])
```

- **Result:**

```
d['Robert']:
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    print "d['Robert']: ", d['Robert']
KeyError: 'Robert'
```

www.spiraltrain.nl

Data Structures

34

```
class dict([arg])
```

Return a new dictionary initialized from an optional positional argument or from a set of keyword arguments. If no arguments are given, return a new empty dictionary. If the positional argument `arg` is a mapping object, return a dictionary mapping the same keys to the same values as does the mapping object. Otherwise the positional argument must be a sequence, a container that supports iteration, or an iterator object. The elements of the argument must each also be of one of those kinds, and each must in turn contain exactly two objects. The first is used as a key in the new dictionary, and the second as the key's value. If a given key is seen more than once, the last value associated with it is retained in the new dictionary.

If keyword arguments are given, the keywords themselves with their associated values are added as items to the dictionary. If a key is specified both in the positional argument and as a keyword argument, the value associated with the keyword is retained in the dictionary. For example, these all return a dictionary equal to `{"one": 1, "two": 2}`:

```
dict(one=1, two=2)
dict({'one': 1, 'two': 2})
dict(zip(('one', 'two'), (1, 2)))
dict(['two', 2], ['one', 1])
```

The first example only works for keys that are valid Python identifiers; the others work with any valid keys.

Updating Dictionaries

- **A dictionary can be updated by:**

- Adding a new entry or item (i.e., a key-value pair)
- Modifying an existing entry
- Deleting an existing entry

```
d = {'Name': 'Albert', 'Age': 55, 'Class': 'First'}
d['Age'] = 56 # Update existing entry
d['University'] = "Zurich"; # Add new entry
print("d['Age']: ", d['Age'])
print("d['University']: ", d['University'])
```

- **Result:**

```
d['Age']: 56
d['University']: Zurich
```

What does a dictionary look like?:

You can create a dictionary as an empty pair of curly braces, {}.

Alternatively, you can initially populate a dictionary by placing a comma-separated list of key:value pairs within the braces.

Can I add key:value pairs later?:

Later on, you can add new key:value pairs through indexing and assignment, using the new key as the index.

Keys must be unique, otherwise..:

Each of the keys within a dictionary must be unique. If you make an assignment using an existing key as the index, the old value associated with that key is overwritten by the new value.

New key:value pairs add to the dictionary:

If you make an assignment using a new key as the index, the new key:value pair will be added to the dictionary. Thus, the size of a dictionary can increase at runtime.

Removing key:value pairs:

You can use del to remove a key:value pair from a dictionary. Thus, the size of a dictionary can also shrink at runtime.

Add/modify key:value pairs:

You add or modify key:value pairs using assignment with the key as an index into the dictionary.

Delete Dictionary Elements

- **Several options to delete with `del` statement are available:**

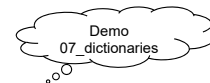
- Individual dictionary elements can be removed
- Entire contents of a dictionary can be cleared with `clear()`
- Entire dictionary in single operation with `del`

```
d = {'Name': 'Albert', 'Age': 55, 'Class': 'First'}
del d['Name'] # remove entry with key 'Name'
d.clear()     # remove all entries in d
del d         # delete entire dictionary
print("d['Age']: ", d['Age'])
```

- **Result :**

```
d['Age']:
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print "d['Age']: ", d['Age']
TypeError: 'type' object is unsubscriptable
```

- **Exception because `d` dictionary does not exist**



www.spiraltrain.nl

Data Structures

36

Fetching values associated with keys:

You extract a value from a dictionary using the associated key as an index. Attempting to extract a value using a non-existent key produces an error.

Getting a list of keys:

You can obtain a list of all of the keys currently contained in a dictionary by invoking the `keys()` method on the dictionary. This produces a list of keys in random order. You can invoke the `sort()` method on the list to sort the keys if need be.

Membership testing:

You can determine if a specific key is contained in the dictionary by invoking the `has_key()` method on the dictionary.

Getting a list of values:

You can obtain a list of all of the values currently contained in a dictionary by invoking the `values()` method on the dictionary.

Properties of Dictionary Keys

- **Dictionary values have no restrictions:**
 - Can be any Python object
 - **There two important points to about dictionary keys:**
 - No duplicate keys allowed
 - Keys must be immutable
 - **In case duplicate keys are encountered during assignment:**
 - Last assignment wins
- ```
>>> d = {'Name': 'Albert', 'Age': 7, 'Name': 'Robert'}

>>> d['Name']
Robert
```

www.spiraltrain.nl

Data Structures

37

Unlike some similar languages, string keys in Python must always be quoted.

```
>>> characters = {'hero': 'Othello', 'villain': 'Iago', 'friend': 'Cassio'}
>>> characters
{'villain': 'Iago', 'hero': 'Othello', 'friend': 'Cassio'}
>>> characters = dict(hero='Othello', villain='Iago', friend='Cassio')
>>> characters
{'villain': 'Iago', 'hero': 'Othello', 'friend': 'Cassio'}
```

**Accessing:**

Dictionary values can be accessed using the subscript operator except you use the key instead of an index as the subscript argument. The presence of keys can also be tested with the `in` keyword.

```
>>> if 'villain' in characters:
... print characters['villain']
...
Iago
```

**Adding:**

A new entry can be created where there is no existing key using the same subscripting notation and assignment.

```
>>> characters['beauty'] = 'Desdemona'
```

**Modifying:**

Existing entries are modified in exactly the same manner.

```
>>> characters['villain'] = 'Roderigo'
>>> characters
{'villain': 'Roderigo', 'hero': 'Othello', 'beauty': 'Desdemona', 'friend': 'Cassio'}
```

## Non Mutable Keys

- **Strings, numbers, or tuples can be dictionary keys:**
  - But something like `['Name']` is not allowed since a list is mutable

```
d = {'Name': 'Albert', 'Age': 55}
print("d['Name']: ", d['Name'])
```

- **Result:**

```
Traceback (most recent call last):
 File "test.py", line 3, in <module>
 d = {'Name': 'Albert', 'Age': 55}
TypeError: list objects are unhashable
```

### Failed Lookups.

If you use the subscript operator and the key is not found, a `KeyError` will be raised. If this behavior is not desired, using the `get()` method of the dictionary will return a supplied default value when the key is not found. If no default is provided, `None` is returned when the key is not found. The `get()` method does not alter the contents of the dictionary itself.

A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary. (For other containers see the built in list, set, and tuple classes, and the collections module.)

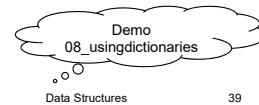
A dictionary's keys are almost arbitrary values. Values that are not hashable, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (such as 1 and 1.0) then they can be used interchangeably to index the same dictionary entry. (Note however, that since computers store floating-point numbers as approximations it is usually unwise to use them as dictionary keys.)

Dictionaries can be created by placing a comma-separated list of key: value pairs within braces, for example: `{'jack': 4098, 'sjoerd': 4127}` or `{4098: 'jack', 4127: 'sjoerd'}`, or by the `dict` constructor.

## Dictionary Functions and Methods

| Name                           | Description                                                    |
|--------------------------------|----------------------------------------------------------------|
| <code>len(dict)</code>         | Gives the total length or nr. of items in the dictionary.      |
| <code>str(dict)</code>         | Produces a printable string representation of a dictionary     |
| <code>fromkeys()</code>        | Create new dictionary with keys from seq , values set to value |
| <code>dict.clear()</code>      | Removes all elements of dictionary dict                        |
| <code>dict.copy()</code>       | Returns a shallow copy of dictionary dict                      |
| <code>dict.get(key)</code>     | For key key, returns value or default if key not in dictionary |
| <code>dict.popitem()</code>    | Remove and return arbitrary (key, value) pair from dictionary  |
| <code>dict.items()</code>      | Returns a list of dict's (key, value) tuple pairs              |
| <code>dict.keys()</code>       | Returns list of dictionary dict's keys                         |
| <code>dict.update(dict)</code> | Adds dictionary dict's key-values pairs to dict                |
| <code>dict.values()</code>     | Returns list of dictionary dict's values                       |

www.spiraltrain.nl



Data Structures

39

`dict.setdefault(key, default=None):`

Similar to `get()`, but will set `dict[key]=default` if key is not already in dict

`iter(d):`

Return an iterator over the keys of the dictionary. This is a shortcut for `iterkeys()`.

`clear():`

Remove all items from the dictionary.

`copy():`

Return a shallow copy of the dictionary.

`len(d):`

Return the number of items in the dictionary d.

`d[key]:`

Return the item of d with key key. Raises a `KeyError` if key is not in the map.

`has_key(key):`

Is a Python 2.x function. Test for the presence of key in the dictionary.

`has_key()` is deprecated in favor of `key in d` in Python 3.0

`items():`

Return a copy of the dictionary's list of (key, value) pairs.

`get(key[, default]):`

Return the value for key if key is in the dictionary, else default. If default is not given, it defaults to None, so that this method never raises a `KeyError`.

## Control Flow Constructs

- **Every programming has control flow constructs:**
  - Sequential constructs like statements
  - Branching constructs to take a certain direction
  - Iteration construct to repeat code a number of times
- **Branching Statements:**
  - `if`
  - `if...else`
  - `elif`
- **Iteration Statements:**
  - `for`
  - `while`
  - `continue`
  - `pass`

In computer science, control flow (or alternatively, flow of control) refers to the order in which the individual statements, instructions, or function calls of an imperative or a declarative program are executed or evaluated.

Within an imperative programming language, a control flow statement is a statement whose execution results in a choice being made as to which of two or more paths should be followed. For non-strict functional languages, functions and language constructs exist to achieve the same result, but they are not necessarily called control flow statements.

The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

- continuation at a different statement (unconditional branch or jump),
- executing a set of statements only if some condition is met (choice - i.e. conditional branch),
- executing a set of statements zero or more times, until some condition is met (i.e. loop - the same as conditional branch),
- executing a set of distant statements, after which the flow of control usually returns (subroutines, coroutines, and continuations),
- stopping the program, preventing any further execution (unconditional halt).

Interrupts and signals are low-level mechanisms that can alter the flow of control in a way similar to a subroutine, but usually occur as a response to some external stimulus or event (that can occur asynchronously), rather than execution of an 'in-line' control flow statement. Self-modifying code can also be used to affect control flow through its side effects, but usually does not involve an explicit control flow statement (an exception being the ALTER verb in COBOL).

At the level of machine or assembly language, control flow instructions usually work by altering the program counter. For some CPUs the only control flow instructions available are conditional or unconditional branch instructions (also called jumps).



## if Statement

- **if statement in Python is similar to that of other languages:**

- Contains a logical expression using which data is compared
- Decision is made based on the result of the comparison

- **Syntax if statement:**

```
if expression:
 statement(s)
```

- **Expression is evaluated first:**

- If true that is, if its value is nonzero
- Then statement(s) block are executed
- Otherwise next statement following statement(s) block is executed

- **Grouping statements:**

- Statements indented by the same number of character spaces:
  - Considered to be part of a single block of code.
- Python uses indentation as its method of grouping statements

Python provides the if statement to allow branching based on conditions. Multiple elif checks can also be performed followed by an optional else clause. The if statement can be used with any evaluation of truthiness.

```
>>> i = 3
>>> if i < 3:
... print 'less than 3'
... elif i < 5:
... print 'less than 5'
... else:
... print '5 or more'
...
less than 5
```

The code that is executed when a specific condition is met is defined in a "block." In Python, the block structure is signalled by changes in indentation. Each line of code in a certain block level must be indented equally and indented more than the surrounding scope. The standard (defined in PEP-8) is to use 4 spaces for each level of block indentation. Statements preceding blocks generally end with a colon (:).

Because there are no semi-colons or other end-of-line indicators in Python, breaking lines of code requires either a continuation character (\ as the last char) or for the break to occur inside an unfinished structure (such as open parentheses).

## if Example

```
var1 = 100
if var1:
 print("1 - Got a true expression value")
 print(var1)

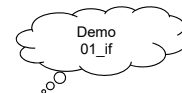
var2 = 0
if var2:
 print("2 - Got a true expression value")
 print var2
print("Good bye!")
```

- **Result:**

```
1 - Got a true expression value
100
Good bye!
```

- **In case if has only one line it may go on same line:**

```
if expression == 1: print("Value of expression is 1")
```



Conditional constructs are used to incorporate decision making into programs. The result of this decision making determines the sequence in which a program will execute instructions. You can control the flow of a program by using conditional constructs.

The if statement of Python is similar to that of other languages. The if statement contains a logical expression using which data is compared, and a decision is made based on the result of the comparison. The syntax of the if statement is:

```
if expression:
 statement(s)
```

Here if statement, condition is evaluated first. If condition is true that is, if its value is nonzero then the statement(s) block are executed. Otherwise, the next statement following the statement(s) block is executed. In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

```
from datetime import datetime
hour = datetime.now().hour
if hour < 12:
 time_of_day = 'morning'
else:
 time_of_day = 'afternoon'
print 'Good %s, world!' % time_of_day
```

## Multiple Cases

Python does not have a switch or case statement. Generally, multiple cases are handled with an if-elif-else structure and you can use as many elif's as you need.

## else Statement

- **Is combined with an if statement:**
  - Contains code that executes in case the `if` statement resolves to 0 or false
- **else statement is an optional statement:**
  - Only one `else` statement can follow the `if` statement
- **Syntax `if...else` statement:**

```
if expression:
 statement(s)
else:
 statement(s)
```



www.spiraltrain.nl

Control Flow and Operators

43

The `if` statement is used to check a condition and *if* the condition is true, we run a block of statements (called the *if-block*), *else* we process another block of statements (called the *else-block*). The *else* clause is optional.

```
number = 23
guess = int(input('Enter an integer: '))
if guess == number:
 print('Congratulations, you guessed it.') # New block
 print("(but you do not win prizes!)") # New block ends
elif guess < number:
 print('No, is little higher than that') # Another block
 # You can do whatever you want in a block ...
else:
 print 'No, it is a little lower than that'
 # you must have guess > number to reach here
print 'Done'

This last statement is always executed, after the if statement is
executed
```

In this program, we take guesses from the user and check if it is the number that we have. We set the variable `number` to any integer we want, say 23. Then, we take the user's guess using the `input()` function. Functions are just reusable pieces of programs.

## elif Statement

- **Allows checking of multiple expressions for truth value:**
  - Execute a block of code as soon as one of the conditions evaluates to true
- **elif statement is optional:**
  - Arbitrary number of **elif** statements may be following an **if**
- **Syntax of the if...elif statement:**

```
if expression1:
 statement(s)
elif expression2:
 statement(s)
elif expression3:
 statement(s)
else:
 statement(s)
```

- **No support for switch or case statements as in other languages**



www.spiraltrain.nl

Control Flow and Operators

44

After Python has finished executing the complete if statement along with the associated elif and else clauses, it moves on to the next statement in the block containing the if statement. In this case, it is the main block where execution of the program starts and the next statement is the print 'Done' statement. After this, Python sees the ends of the program and simply finishes up.

Although this is a very simple program, I have been pointing out a lot of things that you should notice even in this simple program. All these are pretty straightforward (and surprisingly simple for those of you from C/C++ backgrounds) and requires you to become aware of all these initially, but after that, you will become comfortable with it and it'll feel 'natural' to you.

### Note for C/C++ Programmers

There is no switch statement in Python. You can use an if...elif...else statement to do the same thing

#### The elif Statement:

The **elif** statement allows you to check multiple expressions for truth value and execute a block of code as soon as one of the conditions evaluates to true.

Like the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at most one statement, there can be an arbitrary number of **elif** statements following an **if**.

#### The Nested if...elif...else Construct:

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested if construct.

In a nested if construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

#### Single Statement Suites:

If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement:

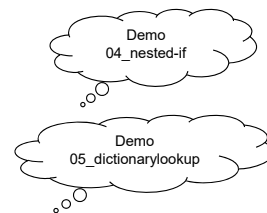
Here is an example of a one-line if clause:

```
if (expression == 1): print "Value of expression is 1"
```

## Nested if...elif...else

- `if...elif...else` **construct in other** `if...elif...else` **construct:**
  - Allows checking another condition after condition resolves to true
- **Syntax nested** `if...elif...else:`

```
if expression1:
 statement(s)
 if expression2:
 statement(s)
 elif expression3:
 statement(s)
 else:
 statement(s)
elif expression4:
 statement(s)
else:
 statement(s)
```



www.spiraltrain.nl

Control Flow and Operators

45

Python doesn't come with a typical switch-case statement. One simple substitute is using a string of if-else blocks, with each if condition being what would have been a matching case. Here is code in Python using if-else blocks:

```
if n == 0:
 print("You typed zero.\n")
elif n== 1 or n == 9 or n == 4:
 print("n is a perfect square\n")
elif n == 2:
 print("n is an even number\n")
elif n== 3 or n == 5 or n == 7:
 print("n is a prime number\n")
```

It certainly works and should be pretty easy to work, but it's not a very elegant solution. Especially if you have more than a handful of cases, you're going to have a very long string if-else cases. Since each of the if conditions must actually be checked, you might run into performance issues if this is a vital part of your code.

The Pythonic solution is to make use of Python's powerful dictionaries. Also known as associative arrays in some languages, Python's dictionaries allow a simple one-to-one matching of a key and a value. When used as part of a switch case statement, the keys are what would normally trigger the case blocks. The interesting part is that the values in the dictionaries refer to functions that contain the code that would normally be inside the case blocks. Here's the above code rewritten as a dictionary and functions:

## while Loop

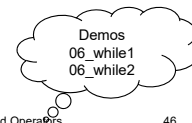
- **Control flow construct to repeat code:**
  - Repetition continues until conditional expression becomes false
  - condition must be a logical expression evaluating to true or false
- `while conditional expression:`  
`statement(s)`
- **while loop ends when conditional expression becomes false**
- **infinite loop when condition of a `while` never resolves to false:**
  - Infinite loop might be useful in client/server programming:  
Server runs continuously on an open port so that clients can communicate with it
- **If `while` clause has only one statement:**
  - Placement on the same line is allowed

```
while expression: statement
```

www.spiraltrain.nl

Control Flow and Operators

46



A loop is a construct that causes a section of a program to be repeated a certain number of times. The repetition continues while the condition set for the loop remains true. When the condition becomes false, the loop ends and the program control is passed to the statement following the loop. The **while** loop is one of the looping constructs available in Python. The **while** loop continues until the expression becomes false. The expression has to be a logical expression and must return either a *true* or a *false* value. The syntax of the while loop is:

`while expression: statement(s)` Here **expression** statement is evaluated first. If expression is *true* that is, then the statement(s) block is executed repeatedly until expression becomes *false*. Otherwise, the next statement following the statement(s) block is executed.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

```
number = 23
running = True
while running:
 guess = int(input('Enter an integer: '))
 if guess == number:
 print('Congratulations, you guessed it.')
 running = False # this causes while loop to stop
 elif guess < number:
 print('No, it is a little higher than that.')
 else:
 print('No, it is a little lower than that.')
else:
 print('The while loop is over.')
print 'Done'
```

## for Loop

- **Control flow construct to repeat code number of times:**

- Iterates over the items of an iterator (any sequence, such as a list or a string)

```
for iterating_var in sequence:
 statements(s)
```

- **for loop operation:**

- First item in the sequence is assigned to iterating variable `iterating_var`
- Next statements block is executed
- Each item in the sequence is assigned to `iterating_var`
- Statements(s) block is executed until the entire sequence is finished



[www.spiraltrain.nl](http://www.spiraltrain.nl)

Control Flow and Operators

47

The `for..in` statement is another looping statement which iterates over a sequence of objects i.e. go through each item in a sequence. What you need to know right now is that a sequence is just an ordered collection of items. Using the `for` statement:

```
for i in range(1, 5):
 print(i)
else:
 print('The for loop is over')
```

### Output:

```
$ python for.py
```

```
1
2
3
4
```

```
The for loop is over
```

In this program we are printing a sequence of numbers. We generate this sequence of numbers using the built-in `range` function.

What we do here is supply it two numbers and `range` returns a sequence of numbers starting from the first number and up to the second number. For example, `range(1,5)` gives the sequence `[1, 2, 3, 4]`. By default, `range` takes a step count of 1. If we supply a third number to `range`, then that becomes the step count. For example, `range(1,5,2)` gives `[1,3]`. Remember that the `range` extends up to the second number i.e. it does not include the second number.

## break and continue Statements

- **break statement:**
  - Terminates the current loop iteration
  - Resumes execution at next statement after the loop
- **continue statement returns control to the beginning of the loop:**
  - Remaining statements in the current iteration of the loop are skipped
  - Control moves back to the top of the loop
- **break and continue can be used in both while and for loops**
- **Example continue statement:**

```
for letter in 'Python':
 if letter == 'h':
 continue
 print('Current Letter:', letter)
print("Good bye!")
```
- **In result letter h is not printed**

www.spiraltrain.nl

Control Flow and Operators

48

In the program on the previous page, we repeatedly take the user's input and print the length of each input each time. We are providing a special condition to stop the program by checking if the user input is 'quit'. We stop the program by breaking out of the loop and reach the end of the program.

The length of the input string can be found out using the built-in len function.

Remember that the break statement can be used with the for loop as well.

The continue statement is used to tell Python to skip the rest of the statements in the current loop block and to continue to the next iteration of the loop.

```
while True:
 s = input('Enter something: ')
 if s == 'quit':
 break
 if len(s) < 3:
 continue
 print('Input is of sufficient length')
 # Do other kinds of processing here...
```

In this program, we accept input from the user, but we process them only if they are at least 3 characters long. So, we use the built-in len function to get the length and if the length is less than 3, we skip the rest of the statements in the block by using the continue statement. Otherwise, the rest of the statements in the loop are executed and we can do any kind of processing we want to do here.

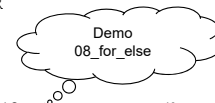
Note that the continue statement works with the for loop as well.



## Loop with else Combination

- **else statement can be associated with loop statements**
- **If else statement is used with a for loop:**
  - else statement is executed when the loop has exhausted iterating the list.
- **If else statement is used with a while loop:**
  - else statement is executed when the condition becomes false
- **In the example on the next slide:**
  - else statement is combined with of a for statement
  - Search is done for prime numbers from 10 through 20

```
for num in range(10,20): # to iterate between 10 to 20
 for i in range(2,num): # to iterate on the factors of the number
 if num%i == 0: # has factor, cannot be prime
 j=num/i
 print('%d equals %d * %d' % (num,i,j))
 break # to move to the next number, the #first FOR
 else: # else part of the loop
 print(num, 'is a prime number')
```



www.spiraltrain.nl

Control Flow and Operators

49

An addition to both for and while loops in Python that is not common to all languages is the availability of an else clause. The else block after a loop is executed in the case where no break occurs inside the loop.

The most common paradigm for using this clause occurs when evaluating a dataset for the occurrences of a certain condition and breaking as soon as it is found. Rather than setting a flag when found and checking after to see the result, the else clause simplifies the code.

In the following example, if a multiple of 5 is found, the break exits the for loop without executing the else clause.

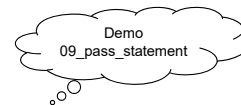
```
>>> for x in range(1,5):
... if x % 5 == 0:
... print '%d is a multiple of 5' % x
... break
... else:
... print 'No multiples of 5'
...
No multiples of 5
```

```
>>> for x in range(11,20):
... if x % 5 == 0:
... print '%d is a multiple of 5' % x
... break
... else:
... print 'No multiples of 5'
...
15 is a multiple of 5
```

## pass Statement

- **Used when a statement is required syntactically:**
  - But it is not desired that any command or code executes
- **pass statement is a null operation:**
  - Nothing happens when it executes
  - Also useful in places where code has not been written yet (e.g. in stubs )
- **pass statement is helpful when:**
  - A code block is created but it is no longer required
  - Statements inside the block may then be removed
  - Block may remain with pass statement so it doesn't interfere with other code parts

```
for letter in 'Python':
 if letter == 'h':
 pass
 print('This is pass block')
 print('Current Letter: ', letter)
print("Good bye!")
```



www.spiraltrain.nl

Control Flow and Operators

50

The **pass** statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

```
for letter in 'Python':
 if letter == 'h':
 pass
 print 'This is pass block'
 print 'Current Letter:', letter
print "Good bye!"
```

The preceding code does not execute any statement or code if the value of *letter* is 'h'. The *pass* statement is helpful when you have created a code block but it is no longer required.

You can then remove the statements inside the block but let the block remain with a *pass* statement so that it doesn't interfere with other parts of the code.

The *pass* statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
... pass # Busy-wait for keyboard interrupt
```

The body of a Python compound statement cannot be empty; it must always contain at least one statement. You can use a *pass* statement, which performs no action, as a placeholder when a statement is syntactically required but you have nothing to do.

## Python Arithmetic and Assignment Operators

| Operator | Arithmetic Operator Description                                       |
|----------|-----------------------------------------------------------------------|
| +        | Adds values on either side of the operator                            |
| -        | Subtracts right hand operand from left hand operand                   |
| *        | Multiplies values on either side of the operator                      |
| /        | Divide left hand operand by right hand operand with decimal result    |
| //       | Division with quotient result with digits after decimal point removed |
| %        | Divides left hand operand by right hand operand, returns remainder    |
| **       | Performs exponential (power) calculation on operators                 |

| Operator | Assignment Operator Description                                               |
|----------|-------------------------------------------------------------------------------|
| =        | Assigns values from right side operands to left side operand                  |
| +=       | Adds right operand to left operand and assigns result to left operand         |
| -=       | Subtracts right operand from left operand and assigns result to left operand  |
| *=       | Multiplies right operand with left operand and assigns result to left operand |
| /=       | Divides left operand by right operand and assigns result to left operand      |
| %=       | Takes modulus using two operands and assigns result to left operand           |

www.spiraltrain.nl

Control Flow and Operators

51

There are eight comparison operations in Python. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example,  $x < y \leq z$  is equivalent to  $x < y$  and  $y \leq z$ , except that  $y$  is evaluated only once (but in both cases  $z$  is not evaluated at all when  $x < y$  is found to be false).

Objects of different types, except different numeric types, never compare equal.

Furthermore, some types (for example, function objects) support only a degenerate notion of comparison where any two objects of that type are unequal. The  $<$ ,  $\leq$ ,  $>$  and  $\geq$  operators will raise a `TypeError` exception when comparing a complex number with another built-in numeric type, when the objects are of different types that cannot be compared, or in other cases where there is no defined ordering.

Non-identical instances of a class normally compare as non-equal unless the class defines the `__eq__()` method.

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines enough of the methods `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` (in general, `__lt__()` and `__eq__()` are sufficient, if you want the conventional meanings of the comparison operators).

The behavior of the `is` and `is not` operators cannot be customized; also they can be applied to any two objects and never raise an exception.

Two more operations with the same syntactic priority, `in` and `not in`, are supported only by sequence types.

Integers support additional operations that make sense only for bit-strings. Negative numbers are treated as their 2's complement value (this assumes a sufficiently large number of bits that no overflow occurs during the operation).

The priorities of the binary bitwise operations are all lower than the numeric operations and higher than the comparisons; the unary operation `~` has the same priority as the other unary numeric operations (`+` and `-`). Negative shift counts are illegal and cause a `ValueError` to be raised. A left shift by  $n$  bits is equivalent to multiplication by `pow(2, n)` without overflow check. A right shift by  $n$  bits is equivalent to division by `pow(2, n)` without overflow check.

## Python Comparison and Bitwise Operators

| Operator | Comparison Operator Description                                             |
|----------|-----------------------------------------------------------------------------|
| ==       | Checks if values of two operands are equal or not. If yes condition is true |
| !=       | Checks if values of two operands are equal or not. If not condition is true |
| <>       | Same as != operator                                                         |
| >        | Checks if value of left operand is greater than right operand               |
| <        | Checks if value of left operand is less than right operand                  |
| >=       | Checks if value of left operand is greater than or equal to right operand   |
| <=       | Checks if value of left operand is less than or equal to right operand      |

| Operator | Bitwise Operator Description                                                 |
|----------|------------------------------------------------------------------------------|
| &        | Copies a bit to the result if it exists in both operands                     |
|          | Copies a bit to the result if it exists in one or both of the operands       |
| ^        | Copies a bit to the result if it is set in one but not both operands         |
| ~        | Unary Binary Ones Complement Operator. Has effect of flipping the bits       |
| <<       | Left operands value is moved left by nr. of bits specified by right operand  |
| >>       | Left operands value is moved right by nr. of bits specified by right operand |

www.spiraltrain.nl

Control Flow and Operators

52

The operators in and not in test for membership. `x in s` evaluates to true if `x` is a member of `s`, and false otherwise. `x not in s` returns the negation of `x in s`. All built-in sequences and set types support this as well as dictionary, for which in tests whether a the dictionary has a given key. For container types such as list, tuple, set, frozenset, dict, or collections.deque, the expression `x in y` is equivalent to `any(x is e or x == e for e in y)`.

For the string and bytes types, `x in y` is true if and only if `x` is a substring of `y`. An equivalent test is `y.find(x) != -1`. Empty strings are always considered to be a substring of any other string, so `""` in `"abc"` will return True.

For user-defined classes which define the `__contains__()` method, `x in y` is true if and only if `y.__contains__(x)` is true.

For user-defined classes which do not define `__contains__()` but do define `__iter__()`, `x in y` is true if some value `z` with `x == z` is produced while iterating over `y`. If an exception is raised during the iteration, it is as if in raised that exception.

Lastly, the old-style iteration protocol is tried: if a class defines `__getitem__()`, `x in y` is true if and only if there is a non-negative integer index `i` such that `x == y[i]`, and all lower integer indices do not raise IndexError exception. (If any other exception is raised, it is as if in raised that exception).

The operator not in is defined to have the inverse true value of in.

The operators is and is not test for object identity: `x is y` is true if and only if `x` and `y` are the same object. `x is not y` yields the inverse truth value.

The power operator binds more tightly than unary operators on its left; it binds less tightly than unary operators on its right. The syntax is:

```
power ::= primary ["**" u_expr]
```

Thus, in an unparenthesized sequence of power and unary operators, the operators are evaluated from right to left (this does not constrain the evaluation order for the operands): `-1**2` results in `-1`.

## Logical, Membership and Identity Operators

| Operator | Logical Operator Description                                        |
|----------|---------------------------------------------------------------------|
| and      | If both operands are true then the condition becomes true           |
| or       | If any of the operands are non-zero then the condition becomes true |
| not      | Reverses logical state of operand. True condition will become false |

| Operator | Membership Operator Description                                               |
|----------|-------------------------------------------------------------------------------|
| in       | Evaluates to true if it finds variable in specified sequence, false otherwise |
| not in   | Evaluates to true if it does not find a variable in sequence, false otherwise |

| Operator | Identity Operator Description                                                |
|----------|------------------------------------------------------------------------------|
| is       | Evaluates to true if variables on both sides point to the same object        |
| is not   | Evaluates to true if variables on both sides do not point to the same object |



The power operator has the same semantics as the built-in `pow()` function, when called with two arguments: it yields its left argument raised to the power of its right argument. The numeric arguments are first converted to a common type, and the result is of that type. For int operands, the result has the same type as the operands unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns 100, but `10**-2` returns 0.01.

Raising 0.0 to a negative power results in a `ZeroDivisionError`. Raising a negative number to a fractional power results in a complex number. (In earlier versions it raised a `ValueError`.)

| Operation            | Result                               |
|----------------------|--------------------------------------|
| <code>x or y</code>  | if x is false, then y, else x        |
| <code>x and y</code> | if x is false, then x, else y        |
| <code>not x</code>   | if x is false, then True, else False |

**Booleans (bool)** These represent the truth values False and True. The two objects representing the values False and True are the only Boolean objects. The Boolean type is a subtype of the integer type, and Boolean values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that when converted to a string, the strings "False" or "True" are returned, respectively.

The rules for integer representation are intended to give the most meaningful interpretation of shift and mask operations involving negative integers.

## Function Syntax

- **First statement of a function can be an optional statement:**

- Documentation string of the function or `docstring`

```
def functionname(parameters):
 "function_docstring"
 function_suite
 return [expression]
```

- **By default:**

- Parameters have a positional behavior
- Pass them in the same order that they were defined

- **Example of a simple Python function:**

- Takes a string as input parameter and prints it on standard screen

```
def printme(s):
 "This prints a passed string into this function"
 print(s)
 return # redundant
```

www.spiraltrain.nl

Functions

54

A function can take parameters which are just values you supply to the function so that the function can *do* something utilising those values. These parameters are just like variables except that the values of these variables are defined when we call the function and are not assigned values within the function itself. Parameters are specified within the pair of parentheses in the function definition, separated by commas. When we call the function, we supply the values in the same way. Note the terminology used - the names given in the function definition are called *parameters* whereas the values you supply in the function call are called *arguments*.

```
def printMax(a, b):
 if a > b:
 print a, 'is maximum'
 else:
 print b, 'is maximum'

printMax(3, 4) # directly give literal values
x = 5
y = 7
printMax(x, y) # give variables as arguments
```

**Output:**

```
$ python func_param.py
4 is maximum
7 is maximum
```

Here, we define a function called `printMax` where we take two parameters called `a` and `b`. We find out the greater number using a simple `if..else` statement and then print the bigger number. In the first usage of `printMax`, we directly supply the numbers i.e. arguments. In the second usage, we call the function using variables. `printMax(x, y)` causes value of argument `x` to be assigned to parameter `a` and the value of argument `y` assigned to parameter `b`. The `printMax` function works the same in both the cases.

## Calling Functions

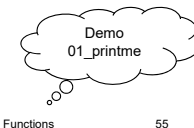
- **Execute functions by calling them:**
  - From another function or directly from the Python prompt
  - If functions specify parameters, it is required to pass them
- **Function must be defined before they can be called:**
  - Order of definition and call is important
- **Example of calling the previously defined `printme()` function:**

```
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
printme() # results in error
```

- **Result:**

```
I'm first call to user defined function!
Again second call to the same function
Traceback (most recent call last):
 File "01_printme.py", line 8, in <module>
 printme()
TypeError: printme() takes exactly 1 argument (0 given)
```

www.spiraltrain.nl



Functions

55

Formal parameters that are just identifiers indicate mandatory parameters. Each call to the function must supply a corresponding value (argument) for each mandatory parameter. In the comma-separated list of parameters, zero or more mandatory parameters may be followed by zero or more optional parameters, where each optional parameter has the syntax:

```
identifier=expression
```

The `def` statement evaluates each such expression and saves a reference to the expression's value, known as the default value for the parameter, among the attributes of the function object. When a function call does not supply an argument corresponding to an optional parameter, the call binds the parameter's identifier to its default value for that execution of the function. Note that each default value gets computed when the `def` statement evaluates, not when the resulting function gets called. In particular, this means that the same object, the default value, gets bound to the optional parameter whenever the caller does not supply a corresponding argument. This can be tricky when the default value is a mutable object and the function body alters the parameter. For example:

```
def f(x, y=[]):
 y.append(x)
 return y

print f(23) # prints: [23]
print f(42) # prints: [23, 42]
```

## Pass by sharing

- **Python passes function parameters by value (i.e. by object):**

- Parameter in function is bound to same object as referred to by variable used in call
- Change to object via parameter in function is seen by variable in function's caller
- So changes to mutable objects made by function will also affect variable in call

```
def changeme(mylist):
 mylist.append([1,2,3])
 print("Values inside the function: ", mylist)
 return
```

- **Call to changeme function:**

```
mylistouter = [10,20,30]
changeme(mylistouter)
print("Values outside the function: ", mylistouter)
```

- **Reference of passed object is used to append values:**

```
Values inside function: [10, 20, 30, [1, 2, 3]]
Values outside function: [10, 20, 30, [1, 2, 3]]
```



www.spiraltrain.nl

Functions

56

The second print statement prints [23, 42] because the first call to f altered the default value of y, originally an empty list [], by appending 23 to it. If you want y to be bound to a new empty list object each time f is called with a single argument, use the following style instead:

```
def f(x, y=None):
 if y is None: y = []
 y.append(x)
 return y
print f(23) # prints: [23]
print f(42) # prints: [42]
```

At the end of the parameters, you may optionally use either or both of the special forms `*identifier1` and `**identifier2`. If both forms are present, the form with two asterisks must be last. `*identifier1` specifies that any call to the function may supply any number of extra positional arguments, while `**identifier2` specifies that any call to the function may supply any number of extra named arguments (positional and named arguments are covered in "Calling Functions" on page 73). Every call to the function binds `identifier1` to a tuple whose items are the extra positional arguments (or the empty tuple, if there are none). Similarly, `identifier2` gets bound to a dictionary whose items are the names and values of the extra named arguments (or the empty dictionary, if there are none). Here's a function that accepts any number of positional arguments and returns their sum:

```
def sum_args(*numbers):
 return sum(numbers)
print sum_args(23, 42) # prints: 65
```

The number of parameters of a function, together with the parameters' names, the number of mandatory parameters, and the information on whether (at the end of the parameters) either or both of the single- and double-asterisk special forms are present, collectively form a specification known as the function's signature. A function's signature defines the ways in which you can call the function.



## Overwriting parameters

- **Parameter may be overwritten in the function:**

```
def changeme(mylist):
 mylist = [1,2,3,4] # This would assign new reference
 print("Values inside the function: ", mylist)
 return
```

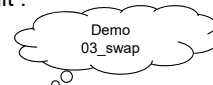
- **Call changeme function:**

```
mylistouter = [10,20,30]
changeme(mylistouter)
print("Values outside the function: ", mylistouter)
```

- **Parameter `mylist` is changed in the function `changeme`:**

- Now changing `mylist` within the function does not affect `mylistouter`
- Function does nothing and finally this would produce following result :

```
Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]
```



- **Changing immutable references always results in new objects:**
  - Is true for Strings and Numbers

www.spiraltrain.nl

Functions

57

## C is pass-by-value

Straightforward. You can simulate pass-by-reference with pointers. Not much else to say here.

## Java is pass-by-value

Primitive Types (non-object built-in types) are simply passed by value. Passing Object References feels like pass-by-reference, but it isn't. What you are really doing is passing references-to-objects by value.

## OK, so what about Python?

Python passes references-to-objects by value (like Java), and everything in Python is an object. This sounds simple, but then you will notice that some data types seem to exhibit pass-by-value characteristics, while others seem to act like pass-by-reference... what's the deal?

It is important to understand mutable and immutable objects. Some objects, like strings, tuples, and numbers, are immutable. Altering them inside a function/method will create a new instance and the original instance outside the function/method is not changed. Other objects, like lists and dictionaries are mutable, which means you can change the object in-place. Therefore, altering an object inside a function/method will also change the original object outside.

A function attribute is the documentation string, also known as the docstring. You may use or rebind a function's docstring attribute as either `func_doc` or `__doc__`. If the first statement in the function body is a string literal, the compiler binds that string as the function's docstring attribute. A similar rule applies to classes and modules. Docstrings most often span multiple physical lines, so you normally specify them in triple-quoted string literal form. For example:

```
def sum_args(*numbers):
 '''Accept arbitrary numerical arguments and return sum.
 Arguments are zero or more numbers. Result is sum.'''
 return sum(numbers)
```

## Keyword Arguments

- **Function definition:**

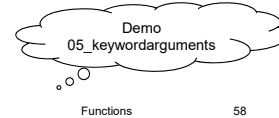
```
def printinfo(name, age):
 "This prints name and age passed into this function"
 print("Name:", name)
 print("Age", age)
 return
```

- **Function call with keyword arguments:**

```
printinfo(age=50, name="miki")
```

- **Result:**

```
Name: miki
Age 50
```



www.spiraltrain.nl

If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called keyword arguments - we use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function.

There are two advantages - one, using the function is easier since we do not need to worry about the order of the arguments. Two, we can give values to only those parameters to which we want to, provided that the other parameters have default argument values.

```
def func(a, b=5, c=10):
 print('a is', a, 'and b is', b, 'and c is', c)
func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

### Output:

```
$ python func_key.py
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

## Default Arguments

- **Function definition:**

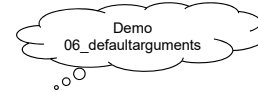
```
def printinfo(name, age = 35):
 "This prints name and age passed into this function"
 print("Name: ", name)
 print("Age ", age)
 return
```

- **Functions calls with and without default arguments:**

```
printinfo(age=50, name="miki")
printinfo(name="miki")
```

- **Results:**

```
Name: miki
Age 50
Name: miki
Age 35
```



www.spiraltrain.nl

Functions

59

The function named `func` has one parameter without a default argument value, followed by two parameters with default argument values.

In the first usage, `func(3, 7)`, the parameter `a` gets the value 3, the parameter `b` gets the value 7 and `c` gets the default value of 10.

In the second usage `func(25, c=24)`, the variable `a` gets the value of 25 due to the position of the argument. Then, the parameter `c` gets the value of 24 due to naming i.e. keyword arguments. The variable `b` gets the default value of 5.

In the third usage `func(c=50, a=100)`, we use keyword arguments for all specified values. Notice that we are specifying the value for parameter `c` before that for `a` even though `a` is defined before `c` in the function definition

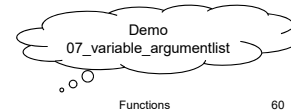
Documentation strings should be part of any Python code you write. They play a role similar to that of comments in any programming language, but their applicability is wider, since they remain available at runtime. Development environments and tools can use docstrings from function, class, and module objects to remind the programmer how to use those objects. The `doctest` module makes it easy to check that sample code present in docstrings is accurate and correct.

To make your docstrings as useful as possible, you should respect a few simple conventions. The first line of a docstring should be a concise summary of the function's purpose, starting with an uppercase letter and ending with a period. It should not mention the function's name, unless the name happens to be a natural-language word that comes naturally as part of a good, concise summary of the function's operation. If the docstring is multiline, the second line should be empty, and the following lines should form one or more paragraphs, separated by empty lines, describing the function's parameters, preconditions, return value, and side effects (if any). Further explanations, bibliographical references, and usage examples (which you should check with `doctest`) can optionally follow toward the end of the docstring.

## Variable-length arguments

```
def functionname([args,] *var_args_tuple, **var_args_keywords):
 "function_docstring"
 function_suite
 return [expression]
```

- `*var_args_tuple`:
  - Asterisk (\*) indicates variable holding values of all non-keyword variable arguments
  - Tuple remains empty if no additional arguments are specified during function call
- `**var_args_keywords`:
  - Double asterisk (\*\*) indicates a variable-length argument list with keywords
  - Dictionary remains empty if no keyword arguments are passed



www.spiraltrain.nl

Functions

60

Sometimes you might want to define a function that can take *any* number of parameters, this can be achieved by using the stars:

```
def total(initial=5, *numbers, **keywords):
 count = initial
 for number in numbers:
 count += number
 for key in keywords:
 count += keywords[key]
 return count

print(total(10, 1, 2, 3, vegetables=50, fruits=100))
```

### Output:

```
$ python total.py
166
```

When we declare a starred parameter such as `*param`, then all the positional arguments from that point till the end are collected as a tuple called 'param'.

Similarly, when we declare a double-starred parameter such as `**param`, then all the keyword arguments from that point till the end are collected as a dictionary called 'param'.

### Keyword-only Parameters:

If we want to specify certain keyword parameters to be available as keyword-only and not as positional arguments, they can be declared after a starred parameter.

Declaring parameters after a starred parameter results in keyword-only arguments. If these arguments are not supplied a default value, then calls to the function will raise an error if the keyword argument is not supplied.

## Anonymous Functions

- `lambda` **keyword is used to create small anonymous functions:**
  - Not declared in the standard manner by using the `def` keyword
- **Lambda form can take any number of arguments:**
  - Return just one value in the form of an expression
  - Cannot contain commands or multiple expressions
- **Characteristics of lambda functions:**
  - Have their own local namespace
  - Can only access variables in parameter list and in global namespace
  - A single expression as body

Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct called "lambda". This is not exactly the same as lambda in functional programming languages, but it is a very powerful concept that's well integrated into Python and is often used in conjunction with typical functional concepts like `filter()`, `map()` and `reduce()`.

This piece of code shows the difference between a normal function definition ("`f`") and a lambda function ("`g`"):

```
>>> def f (x): return x**2
...
>>> print(f(8))
64
>>>
>>> g = lambda x: x**2
>>>
>>> print(g(8))
64
```

As you can see, `f()` and `g()` do exactly the same and can be used in the same ways. Note that the lambda definition does not include a "return" statement -- it always contains an expression which is returned. Also note that you can put a lambda definition anywhere a function is expected, and you don't have to assign it to a variable at all.

## Syntax Lambda Functions

- **Lambda functions contain only a single statement:**

```
lambda [arg1 [,arg2,.....argn]]:expression
```

- **Function definition:**

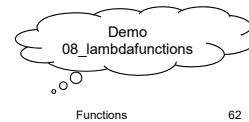
```
sum = lambda arg1, arg2: arg1 + arg2
```

- **Function call:**

```
print("Value of total: ", sum(10, 20))
print("Value of total: ", sum(20, 20))
```

- **Result:**

```
Value of total: 30
Value of total: 40
```



www.spiraltrain.nl

The following code fragments demonstrate the use of lambda functions. Note that you should have Python 2.2 or newer, in order to have support for nested scopes (in older versions you have to pass "n" through a default argument to make this example work).

```
>>> def make_incrementor (n): return lambda x: x + n
>>>
>>> f = make_incrementor(2)
>>> g = make_incrementor(6)
>>>
>>> print(f(42), g(42))
44 48
>>>
>>> print(make_incrementor(22)(33))
55
```

The above code defines a function "make\_incrementor" that creates an anonymous function on the fly and returns it. The returned function increments its argument by the value that was specified when it was created.

You can now create multiple different incrementor functions and assign them to variables, then use them independent from each other. As the last statement demonstrates, you don't even have to assign the function anywhere -- you can just use it instantly and forget it when it's not needed anymore.

## return Statement

- `return [expression]` **exits a function:**
  - Optionally passing back an expression to the caller
  - `return` statement with no arguments is the same as `return None`

- **Function definition:**

```
def sum(arg1, arg2):
 # Add both the parameters and return them."
 total = arg1 + arg2
 print("Inside the function: ", total)
 return total
```

- **Function call:**

```
total = sum(10, 20);
print("Outside the function: ", total)
```

- **Result:**

```
Inside the function: 30
Outside the function: 30
```

[www.spiraltrain.nl](http://www.spiraltrain.nl)

Functions

63

The `return` statement is used to return from a function i.e. break out of the function. We can optionally return a value from the function as well.

```
def maximum(x, y):
 if x > y:
 return x
 elif x == y:
 return 'The numbers are equal'
 else:
 return y

print(maximum(2, 3))
```

The `maximum` function returns the maximum of the parameters, in this case the numbers supplied to the function. It uses a simple `if..else` statement to find the greater value and then returns that value.

Note that a `return` statement without a value is equivalent to `return None`. `None` is a special type in Python that represents nothingness. For example, it is used to indicate that a variable has no value if it has a value of `None`.

Every function implicitly contains a `return None` statement at the end unless you have written your own `return` statement. You can see this by running `print(someFunction())` where the function `someFunction` does not use the `return` statement such as:

```
def someFunction():
 pass
```

The `pass` statement is used in Python to indicate an empty block of statements.

There is a built-in function called `max` that already implements the 'find maximum' functionality, so use this built-in function whenever possible.

## Scope of Variables

- **Accessibility of variable depends on where it is declared:**
  - Scope determines where in program you can access a particular identifier
- **There are two basic scopes of variables in Python:**
  - Global variables
  - Local variables
- **Global versus Local variables:**
  - Variables defined inside a function body have a local scope
  - Variables defined outside have a global scope
- **Local variables:**
  - Can be accessed only inside the function in which they are declared
  - When you call a function, the variables declared inside it are brought into scope
- **Global variables:**
  - Can be accessed throughout the program body by all functions
  - Refer to global instead of a local variable inside function with `global` keyword

www.spiraltrain.nl

Functions

64

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function i.e. variable names are local to the function. This is called the scope of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

```
def func(x):
 print('x is', x)
 x = 2
 print('Changed local x to', x)

x = 50
func(x)
print('x is still', x)
```

### Output:

```
$ python func_local.py
x is 50
Changed local x to 2
x is still 50
```

In the function, the first time that we use the value of the name `x`, Python uses the value of the parameter declared in the function.

Next, we assign the value 2 to `x`. The name `x` is local to our function. So, when we change the value of `x` in the function, the `x` defined in the main block remains unaffected.

In the last print statement, we confirm that the value of `x` in the main block is actually unaffected.



## Example Variable Scope

```
total = 0 # This is a global variable
```

- **Function definition:**

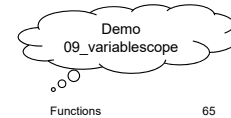
```
def sum(arg1, arg2):
 # Add both the parameters and return them."
 total = arg1 + arg2; # Here total is local variable.
 print("Inside the function local total: ", total)
 return total;
```

- **Function call:**

```
sum(10, 20)
print("Outside the function global total: ", total)
```

- **Result:**

```
Inside the function local total: 30
Outside the function global total: 0
```



[www.spiraltrain.nl](http://www.spiraltrain.nl)

Functions

65

If you want to assign a value to a name defined outside the function, then you have to tell Python that the name is not local, but it is *global*. We do this using the global statement. It is impossible to assign a value to a variable defined outside a function without the global statement.

You can use the values of such variables defined outside the function (assuming there is no variable with the same name within the function). However, this is not encouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable's definition is. Using the global statement makes it amply clear that the variable is defined in an outer block.

```
def func():
 global x
 print('x is', x)
 x = 2
 print('Changed global x to', x)
x = 50
func()
print('Value of x is', x)
```

The global statement is used to declare that x is a global variable - hence, when we assign a value to x inside the function, that change is reflected when we use the value of x in the main block.

You can specify more than one global variable using the same global statement. For example, global x, y, z.

## Modules

- **Used to organize number of Python definitions in single file:**
  - Used to logically organize your Python code
  - Can be bind and referenced
- **Module is a file consisting of Python code:**
  - Defines functions, classes, and variables
  - May also include runnable code
  - Grouping related code into a module makes code easier to understand and use
- **Example:**
  - Python code for a module named test normally resides in a file named `test.py`
- `test.py`:
 

```
def print_func(par):
 print("Hello: ", par)
```

www.spiraltrain.nl

Modules

66

You have seen how you can reuse code in your program by defining functions once. What if you wanted to reuse a number of functions in other programs that you write? As you might have guessed, the answer is modules.

There are various methods of writing modules, but the simplest way is to create a file with a `.py` extension that contains functions and variables.

Another method is to write the modules in the native language in which the Python interpreter itself was written. For example, you can write modules in the C programming language and when compiled, they can be used from your Python code when using the standard Python interpreter.

A module can be *imported* by another program to make use of its functionality. This is how we can use the Python standard library as well. First, we will see how to use the standard library modules.

```
import sys
print('The command line arguments are:')
for i in sys.argv:
 print(i)
print('\n\nThe PYTHONPATH is', sys.path, '\n')
```

First, we *import* the `sys` module using the `import` statement. Basically, this translates to us telling Python that we want to use this module. The `sys` module contains functionality related to the Python interpreter and its environment i.e. the system.

When Python executes the `import sys` statement, it looks for the `sys` module. In this case, it is one of the built-in modules, and hence Python knows where to find it.

If it was not a compiled module i.e. a module written in Python, then the Python interpreter will search for it in the directories listed in its `sys.path` variable. If the module is found, then the statements in the body of that module are run and the module is made *available* for you to use. Note that the initialization is done only the *first* time that we import a module.

## import Statement

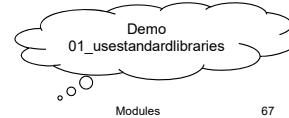
- **Any Python source file can be used as a module:**
  - By executing an `import` statement in some other Python source file
- **`import` has the following syntax:**

```
import module1[, module2[,... moduleN]
```
- **On encountering an `import` statement:**
  - Module is imported if the module is present in the search path
  - Search path is list of directories that interpreter searches for modules to import
  - A module is loaded only once, regardless of the number of times it is imported
- **Example:**
  - To import module `test.py`, put the following command at the top of the script:
- **Name module can be changed with:**

```
import test # Now you can call defined function
test.print_func('pipo') => Hello: pipo
```

```
import test as t
t.print_func('pipo') => Hello: pipo
```

[www.spiraltrain.nl](http://www.spiraltrain.nl)



The `argv` variable in the `sys` module is accessed using the dotted notation i.e. `sys.argv`. It clearly indicates that this name is part of the `sys` module. Another advantage of this approach is that the name does not clash with any `argv` variable used in your program.

The `sys.argv` variable is a *list* of strings. Specifically, the `sys.argv` contains the list of *command line arguments* i.e. the arguments passed to your program using the command line.

If you are using an IDE to write and run these programs, look for a way to specify command line arguments to the program in the menus.

Here, when we execute `python using_sys.py` we are arguments, we run the module `using_sys.py` with the `python` command and the other things that follow are arguments passed to the program. Python stores the command line arguments in the `sys.argv` variable for us to use.

Remember, the name of the script running is always the first argument in the `sys.argv` list. So, in this case we will have `'using_sys.py'` as `sys.argv[0]`, `'we'` as `sys.argv[1]`, `'are'` as `sys.argv[2]` and `'arguments'` as `sys.argv[3]`. Notice that Python starts counting from 0 and not 1.

The `sys.path` contains the list of directory names where modules are imported from. Observe that the first string in `sys.path` is empty - this empty string indicates that the current directory is also part of the `sys.path` which is same as the `PYTHONPATH` environment variable. This means that you can directly import modules located in the current directory. Otherwise, you will have to place your module in one of the directories listed in `sys.path`.

Note that the current directory is the directory from which the program is launched. Run `import os; print(os.getcwd())` to find out the current directory of your program.

## from...import Statement

- Imports specific attributes from module into current namespace:

```
from modname import name1[, name2[, ... nameN]]
```

- Function `sqrt` from module `math` is imported with:

```
from math import sqrt
```

- Statement does not import the entire module `math`:

- Only item `sqrt` from module `math` in global symbol table of importing module

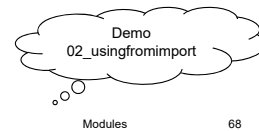
- Now the function `sqrt` can be used without prefix:

```
print(sqrt(9)) => 3.0
```

- Also possible is to import all names from a module with:

```
from modname import *
```

- Statement should be used sparingly



www.spiraltrain.nl

Modules

68

If you want to directly import the `argv` variable into your program (to avoid typing the `sys.` everytime for it), then you can use the `from sys import argv` statement. If you want to import all the names used in the `sys` module, then you can use the `from sys import *` statement. This works for any module. In general, you should avoid using this statement and use the `import` statement instead since your program will avoid name clashes and will be more readable. Example:-

```
from math import *
n = int(input("Enter range:- "))
p = [2, 3]
count = 2
a = 5
while(count < n):
 b = 0
 for i in range(2, a):
 if(i <= sqrt(a)):
 if(a % i == 0):
 #print a, "is not a prime"
 b = 1
 else:
 pass
 if(b != 1):
 #print a, "is a prime"
 p = p + [a]
 count = count + 1
 a = a + 2
print p
```

## Locating Modules

- **On import Python interpreter searches for module as follows:**
  - Current directory
  - Each directory in the shell variable `PYTHONPATH`
  - Default path which is OS dependent
- **Search path is stored in system module `sys` as `sys.path` variable**
- **`PYTHONPATH` variable:**
  - Environment variable, consisting of a list of directories
  - Syntax of `PYTHONPATH` is same as that of the shell variable `PATH`
- **Typical `PYTHONPATH` from Windows system:**

```
set PYTHONPATH=c:\python20\lib;
```
- **Typical `PYTHONPATH` from UNIX system:**

```
set PYTHONPATH=/usr/local/lib/python
```

[www.spiraltrain.nl](http://www.spiraltrain.nl)

Modules

69

For modules to be available for use, the Python interpreter must be able to locate the module file. Python has a set of directories in which it looks for module files. This set of directories is called the search path, and is analogous to the `PATH` environment variable used by an operating system to locate an executable file.

Python's search path is built from a number of sources:

`PYTHONHOME` is used to define directories that are part of the Python installation. If this environment variable is not defined, then a standard directory structure is used. For Windows, the standard location is based on the directory into which Python is installed. For most Linux environments, Python is installed under `/usr/local`, and the libraries can be found there. For Mac OS, the home directory is under `/Library/Frameworks/Python.framework`.

`PYTHONPATH` is used to add directories to the path. This environment variable is formatted like the OS `PATH` variable, with a series of filenames separated by `:`s (or `;`s for Windows).

**Script Directory.** If you run a Python script, that script's directory is placed first on the search path so that locally-defined modules will be used instead of built-in modules of the same name.

The site module's locations are also added. (This can be disabled by starting Python with the `-S` option.) The site module will use the `PYTHONHOME` location(s) to create up to four additional directories. Generally, the most interesting one is the site-packages directory. This directory is a handy place to put additional modules you've downloaded. Additionally, this directory can contain `.PTH` files. The site module reads `.PTH` files and puts the named directories onto the search path.

The search path is defined by the `path` variable in the `sys` module. If we import `sys`, we can display `sys.path`. This is very handy for debugging. When debugging shell scripts, it can help to run `'python -c 'import sys; print sys.path'` just to see parts of the Python environment settings.

## Creating and Using Modules

- **Every Python program is already a module:**

- Modules must have `.py` extension

```
Filename: mymodule.py
def sayhi():
 print('Hi, this is mymodule speaking.')
__version__ = '0.1'
End of mymodule.py
```

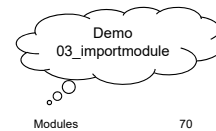
- **Module can be imported by other module if on `sys.path`:**

```
import mymodule
mymodule.sayhi()
print('Version', mymodule.__version__)
```

- **Alternatively `from .. import` can be used:**

```
from mymodule import sayhi, __version__
sayhi()
print('Version', __version__)
```

[www.spiraltrain.nl](http://www.spiraltrain.nl)



70

Creating your own modules is easy, you've been doing it all along! This is because every Python program is also a module. You just have to make sure it has a `.py` extension.

A sample module and usage is on the slide. As you can see, there is nothing particularly special about it compared to our usual Python program. We will next see how to use this module in our other Python programs.

Remember that the module should be placed either in the same directory as the program from which we import it, or in one of the directories listed in `sys.path`.

Notice that we use the same dotted notation to access members of the module. Python makes good reuse of the same notation to give the distinctive 'Pythonic' feel to it so that we don't have to keep learning new ways to do things.

Notice that if there was already a `__version__` name declared in the module that imports `mymodule`, there would be a clash. This is also likely because it is common practice for each module to declare its version number using this name. Hence, it is always recommended to prefer the import statement even though it might make your program a little longer.

You could also use:

```
from mymodule import *
```

This will import all public names such as `sayhi` but would not import `__version__` because it starts with double underscores.

Zen of Python:

One of Python's guiding principles is that "Explicit is better than Implicit".

## dir Function

- Returns sorted list of strings with names defined by a module
- List contains the names defined in a module:
  - Modules, Variables, Functions
- **Example** `import built-in module math:`

```
import math
content = dir(math)
print(content)
```

- **Result:**

```
['__doc__', '__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', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'trunc']
```

- `__name__` is module's name
- `__package__` is package name of module

www.spiraltrain.nl



You can use the built-in `dir` function to list the identifiers that an object defines. For example, for a module, the identifiers include the functions, classes and variables defined in that module.

When you supply a module name to the `dir()` function, it returns the list of the names defined in that module. When no argument is applied to it, it returns the list of names defined in the current module.

```
$ python
>>> import sys # get list of attributes, in this case, for the sys
module
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__',
'tderr__', '__stdin__', '__stdout__', '_clear_type_cache',
'_current_frames', '_getframe', 'api_version', 'argv',
', 'meta_path', 'modules', 'path', 'path_hooks',
m', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setprofile',
', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',
fo', 'warnoptions', 'winver']
>>> dir() # get list of attributes for current module
['__builtins__', '__doc__', '__name__', '__package__', 'sys']
```

First, we see the usage of `dir` on the imported `sys` module. We can see the huge list of attributes that it contains.

Next, we use the `dir` function without passing parameters to it. By default, it returns the list of attributes for the current module. Notice that the list of imported modules is also part of this list.

## Packages in Python

- **Package is way to organize number of modules together as unit:**
  - Has a hierarchical file directory structure
  - Consists of modules and subpackages and sub-subpackages, and so on
- **Multiple functions in files and different Python classes:**
  - Create packages out of those classes
- **Typically modules of a package are placed in a directory:**
  - Consider for example `Canon.py`, `Leika.py` and `Nikon.py` in `cameras` directory
- `Canon.py` could define a simple function like:
 

```
def CanonInfo():
 print("Canon camera")
```
- **Other two files could define similar but different functions:**
  - `def Leika()` and `def Nikon()`
- **Package can have `__init__.py` in its directory (must < python3.3):**
  - Makes functions available when package is imported

www.spiraltrain.nl

Modules

72

A package is a collection of Python modules. Packages allow us to structure a collection of modules. A package is a directory that contains modules. Having a directory of modules allows us to have modules contained within other modules. This allows us to use qualified module names, clarifying the organization of our software. We can, for example, have several simulations of casino games. Rather than pile all of our various files into a single, flat directory, we might have the following kind of directory structure. (This isn't technically complete, it needs a few additional files.)

```
casino/
 craps/
 dice.py
 game.py
 player.py
 roulette/
 wheel.py
 game.py
 player.py
 blackjack/
 cards.py
 game.py
 player.py
 strategy/
 basic.py
 martingale.py
 bet1326.py
 cancellation.py
```



## Explicit Import of Package Modules

- modules of a package can be imported explicitly:

```
import cameras.Canon
import cameras.Leika
import cameras.Nikon
```

- Functions in package are called with full name:

```
cameras.Nikon.NikonInfo()
cameras.Canon.CanonInfo()
cameras.Leika.LeikaInfo()
```

- Full name consists of:

- Package name (directory), module name, function name



www.spiraltrain.nl

Modules

73

Given this directory structure, our overall simulation might include statements like the following.

```
import craps.game, craps.player
import strategy.basic as betting
class MyPlayer(craps.player.Player):
 def __init__(self, stake, turns):
 betting.initialize(self)
```

We imported the game and player modules from the craps package. We imported the basic module from the strategy package. We defined a new player based on a class named Player in the craps.player package.

We have a number of alternative betting strategies, all collected under the strategy package. When we import a particular betting strategy, we name the module betting. We can then change to a different betting strategy by changing the **import** statement.

There are two reasons for using a package of modules.

There are a lot of modules, and the package structure clarifies the relationships among the modules. If we have several modules related to the game of craps, we might have the urge to create a craps\_game.py module and a craps\_player.py module. As soon as we start structuring the module names to show a relationship, we can use a package instead.

There are alternative implementations, and the package contains polymorphic modules. In this case, we will often use an import *package.alternative* as *interface* kind of **import** statement. This is often used for interfaces and drivers to isolate the interface details and provide a uniform API to the rest of the Python application.

It is possible to go overboard in package structuring. The general rule is to keep the package structure relatively flat. Having only one module at the bottom of deeply-nested packages isn't really very informative or helpful.

## Implicit Import of Package Modules

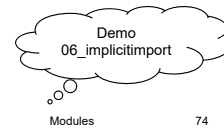
- **Following lines can be added to `__init__.py` in package:**  

```
from cameras.Canon import CanonInfo
from cameras.Leika import LeikaInfo
```
- **Now modules can be imported by importing the package:**  

```
import cameras
```
- **Import `cameras` package and call methods:**  

```
cameras.CanonInfo()
cameras.LeikaInfo()
```
- **Also from `.. import` can be used:**  

```
from cameras import NikonInfo, CanonInfo, LeikaInfo
NikonInfo()
CanonInfo()
LeikaInfo()
```



[www.spiraltrain.nl](http://www.spiraltrain.nl)

Modules

74

`__init__`. Valid Python names are composed of letters, digits and underscores. See the section called “Variables” for more information.

The `__init__` module is often an empty file, `__init__.py` in the package directory. Nothing else is required to make a directory into a package. The `__init__.py` file, however, is essential. Without it, you'll get an `ImportError`.

For example, consider a number of modules related to the definition of cards. We might have the following files.

```
cards/
 __init__.py
 standard.py
 blackjack.py
 poker.py
```

The `cards.standard` module would provide the base definition of card as an object with suit and rank. The `cards.blackjack` module would provide the subclasses of cards that we looked at in the section called “Blackjack Hands”. The `cards.poker` module would provide the subclasses of cards that we looked at in the section called “Poker Hands”.

The `cards.blackjack` module and the `cards.poker` module should both import the `cards.standard` module to get the base definition for the `Card` and `Deck` classes.

The `__init__` module. The `__init__` module is the “initialization” module in a package. It is processed the first time that a package name is encountered in an import statement. In effect, it initializes Python's understanding of the package. Since it is always loaded, it is also effectively the default module in a package. There are a number of consequences to this. We can import the package, without naming a specific module. In this case we've imported just the initialization module, `__init__`. If this is part of our design, we'll put some kind of default or top-level definitions in the `__init__` module.

We can import a specific module from the package. In this case, we also import the initialization module along with the requested module. In this case, the `__init__` module can provide additional definitions; or it can simply be empty.

## globals and locals Functions

- **Used to return the names in the global and local namespaces**
- `locals()` **called from within a function:**
  - Return all the names that can be accessed locally from that function
- `globals()` **called from within a function:**
  - Return all the names that can be accessed globally from that function
- **Return type of both these functions is dictionary:**
  - Names can be extracted using the `keys()` function

In our cards example, above, we would do well to make the `__init__` module define the basic Card and Deck classes. If we import the package, `cards`, we get the default module, `__init__`, which gives us `cards.Card` and `cards.Deck`. If we import a specific module like `cards.blackjack`, we get the `__init__` module plus the named module within the package.

### Package Use

If the `__init__` module in a package is empty, the package is little more than a collection of module files. In this case, we don't generally make direct use of a package. We merely mention it in an import statement: `import cards.poker`.

On other hand, if the `__init__` module has some definitions, we can import the package itself. Importing a package just imports the `__init__` module from the package directory. In this case, we mention the package in an import statement: `import cards`.

Even if the `__init__` module has some definitions in it, we can always import a specific module from within the package. Indeed, it is possible for the `__init__` module in a package is to do things like adjust the search path prior to locating individual module files.

## reload Function

- **imports a previously imported module again:**
  - When module is imported, code in the top-level portion is executed only once
- **To reexecute top-level code in a module:**
  - Use the `reload()` function from `importlib` module
  - In previous version `imp` module
- **Syntax:**

```
reload(module_name)
```
- `module_name`:
  - Name of the module you want to reload
  - Not the string containing the module name
- **To re-load hello module:**

```
reload(hello)
```

Reload a previously imported module. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the module argument).

When `reload(module)` is executed:

Python modules' code is recompiled and the module-level code reexecuted, defining a new set of objects which are bound to names in the module's dictionary. The `init` function of extension modules is not called a second time.

As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.

The names in the module namespace are updated to point to any new or changed objects.

Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

If a module is syntactically correct but its initialization fails, the first import statement for it does not bind its name locally, but does store a (partially initialized) module object in `sys.modules`. To reload the module you must first import it again (this will bind the name to the partially initialized module object) before you can `reload()` it.

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains.

## Namespaces and Scoping

- **Namespace is a dictionary containing:**
  - Variable names (keys) and their corresponding objects (values)
- **Variables can be in local namespace or global namespace:**
  - Variables declared outside functions are in the global namespace
  - Variables declared in function are in the local namespace of that function
  - Class methods follow the same scoping rule as ordinary functions
- **If local and global variable have the same name:**
  - Local variable shadows the global variable
- **Python assumes whether variables are local or global:**
  - Any variable assigned a value in a function is local
  - Use the `global` statement to assign a value to a global variable within a function
- `global VarName` **tells Python that `VarName` is a global variable:**
  - Python stops searching the local namespace for the variable

www.spiraltrain.nl

Modules

77

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as `abs()`, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [\[1\]](#)

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

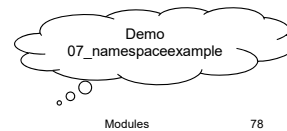
Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`)

## Example Namespaces

- **Define variable `Money` in the global namespace:**
  - Assign `Money` a value within the function `AddMoney`
  - Python assumes `Money` is a local variable
- **If value of local variable `Money` is accessed before setting it:**
  - `UnboundLocalError` is the result
  - Uncommenting the global statement fixes the problem

```
Money = 2000
def AddMoney():
 # Uncomment the following line to fix the code:
 # global Money
 Money = Money + 1

print(Money)
AddMoney()
print(Money)
```



www.spiraltrain.nl

Modules

78

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:

the innermost scope, which is searched first, contains the local names

the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names

the next-to-last scope contains the current module’s global names

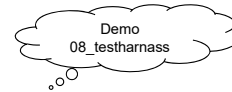
the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the middle scope containing the module’s global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variable are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module’s namespace. Class definitions place yet another namespace in the local scope.

## Test Harnass

- **To run code only when script is run, not when imported**
- **`if __name__ == '__main__':`**
  - Only true only when the file is run
  - Not when the module is imported
  - Good practice to test how module is used
- **Code in module only evaluated first time it is imported:**
  - Python maintains an internal list of all modules that have been imported
- **When you import a module for the first time:**
  - Module script is executed in its own namespace until the end
  - Internal list is updated, and execution of continues after the import statement
- **Can use reload to force new import e.g. when module changed**



[www.spiraltrain.nl](http://www.spiraltrain.nl)

Modules

79

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no global statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, import statements and function definitions bind the module or function name in the local scope.

The global statement can be used to indicate that particular variables live in the global scope and should be rebound there; the nonlocal statement indicates that particular variables live in an enclosing scope and should be rebound there.

## Installing Packages

- **From PyPI or from VCS, other indexes, local archives**
- **On Linux or OS X:**
  - `pip install -U pip setuptools`
- **On Windows:**
  - `python -m pip install -U pip setuptools`
- **Common usage of `pip`:**
  - Install from the Python Package Index using a requirement specifier

```
pip install package
pip install package==1.3
pip install package~=1.3
pip install package>1.3
pip install --upgrade package
pip install -r requirements.txt
pip freeze > requirements.txt
pip list
pip uninstall package
```

[www.spiraltrain.nl](http://www.spiraltrain.nl)

Modules

80

It's important to note that the term “package” in this context is being used as a synonym for a distribution (i.e. a bundle of software to be installed), not to refer to the kind of package that you import in your Python source code (i.e. a container of modules).

It is common in the Python community to refer to a distribution using the term “package”. Using the term “distribution” is often not preferred, because it can easily be confused with a Linux distribution, or another larger software distribution like Python itself.

`pip` is the recommended installer. There are a few cases where you might want to use `easy_install` instead of `pip`.

### Install `pip`, `setuptools`, and `wheel`

If you have Python 2 >=2.7.9 or Python 3 >=3.4 installed from [python.org](http://python.org), you will already have `pip` and `setuptools`, but will need to upgrade to the latest version:

On Linux or macOS:

```
pip install -U pip setuptools
```

On Windows:

```
python -m pip install -U pip setuptools
```

The most common usage of `pip` is to install from the Python Package Index using a requirement specifier. Generally speaking, a requirement specifier is composed of a project name followed by an optional version specifier. PEP 440 contains a full specification of the currently supported specifiers. Below are some examples.

To install the latest version of “SomeProject”:

```
pip install 'SomeProject'
```

To install a specific version:

```
pip install 'SomeProject==1.4'
```



## Virtual Environments

- Install packages in separate location, not shared globally
- Each environment has its own Python binary
- Handles different apps requiring different versions of package
- Applications won't break by subsequent library updates
- Makes it easier to create distributions
- Create virtual environment as follows:

```
pip install virtualenv
virtualenv c:\path\to\myenv
```

- **OR:**

```
python -m venv myenv c:\path\to\myenv (>Python3.4)
```

- **To make using virtual environments even easier:**

```
pip install virtualenvwrapper
```

[www.spiraltrain.nl](http://www.spiraltrain.nl)

Modules

81

Python “Virtual Environments” allow Python packages to be installed in an isolated location for a particular application, rather than being installed globally.

Imagine you have an application that needs version 1 of LibFoo, but another application requires version 2. How can you use both these applications? If you install everything into /usr/lib/python2.7/site-packages (or whatever your platform’s standard location is), it’s easy to end up in a situation where you unintentionally upgrade an application that shouldn’t be upgraded.

Or more generally, what if you want to install an application and leave it be? If an application works, any change in its libraries or the versions of those libraries can break the application.

Also, what if you can’t install packages into the global site-packages directory? For instance, on a shared host.

In all these cases, virtual environments can help you. They have their own installation directories and they don’t share libraries with other virtual environments.

Currently, there are two viable tools for creating Python virtual environments:

`venv` is available by default in Python 3.3 and later, and installs pip and setuptools into created virtual environments in Python 3.4 and later.

`virtualenv` needs to be installed separately, but supports Python 2.6+ and Python 3.3+, and pip, setuptools and wheel are always installed into created virtual environments by default (regardless of Python version).

## Switching Environments

- `activate` **to set path and change prompt**
- `deactivate` **to return to standard python environment**

| Platform | Shell      | Command to activate virtual environment |
|----------|------------|-----------------------------------------|
| Posix    | bash/zsh   | \$ source <venv>/bin/activate           |
|          | fish       | \$ . <venv>/bin/activate.fish           |
|          | csh/tcsh   | \$ source <venv>/bin/activate.csh       |
| Windows  | cmd.exe    | C:\> <venv>\Scripts\activate.bat        |
|          | PowerShell | PS C:\> <venv>\Scripts\Activate.ps1     |

- `Virtualenvwrapper` **makes switching environments easier:**  
`mkvirtualenv env`  
`workon`  
`rmvirtualenv`

`virtualenvwrapper` is a set of extensions to Ian Bicking's `virtualenv` tool. The extensions include wrappers for creating and deleting virtual environments and otherwise managing your development workflow, making it easier to work on more than one project at a time without introducing conflicts in their dependencies.

### Features

Organizes all of your virtual environments in one place.

Wrappers for managing your virtual environments (create, delete, copy).

Use a single command to switch between environments.

Tab completion for commands that take a virtual environment as argument.

User-configurable hooks for all operations.

Plugin system for more creating sharable extensions.

## Creating classes (i.e. data types)

- **class statement creates a new class definition:**

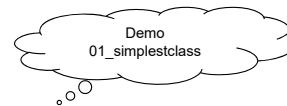
```
class ClassName:
 'Optional class documentation string'
 class_suite
```

- **The class has a documentation string:**

- Can be access via `ClassName.__doc__`

- **class\_suite consists of statements, executed when defined:**

- definitions of functions in namespace of class (**methods**)
- assignment of class attributes



[www.spiraltrain.nl](http://www.spiraltrain.nl)

Classes and Objects

83

You must be wondering how Python gives the value for self and why you don't need to give a value for it. An example will make this clear. Say you have a class called MyClass and an instance of this class called myobject. When you call a method of this object as `myobject.method(arg1, arg2)`, this is automatically converted by Python into `MyClass.method(myobject, arg1, arg2)` - this is all the special self is about.

This also means that if you have a method which takes no arguments, then you still have to have one argument - the self.

The simplest class possible is shown in the following example:

```
class Person:
 pass # An empty block

p = Person()
print(p)
```

### Output:

```
$ python simplestclass.py
<__main__.Person object at 0x019F85F0>
```

We create a new class using the class statement and the name of the class. This is followed by an indented block of statements which form the body of the class. In this case, we have an empty block which is indicated using the pass statement.

Next, we create an object/instance of this class using the name of the class followed by a pair of parentheses. For our verification, we confirm the type of the variable by simply printing it. It tells us that we have an instance of the Person class in the `__main__` module.

Notice that the address of the computer memory where your object is stored is also printed. The address will have a different value on your computer since Python can store the object wherever it finds space.

## Example Creating Classes and Objects

```
class Employee:
 'Common base class for all employees'
 empCount = 0
 def __init__(self, name, salary):
 self.name = name
 self.salary = salary
 Employee.empCount += 1
 def displayCount(self):
 print("Total Employee %d" % Employee.empCount)
 def displayEmployee(self):
 print("Name: ", self.name, " , Salary: ", self.salary)
```

- **To create instances of a class:**
  - Call class using class name and pass arguments that `__init__` method accepts
- **Create object of `Employee` class:**

```
empl = Employee("Zara", 2000)
```



www.spiraltrain.nl

Classes and Objects

84

We have already discussed that classes/objects can have methods just like functions except that we have an extra `self` variable. We will now see an example.

```
class Person:
 def sayHi(self):
 print('Hello, how are you?')
p = Person()
p.sayHi()
```

# This short example can also be written as `Person().sayHi()`

### Output:

```
$ python method.py
Hello, how are you?
```

Here we see the `self` in action. Notice that the `sayHi` method takes no parameters but still has the `self` in the function definition.

## Class Members

- `empCount` :
  - Class variable whose value would be shared among all instances of a this class
  - Accessible as `Employee.empCount` from inside the class or outside the class
- `__init__()` :
  - Special method which is called class constructor or initialization method
  - Called by Python when you create a new instance of this class
- **Instance methods:**
  - Declared like normal functions but first argument to each method is `self`
  - No need to include this argument when you call methods since Python adds it
- `self` **must be explicitly listed as first argument to method:**
  - Instance variables are referred to with "`self.XXX`"
- `self` **is a reference to the instance:**
  - Reference to the container for state of object
  - In Python, instance is visible and *explicit* in method definitions



www.spiraltrain.nl

Classes and Objects

85

There are many method names which have special significance in Python classes. We will see the significance of the `__init__` method now.

The `__init__` method is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object. Notice the double underscores both at the beginning and at the end of the name.

```
class Person:
 def __init__(self, name):
 self.name = name
 def sayHi(self):
 print('Hello, my name is', self.name)

p = Person('Pipo')
p.sayHi()

This short example can also be written as Person('Pipo').sayHi()
```

### Output:

```
$ python class_init.py
Hello, my name is Swaroop
```

## Creating and Using Objects

- **Creating objects:**

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```

- **Access object's attributes using dot operator with object :**

```
emp1.displayEmployee()
emp2.displayEmployee()
```

- **Access class variable using class name:**

```
print("Total Employee %d" % Employee.empCount)
```

- **Result:**

```
Name: Zara ,Salary: 2000
Name: Manni ,Salary: 5000
Total Employee 2
```



www.spiraltrain.nl

Classes and Objects

86

Here, we define the `__init__` method as taking a parameter name (along with the usual `self`). Here, we just create a new field also called `name`. Notice these are two different variables even though they are both called 'name'. There is no problem because the dotted notation `self.name` means that there is something called "name" that is part of the object called "self" and the other name is a local variable. Since we explicitly indicate which name we are referring to, there is no confusion.

Most importantly, notice that we do not explicitly call the `__init__` method but pass the arguments in the parentheses following the class name when creating a new instance of the class. This is the special significance of this method.

Now, we are able to use the `self.name` field in our methods which is demonstrated in the `sayHi` method.

### Class And Object Variables

We have already discussed the functionality part of classes and objects (i.e. methods), now let us learn about the data part. The data part, i.e. fields, are nothing but ordinary variables that are *bound* to the **namespaces** of the classes and objects. This means that these names are valid within the context of these classes and objects only. That's why they are called *name spaces*.

There are two types of *fields* - class variables and object variables which are classified depending on whether the class or the object *owns* the variables respectively.

*Class variables* are shared - they can be accessed by all instances of that class. There is only one copy of the class variable and when any one object makes a change to a class variable, that change will be seen by all the other instances.

*Object variables* are owned by each individual object/instance of the class. In this case, each object has its own copy of the field i.e. they are not shared and are not related in any way to the field by the same name in a different instance.

## Accessing Attributes

- In Python can add, remove, or modify attributes at any time:

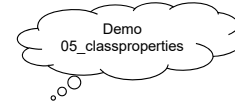
```
empl.age = 7 # Add an 'age' attribute
empl.age = 8 # Modify 'age' attribute
del empl.age # Delete 'age' attribute
```

- There are alternative functions to access attributes:

- `getattr(obj, name[, default])`: Access the attribute of object
- `hasattr(obj, name)`: Check if attribute exists or not
- `setattr(obj, name, value)`: Set or create attribute
- `delattr(obj, name)`: Delete an attribute

- Example code:

```
hasattr(empl, 'age') # Returns true if 'age' attribute exists
getattr(empl, 'age') # Returns value of 'age' attribute
setattr(empl, 'age', 8) # Set attribute 'age' at 8
delattr(empl, 'age') # Delete attribute 'age'
```



www.spiraltrain.nl

Classes and Objects

87

```
class Robot:
 population = 0
 def __init__(self, name):
 '''Initializes the data.'''
 self.name = name
 print('(Initializing {0})'.format(self.name))
 Robot.population += 1
 def __del__(self):
 '''I am dying.'''
 print('{0} is being destroyed!'.format(self.name))
 Robot.population -= 1
 if Robot.population == 0:
 print('{0} was the last one.'.format(self.name))
 else:
 print('There are still {0:d} robots /
 working.'.format(Robot.population))
 def sayHi(self):
 '''Greeting by the robot.
 Yeah, they can do that.'''
 print('Greetings, my masters call me /
 {0}'.format(self.name))
 def howMany():
 '''Prints the current population.'''
 print('We have {0:d} /
 robots.'.format(Robot.population))
 howMany = staticmethod(howMany)
```

## Built-In Class Attributes

- **Every Python class has following built-in attributes:**
  - `__dict__` : Dictionary containing the class's namespace
  - `__doc__` : Class documentation string, or None if undefined
  - `__name__` : Class name.
  - `__module__` : Module name in which the class is defined  
This attribute is "`__main__`" in interactive mode.
  - `__bases__` : A possibly empty tuple containing the base classes  
In the order of their occurrence in the base class list
- **Can be accessed using dot operator like any other attribute**

www.spiraltrain.nl

Classes and Objects

88

```
droid1 = Robot('R2-D2')
droid1.sayHi()
Robot.howMany()
droid2 = Robot('C-3PO')
droid2.sayHi()
Robot.howMany()
print("\nRobots can do some work here.\n")
print("Robots have finished their work. So let's destroy them.")
del droid1
del droid2
Robot.howMany()
```

This is a long example but helps demonstrate the nature of class and object variables. Here, `population` belongs to the `Robot` class and hence is a class variable. The `name` variable belongs to the object (it is assigned using `self`) and hence is an object variable.

Thus, we refer to the population class variable as `Robot.population` and not as `self.population`. We refer to the object variable name using `self.name` notation in the methods of that object. Remember this simple difference between class and object variables. Also note that an object variable with the same name as a class variable will hide the class variable!

The `howMany` is actually a method that belongs to the class and not to the object. This means we can define it as either a classmethod or a staticmethod depending on whether we need to know which class we are part of. Since we don't need such information, we will go for staticmethod.

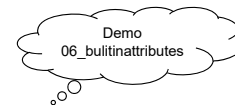


## Example Accessing Built-in Class Attributes

```
print("Employee.__doc__:", Employee.__doc__)
print("Employee.__name__:", Employee.__name__)
print("Employee.__module__:", Employee.__module__)
print("Employee.__bases__:", Employee.__bases__)
print("Employee.__dict__:", Employee.__dict__)
```

- **Result:**

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>,)
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```



www.spiraltrain.nl

Classes and Objects

89

We could have also achieved the same using decorators:

```
@staticmethod
def howMany():
 '''Prints the current population.'''
 print('We have {0:d}
 robots.'.format(Robot.population))
```

Decorators can be imagined to be a shortcut to calling an explicit statement, as we have seen in this example.

Observe that the `__init__` method is used to initialize the `Robot` instance with a name. In this method, we increase the population count by 1 since we have one more robot being added. Also observe that the values of `self.name` is specific to each object which indicates the nature of object variables.

Remember, that you must refer to the variables and methods of the same object using the `self` only. This is called an attribute reference.

In this program, we also see the use of docstrings for classes as well as methods. We can access the class docstring at runtime using `Robot.__doc__` and the method docstring as `Robot.sayHi.__doc__`

Just like the `__init__` method, there is another special method `__del__` which is called when an object is going to die i.e. it is no longer being used and is being returned to the computer system for reusing that piece of memory. In this method, we simply decrease the `Robot.population` count by 1.

## Destroying Objects

- **Python deletes unneeded objects automatically to free space:**
  - This process of periodically reclaiming memory is termed Garbage Collection
- **Python's garbage collector runs during program execution:**
  - Objects deleted when its reference count reaches zero
- **Reference count changes as references to it change:**
  - Increase when it's assigned a new name
  - Increase when placed in a container (list, tuple, or dictionary)
  - Decrease when var deleted, or removed from collection, with `del`
  - Decrease when reference is reassigned or goes out of scope
- **`__del__()` method:**
  - called when object is actually destroyed
  - may be used to clean up resources used by an instance, for logging etc.

The `__del__` method is run when the object is no longer in use and there is no guarantee when that method will be run. If you want to explicitly see it in action, we have to use the `del` statement which is what we have done here.

### Note for C++/Java/C# Programmers

All class members (including the data members) are public and all the methods are virtual in Python.

One exception: If you use data members with names using the double underscore prefix such as `__privatevar`, Python uses name-mangling to effectively make it a private variable.

Thus, the convention followed is that any variable that is to be used only within the class or object should begin with an underscore and all other names are public and can be used by other classes/objects. Remember that this is only a convention and is not enforced by Python (except for the double underscore prefix).

## Example Destructor

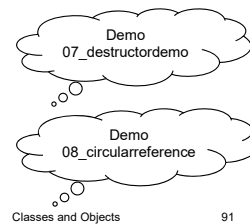
```
class Point:
 def __init__(self, x=0, y=0):
 self.x = x
 self.y = y
 def __del__(self):
 class_name = self.__class__.__name__
 print(class_name, "destroyed")

pt1 = Point()
pt2 = pt1
pt3 = pt1
print(id(pt1), id(pt2), id(pt3)) # prints id's of objects
del pt1
del pt2
del pt3
```

- **Result:**

```
3083401324 3083401324 3083401324
Point destroyed
```

[www.spiraltrain.nl](http://www.spiraltrain.nl)



Classes and Objects

91

```
class Foo:
 def __init__(self, x):
 print "Foo: Hi"
 self.x = x
 def __del__(self):
 print "Foo: bye"

class Bar:
 def __init__(self):
 print "Bar: Hi"
 self.foo = Foo(self) # x = this instance
 def __del__(self):
 print "Bar: Bye"

bar = Bar()
del bar # This doesn't work either.
```

What the above code does is that the Foo instance keeps a reference to its creator class, which is an instance of Bar. The output is:

```
Bar: Hi
Foo: Hi
```

As you can see, the destructors are never called, not even when we add a `del bar` at the end of the program. Removing the `self.x = x` solves (well, makes it disappear) the problem. The reason that `__del__` is never called suddenly becomes obvious when looking at the above code. It's a 'problem' with certain garbage collectors, namely: circular referencing. Python uses a reference counting garbage collecting algorithm. Such an garbage collection algorithm increases a counter on each data instance for each reference that exists to that data instance and decreases the counter when a reference to the data instance is removed. When the counter reaches zero, the data instance is garbage collected because nothing points to it anymore. Reference counting has a problem with circular links.

## Data Hiding

- **Python does not support real data hiding as other languages:**
  - Attributes and methods can however be made more difficult to access
- **Attributes made private with double underscore prefix in name:**

```
__hidden = 0
```

- **Attributes (methods) will not be directly visible to outsiders:**
  - Access of such attributes from outside should include class name

```
instance._InstanceClassName__hidden
```

## Private Variables

There is limited support for class-private identifiers. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `__classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, so it can be used to define class-private instance and class variables, methods, variables stored in globals, and even variables stored in instances. private to this class on instances of *other* classes. Truncation may occur when the mangled name would be longer than 255 characters. Outside classes, or when the class name consists of only underscores, no mangling occurs.

Name mangling is intended to give classes an easy way to define "private" instance variables and methods, without having to worry about instance variables defined by derived classes, or mucking with instance variables by code outside the class. Note that the mangling rules are designed mostly to avoid accidents; it still is possible for a determined soul to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger, and that's one reason why this loophole is not closed. (Buglet: derivation of a class with the same name as the base class makes use of private variables of the base class possible.)

Notice that code passed to `exec`, `eval()` or `execfile()` does not consider the `classname` of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly.

## Class Inheritance

- **Class can be derived from one or more parent classes:**
  - List parent classes in parentheses after the class name
- **Child class inherits the attributes of its parent class:**
  - Attributes can be used as if they were defined in the child class
  - Child class can also override data members and methods from the parent
- **Derived classes are declared much like their parent class:**

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
 'Optional class documentation string'
 class_suite
```
- **Subclasses can initialize base class object by:**
  - Calling `super().__init__()`
  - Calling `BaseClass.__init__(self, name)`

www.spiraltrain.nl

Classes and Objects

93

Of course, a language feature would not be worthy of the name "class" without supporting inheritance. The syntax for a derived class definition looks like this:

```
class DerivedClassName (BaseClassName) :
 <statement-1>
 ...
 <statement-N>
```

The name `BaseClassName` must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName (modname.BaseClassName) :
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `'BaseClassName.methodname(self, arguments)'`. This is occasionally useful to clients as well. (Note that this only works if the base class is defined or imported directly in the global scope.)

## Example Class Derivation

```
class Employee: # define base class
 def __init__(self, name):
 self.name = name
 def calcSalary(self):
 self.salary = 0
 return self.salary
 def setName(self, name): self.name = name
 def getName(self): return self.name

class WageEmployee(Employee): # define derived class
 def __init__(self, name, wage, hours):
 super().__init__(name)
 # Employee.__init__(self, name) # alternative
 self.wage = wage
 self.hours = hours
 def setWage(self, wage): self.wage = wage
 def getWage(self): return self.wage
```

www.spiraltrain.nl

Classes and Objects

94

One of the major benefits of object oriented programming is **reuse** of code and one of the ways this is achieved is through the *inheritance* mechanism. Inheritance can be best imagined as implementing a *type and subtype* relationship between classes.

Suppose you want to write a program which has to keep track of the teachers and students in a college. They have some common characteristics such as name, age and address. They also have specific characteristics such as salary, courses and leaves for teachers and, marks and fees for students.

You can create two independent classes for each type and process them but adding a new common characteristic would mean adding to both of these independent classes. This quickly becomes unwieldy.

A better way would be to create a common class called *SchoolMember* and then have the teacher and student classes *inherit* from this class i.e. they will become sub-types of this type (class) and then we can add specific characteristics to these sub-types.

There are many advantages to this approach. If we add/change any functionality in *SchoolMember*, this is automatically reflected in the subtypes as well. For example, you can add a new ID card field for both teachers and students by simply adding it to the *SchoolMember* class. However, changes in the subtypes do not affect other subtypes. Another advantage is that if you can refer to a teacher or student object as a *SchoolMember* object which could be useful in some situations such as counting of the number of school members. This is called **polymorphism** where a sub-type can be substituted in any situation where a parent type is expected i.e. the object can be treated as an instance of the parent class.

Also observe that we *reuse* the code of the parent class and we do not need to repeat it in the different classes as we would have had to in case we had used independent classes.

The *SchoolMember* class in this situation is known as the *base class* or the *superclass*. The Teacher and Student classes are called the *derived classes* or *subclasses*.

## Example Using Derived Classes

```
e = Employee("Employee 1")
w = WageEmployee("WageEmployee 1", 10, 10)

print(e.getName())
print(e.calcSalary())

print(w.getName())
print(w.calcSalary())
print(w.getWage())
print(w.getHours())
```

- **Result:**

```
Employee 1
0
WageEmployee 1
0
10
10
```



www.spiraltrain.nl

Classes and Objects

95

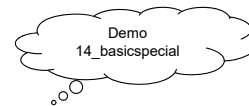
```
class SchoolMember:
 '''Represents any school member.'''
 def __init__(self, name, age):
 self.name = name
 self.age = age
 print('(Initialized SchoolMember:
 {0})'.format(self.name))
 def tell(self):
 '''Tell my details.'''
 print('Name:"{0}" Age:"{1}"'.format
 (self.name, self.age), end=" ")

class Teacher(SchoolMember):
 '''Represents a teacher.'''
 def __init__(self, name, age, salary):
 SchoolMember.__init__(self, name, age)
 self.salary = salary
 print('(Initialized Teacher: {0})'.format(self.name))
 def tell(self):
 SchoolMember.tell(self)
 print('Salary: "{0:d}"'.format(self.salary))
```

## Inheritance

- **Class Special that inherits from a super-class Basic:**

```
class Basic:
 def __init__(self, name): self.name = name
 def show(self): print('Basic -- name: %s' % self.name)
class Special(Basic):
 def __init__(self, name, edible):
 Basic.__init__(self, name)
 self.upper = name.upper()
 self.edible = edible
 def show(self):
 Basic.show(self)
 print('Special -- upper name: %s.' % self.upper)
 if self.edible:
 print("It's edible.")
 else:
 print("It's not edible.")
 def edible(self):
 return self.edible
```



www.spiraltrain.nl

Classes and Objects

96

```
class Student(SchoolMember):
 '''Represents a student.'''
 def __init__(self, name, age, marks):
 SchoolMember.__init__(self, name, age)
 self.marks = marks
 print('(Initialized Student: {0})'.format(self.name))
 def tell(self):
 SchoolMember.tell(self)
 print('Marks: "{0:d}"'.format(self.marks))
t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 25, 75)
print() # prints a blank line
members = [t, s]
for member in members:
 member.tell() # works for both Teachers and Students
```

### Output:

```
$ python inherit.py
(Initialized SchoolMember: Mrs. Shrividya)
(Initialized Teacher: Mrs. Shrividya)
(Initialized SchoolMember: Swaroop)
(Initialized Student: Swaroop)
```

```
Name:"Mrs. Shrividya" Age:"40" Salary: "30000"
Name:"Swaroop" Age:"25" Marks: "75"
```



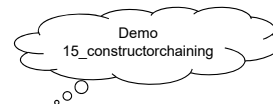
## Constructor Chaining

```
class A(object):
 def __init__(self):
 print("Constructor A was called")

class B(A):
 def __init__(self):
 super().__init__()
 print("Constructor B was called")

class C(B):
 def __init__(self):
 super().__init__()
 print("Constructor C was called")

c = C()
```



www.spiraltrain.nl

97

To use inheritance, we specify the base class names in a tuple following the class name in the class definition. Next, we observe that the `__init__` method of the base class is explicitly called using the `self` variable so that we can initialize the base class part of the object. This is very important to remember - Python does not automatically call the constructor of the base class, you have to explicitly call it yourself. We also observe that we can call methods of the base class by prefixing the class name to the method call and then pass in the `self` variable along with any arguments. Notice that we can treat instances of `Teacher` or `Student` as just instances of the `SchoolMember` when we use the `tell` method of the `SchoolMember` class. Also, observe that the `tell` method of the subtype is called and not the `tell` method of the `SchoolMember` class. One way to understand this is that Python *always* starts looking for methods in the actual type, which in this case it does. If it could not find the method, it starts looking at the methods belonging to its base classes one by one in the order they are specified in the tuple in the class definition. A note on terminology - if more than one class is listed in the inheritance tuple, then it is called *multiple inheritance*.

The `end` parameter is used in the `tell()` method to change a new line to be started at the end of the `print()` call to printing spaces.

```
class Farm(object):
 def __init__(self): pass
class Barn(Farm):
 def __init__(self):
 super(Barn, self).__init__()
```

If you want to do the same for old-style classes you simply can't use `super()` so you'll have to do this:

```
class Farm:
 def __init__(self): pass
class Barn(Farm):
 def __init__(self):
 Farm.__init__(self)
```

## Multiple Inheritance

- **Class can be derived from multiple parent classes:**

```
class A: # define your class A
.....
class B: # define your class B
.....
class C(A, B): # subclass of A and B
```

- **Functions to check relationships of two classes and instances:**

- `issubclass()` or `isinstance()`
- `issubclass(sub, sup)` **boolean function:**
  - Returns true if the given subclass sub is a subclass of the superclass sup
- `isinstance(obj, Class)` **boolean function:**
  - Returns true if obj is instance of class Class or is an instance of a subclass of Class

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
 <statement-1>
 <statement-N>
```

The only rule necessary to explain the semantics is the resolution rule used for class attribute references. This is depth-first, left-to-right. Thus, if an attribute is not found in `DerivedClassName`, it is searched in `Base1`, then (recursively) in the base classes of `Base1`, and only if it is not found there, it is searched in `Base2`, and so on.

(To some people breadth first--searching `Base2` and `Base3` before the base classes of `Base1`---looks more natural. However, this would require you to know whether a particular attribute of `Base1` is actually defined in `Base1` or in one of its base classes before you can figure out the consequences of a name conflict with an attribute of `Base2`. The depth-first rule makes no differences between direct and inherited attributes of `Base1`.)

It is clear that indiscriminate use of multiple inheritance is a maintenance nightmare, given the reliance in Python on conventions to avoid accidental name conflicts. A well-known problem with multiple inheritance is a class derived from two classes that happen to have a common base class. While it is easy enough to figure out what happens in this case (the instance will have a single copy of "instance variables" or data attributes used by the common base class), it is not clear that these semantics are in any way useful.

## Overriding Methods

- **Always possible to override parent class methods:**

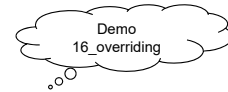
- Reason could be that special or different functionality in your subclass is needed

```
class Parent: # define parent class
 def myMethod(self):
 print('Calling parent method')
class Child(Parent): # define child class
 def myMethod(self):
 print('Calling child method')

c = Child() # instance of child
c.myMethod() # child calls overridden method
```

- **Result:**

Calling child method



www.spiraltrain.nl

Classes and Objects

99

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as `BaseClassName` in the global scope.)

## Override Base Methods

- **Generic functionality that you can override in your own classes:**

- `__init__(self [,args...]):`
  - Constructor (with any optional arguments)  
`obj = className(args)`
- `__del__(self):`
  - Destructor, called by system when object is actually deleted  
`del obj`
- `__repr__(self):`
  - Evaluatable string representation  
`repr(obj)`
- `__str__(self):`
  - Printable string representation  
`str(obj)`

In Python, a class may inherit from more than one superclass. This is called *multiple inheritance*. It is not absolutely essential, but it does have a few uses.

For example, lists, being mutable, cannot be used as keys in dictionaries. But suppose we need to use lists as keys. Suppose we aren't interested in looking up lists by their contents, but by their identities; that is, when we look up the same list object, we want to find it, but when we look up a different list object with the same contents, we do not.

## Class Methods

- **Class methods receive class as first parameter:**
  - Aren't specific to any particular instance, but still involve the class in some way
  - Class methods can be overridden or redefined by subclasses
- **Method can be made a class method in two ways:**
  - Annotate method with `@classmethod` decoration

```
class Bank:
 @classmethod
 def is_valid(cls, key): # code for determining validity here
 return valid
 def add_key(self, key, val):
 if not Bank.is_valid(key):
 raise ValueError()
 # Use method without instance, signals code closely-associated with Bank
 Bank.is_valid('my key')
```

- class method typically is called through the class



www.spiraltrain.nl

Classes and Objects

101

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C:
 @classmethod
 def f(cls, arg1, arg2, ...): ...
```

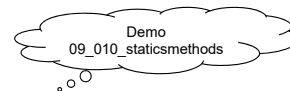
The `@classmethod` form is a function decorator. It can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument. Class methods are different than C++ or Java static methods. One of the best uses of class methods is as constructors. For instance, if you want to have multiple constructors, but don't want to rely on one method that simply accepts different sorts of arguments, then use different class methods.

```
class MyClass:
 def __init__(self): # "base" constructor.
 pass
 @classmethod
 def one_constructor(cls, foo): # Special constructor.
 self = cls()
 self.foo = foo
 return self
 @classmethod
 def another_constructor(cls, bar): # Constructor.
 pass
class MySubclass(MyClass): # Does necessary customizations.
 pass
obj = MySubclass.one_constructor('foo')
```

## Static Methods

- **Static methods do not receive `self` or `cls` as first parameter :**
  - Like regular function but has to be called in namespace of class
- **Method can be made `static` in two ways :**
  - Annotate method with `@staticmethod` decoration

```
class Account :
 interestRate = 10
 def getInterestRate1():
 return Account.interestRate
 getInterestRate1 = staticmethod(getInterestRate1)
 @staticmethod
 def getInterestRate2():
 return Account.interestRate
print (Account.getInterestRate1())
print (Account.getInterestRate2())
```



It is possible to have static methods in Python that you can call them without initializing a class, like :

```
ClassName.StaticMethod ()
```

This can be done using the `staticmethod` decorator

```
class MyClass(object):
 @staticmethod
 def the_static_method(x):
 print x
```

```
MyClass.the_static_method(2) # outputs 2
```

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
class C:
 @staticmethod
 def f(arg1, arg2, ...): ...
```

The `@staticmethod` form is a function decorator. It can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++.

## Operator Overloading

- **Suppose** `Vector` **class represents two-dimensional vectors:**
  - Plus operator will not work to add them automatically
- **Define** `__add__` **method to perform vector addition:**
  - Plus operator would behave as per expectation

```
class Vector:
 def __init__(self, a, b):
 self.a = a
 self.b = b
 def __str__(self):
 return 'Vector (%d, %d)' % (self.a, self.b)
 def __add__(self, other):
 return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print(v1 + v2)
```

- **Result:** `Vector(7,8)`

[www.spiraltrain.nl](http://www.spiraltrain.nl)



Classes and Objects

103

If you aren't familiar with the phrase 'Operator Overloading', it is basically the giving of a different meaning to a program's operators like `+` or `==`. Programming languages such as C, C++, Java, Python and many others offer this ability.

So when and why would you ever want to do this? Well, for myself, I actually ran across this need today with a tool I'm working on. I didn't actually think Python had the ability to do this until I did some Googling on the Internets.

I wrote a `Vector` class in Python that had members for the X, Y and Z components. So, to see if two vectors are the same, I would have needed to do this:

```
if vec_a.x == vec_b.x and vec_a.y == vec_b.y and vec_a.z == vec_b.z:
 ...do something here...
```

Well, that's just messy to look at and is a pain to write out. Wouldn't it be nicer if you could just say:

```
if vec_a == vec_b:
 ...do something here...
```

You can do this as long as you overload the operator. To overload the operator, you need to implement the overload function(s) within your class object. To overload the equality `==` and non-equality `!=` operators, you use `__eq__` and `__ne__`.

## Polymorphism

- Call of same function / method on different types of objects is handled by different forms of that function
- Generic method call results in specific runtime action

```
cir = Circle(1, 2)
rec = Rectangle(3, 4, 2, 2)
list = [cir, rec]
for item in list:
 item.draw();
```

In static-type languages only works if those objects derive from same super-class or interface. In Python you can arrange it that way, but no need to: **duck typing** implies that it is acceptable (works) if you just try it (with proper handling of cases when this fails)

Polymorphism is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation

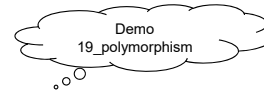
Consider a drawing which consists of a list of different type of figure objects, like circles, rectangles and lines. Each figure has a draw method. In order to display the drawing all the figures must be drawn. We can do this by iterating over the list and thereby executing the draw method for each item in the list. We execute the same draw call for all items in the list but through polymorphism different version of this method gets called.

The concept of polymorphism is often expressed by the phrase “one interface, multiple methods”. This means that it is possible to design a generic interface to a group of related activities. This helps to reduce complexity by allowing the same interface to be used to specify a general class of action. The specific action, that is method, to be called in each situation is decided at run time by the Java Virtual Machine.



## Polymorphism in Python

```
class A(object):
 def show(self, msg):
 print('class A -- msg: "%s"' % (msg,))
class B(object):
 def show(self, msg):
 print('class B -- msg: "%s"' % (msg,))
class C(object):
 def show(self, msg):
 print('class C -- msg: "%s"' % (msg,))
def test():
 objs = [A(), B(), C(), A(),]
 for idx, obj in enumerate(objs):
 msg = 'message # %d' % (idx + 1,)
 obj.show(msg)
if __name__ == '__main__':
 test()
```



www.spiraltrain.nl

Classes and Objects

105

The traditional concept of Polymorphism where a parent class variable can call it's children methods doesn't exist or make a difference. I can use one variable to hold any kind of objects. No matter what kind of object the variable is referring too, as long as the object has the method name I call, everything is OK. Object type is not important. Virtual methods make no difference in terms of Polymorphism in Python, but it does have an effect in inheritance:

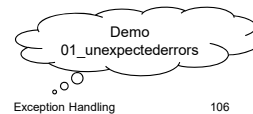
```
class Monkey:
 def show(self):
 print "This is monkey"
class Car:
 def show(self):
 print "This is a car"
class House:
 def show(self):
 print "This is a house"
list = []
list.append (Monkey()); list.append (Car()); list.append (House())
for x in list:
 x.show()
```

The above example uses the usual algorithm other books use to demonstrate polymorphism. It works simpler in Python. However, you probably missed a subtle thing that the example shows. It is that the list can hold three object of different type and call the methods without any typecasting at all. Generic object containers are the holy grails of other OO languages. You can't do this directly in C++, Object Pascal or even in Java. In Java at least you can use list-type interfaces, which the objects have to support. In C++ or Object Pascal, the list has to be the parent type of those objects, which is more restrictive than an interface. Can you see what Python offers for the OO world?

## Unexpected Errors

- **Python provides exceptions to handle unexpected errors:**
  - Exceptions are events disrupting normal program flow
  - May occur during the execution of program instructions
- **When Python encounters conditions it can't cope with:**
  - An exception is raised
  - An exception is a Python object that represents an error
- **When a Python script raises an exception:**
  - The exception must be handled immediately otherwise the script terminates
- **Another feature to handle unexpected errors are assertions:**
  - Assertions are tests which must be true to continue

www.spiraltrain.nl



Exception Handling

106

Exceptions occur when certain *exceptional* situations occur in your program. For example, what if you are going to read a file and the file does not exist? Or what if you accidentally deleted it when the program was running? Such situations are handled using **exceptions**. Similarly, what if your program had some invalid statements? This is handled by Python which **raises** its hands and tells you there is an **error**.

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.

Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.

Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

## Exception Handling

- **Place code that might raise an exception in a `try` block:**
  - Code in a `try` block is tested for exceptions
- **Include an `except` statement after the `try` block:**
  - Place code to handle the problem as elegantly as possible there
  - Generic `except` clause handles any exception

```
try:
 x = 1
 y = 0
 z = x/y # ZeroDivisionError
except:
 print('Cannot divide by zero')
try:
 fh = open("somefile", "r") # IOError
except:
 print('Cannot open file')
```

- **After handling exception in `except` statement:**
  - Normal program flow continues



www.spiraltrain.nl

Exception Handling

107

## Errors

Consider a simple print function call. What if we misspelt print as Print? Note the capitalization. In this case, Python raises a syntax error.

```
>>> Print('Hello World')
Traceback (most recent call last):
 File "<pyshell#0>", line 1, in <module>
 Print('Hello World')
NameError: name 'Print' is not defined
>>> print('Hello World')
Hello World
```

Observe that a `NameError` is raised and also the location where the error was detected is printed. This is what an error handler for this error does.

## Exceptions

We will try to read input from the user. Press ctrl-d and see what happens.

```
>>> s = input('Enter something --> ')
Enter something -->
Traceback (most recent call last):
 File "<pyshell#2>", line 1, in <module>
 s = input('Enter something --> ')
EOFError: EOF when reading a line
```

Python raises an error called `EOFError` which basically means it found an end of file symbol (which is represented by ctrl-d) when it did not expect to see it.

## Typed Exception Handling

- **except clauses can be conditioned on a type:**
  - Single `try` statement can have multiple typed `except` statements (first match executed)
  - Useful when the `try` block might throw different types of exceptions

```
try:
 x = 1
 y = 0
 z = x/y # ZeroDivisionError
 tup = (1,2,3)
 tup[0] = 2 # TypeError
except ZeroDivisionError:
 print('Division by zero')
except TypeError:
 print('Cannot change tuple contents')
```

- **except corresponding to exception type executes**



We can handle exceptions using the `try..except` statement. We basically put our usual statements within the `try`-block and put all our error handlers in the `except`-block:

```
try:
 text = input('Enter something --> ')
except EOFError:
 print('Why did you do an EOF on me?')
except KeyboardInterrupt:
 print('You cancelled the operation.')
else:
 print('You entered {}'.format(text))
```

### Output :

```
$ python try_except.py
Enter something --> # Press ctrl-d
Why did you do an EOF on me?
$ python try_except.py
Enter something --> # Press ctrl-c
You cancelled the operation.
$ python try_except.py
Enter something --> no exceptions
You entered no exceptions
```

## Exception Handling with Else

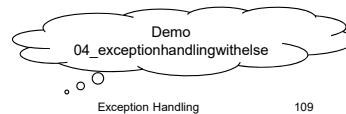
- **After** `except` **clause(s)** `else` **clause** can be included:

- Executes if the code in the `try:` block does not raise an exception

```
try:
 Do you operations here
except:
 If there is any exception, then execute this block
else:
 If there is no exception then execute this block
```

- **Example:**

```
try:
 x = int(input("Please enter a number: "))
except ValueError:
 print("Oops! That was not valid number")
else:
 print("Yeah! That was a really good number")
```



www.spiraltrain.nl

Exception Handling

109

We put all the statements that might raise exceptions/errors inside the `try` block and then put handlers for the appropriate errors/exceptions in the `except` clause/block. The `except` clause can handle a single specified error or exception, or a parenthesized list of errors/exceptions. If no names of errors or exceptions are supplied, it will handle *all* errors and exceptions.

Note that there has to be at least one `except` clause associated with every `try` clause. Otherwise, what's the point of having a `try` block?

If any error or exception is not handled, then the default Python handler is called which just stops the execution of the program and prints an error message. We have already seen this in action above.

You can also have an `else` clause associated with a `try..except` block. The `else` clause is executed if no exception occurs.

The `except` clause with no exceptions:

You can also use the `except` statement with no exceptions defined as follows:

```
try:
 You do your operations here;

except:
 If there is any exception, then execute this block.

else:
 If there is no exception then execute this block.
```

This kind of a `try-except` statement catches all the exceptions that occur. Using this kind of `try-except` statement is not considered a good programming practice, though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

## except Clause Multiple Exceptions

- **Same except clause may also handle multiple exceptions:**

```
try:
 Do you operations here;
except (Exception1[, Exception2[,...ExceptionN]]):
 If there is any exception from the given exception list,
 then execute this block
else:
 If there is no exception then execute this block
```

- **Disadvantage is that error messages are less distinctive**



Programs may name their own exceptions by creating a new exception class (see Classes for more about Python classes). Exceptions should typically be derived from the Exception class, either directly or indirectly. For example:

```
>>> class MyError(Exception):
... def __init__(self, value):
... self.value = value
... def __str__(self):
... return repr(self.value)
...
>>> try:
... raise MyError(2*2)
... except MyError as e:
... print('My exception occurred, value:', e.value)
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

In this example, the default `__init__()` of `Exception` has been overridden. The new behavior simply creates the `value` attribute. This replaces the default behavior of creating the `args` attribute.

## Python Standard Exceptions

| Name               | Description                                                                                      |
|--------------------|--------------------------------------------------------------------------------------------------|
| Exception          | Base class for all exceptions                                                                    |
| StopIteration      | Raised when <code>next()</code> of iterator does not point to any object                         |
| SystemExit         | Raised by the <code>sys.exit()</code> function                                                   |
| StandardError      | Base class for built-in exceptions except <code>StopIteration</code> and <code>SystemExit</code> |
| AritmeticError     | Base class for all errors that occur for numeric calculation                                     |
| OverflowError      | When calculation exceeds maximum limit for numeric type                                          |
| FloatingPointError | Raised when a floating point calculation fails                                                   |
| ZeroDivisionError  | Raised when division by zero occurs for numeric types                                            |
| AssertionError     | Raised in case of failure of the <code>Assert</code> statement                                   |
| AttributeError     | Raised in case of failure of attribute reference or assignment                                   |
| EOFError           | Raised when end of file is reached                                                               |
| ImportError        | Raised when an import statement fails                                                            |
| TypeError          | Raised when operation attempted that is invalid for data type                                    |
| ValueError         | Raised when function for data type has invalid values                                            |
| RuntimeError       | Raised when generated error does not fall into any category                                      |

www.spiraltrain.nl

Exception Handling

111

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```
for line in open("myfile.txt"):
 print(line)
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
 for line in f:
 print(line)
```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Other objects which provide predefined clean-up actions will indicate this in their documentation.

Exceptions should be class objects. The exceptions are defined in the module `exceptions`. This module never needs to be imported explicitly: the exceptions are provided in the built-in namespace as well as the `exceptions` module.

For class exceptions, in a `try` statement with an `except` clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which it is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name. The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an "associated value" indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code). The associated value is the second argument to the `raise` statement. If the exception class is derived from the standard root class `BaseException`, the associated value is present as the exception instance's `args` attribute.

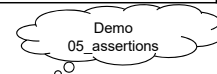
## Python Standard Exceptions

| Name                           | Description                                                                                                                                     |
|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>KeyboardInterrupt</code> | Raised when user interrupts program execution, usually by pressing CTRL-C                                                                       |
| <code>LookupError</code>       | Base class for all lookup errors                                                                                                                |
| <code>IndexError</code>        | Raised when an index is not found in a sequence                                                                                                 |
| <code>KeyError</code>          | Raised when the specified key is not found in the dictionary                                                                                    |
| <code>NameError</code>         | Raised when an identifier is not found in the local or global namespace                                                                         |
| <code>UnboundLocalError</code> | Raised when trying to access a local variable in a function or method but no value has been assigned to it                                      |
| <code>EnvironmentError</code>  | Base class for all exceptions that occur outside the Python environment                                                                         |
| <code>IOError</code>           | Raised when an input /output operation fails, such as the print statement or the open() function when trying to open a file that does not exist |
| <code>OSError</code>           | Raised for operating system related errors                                                                                                      |
| <code>SyntaxError</code>       | Raised when there is an error in Python syntax                                                                                                  |
| <code>IndentationError</code>  | Raised when indentation is not specified properly                                                                                               |
| <code>SystemError</code>       | Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit                  |

www.spiraltrain.nl

Exception Handling

112



When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an `AssertionError` exception. The syntax for assert is:

```
assert Expression[, Arguments]
```

If the assertion fails, Python uses `ArgumentExpression` as the argument for the `AssertionError`. `AssertionError` exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a traceback.

Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature:

```
def KelvinToFahrenheit(Temperature):
 assert (Temperature >= 0), "Colder than absolute zero!"
 return ((Temperature-273)*1.8)+32
print KelvinToFahrenheit(273)
print int(KelvinToFahrenheit(505.78))
print KelvinToFahrenheit(-5)
```

This would produce following result:

```
32.0
451
Traceback (most recent call last):
 File "test.py", line 9, in <module>
 print KelvinToFahrenheit(-5)
 File "test.py", line 4, in KelvinToFahrenheit
 assert (Temperature >= 0), "Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```



## try-finally Clause

- **finally** block may be used along with a **try** block:

- **finally** block is always executed
- Whether or not the try-block raised an exception

- **Syntax of the try-finally:**

```
try:
 Do you operations here;
 Due to any exception, this may be skipped
finally:
 This would always be executed
```

- **Combining finally and except is allowed:**

- **try** block can be followed by multiple **except** clauses and one **finally** clause



www.spiraltrain.nl

Exception Handling

113

Suppose you are reading a file in your program. How do you ensure that the file object is closed properly whether or not an exception was raised? This can be done using the finally block. Note that you can use an except clause along with a finally block for the same corresponding try block. You will have to embed one within another if you want to use both.

```
import time

try:
 f = open('poem.txt')
 while True: # our usual file-reading idiom
 line = f.readline()
 if len(line) == 0:
 break
 print(line, end='')
 time.sleep(2) # To make sure it runs for a while
except KeyboardInterrupt:
 print('!! You cancelled the reading from the file.')
finally:
 f.close()
 print('(Cleaning up: Closed the file)')
```

## Exception Arguments

- **An exception can have an argument:**
  - Value that gives additional information about the problem
  - Contents of the argument vary by exception
- **Capture argument by supplying a variable in `except` clause:**

```
try:
 Do you operations here;
except ExceptionType as var:
 You can print value of var here...
```

- **Variable will receive the value of the exception:**
  - Can receive a single value or multiple values in the form of a tuple
  - Tuple usually contains the error string, the error number and an error location

When an exception occurs, it may have an associated value, also known as the exception's argument. The presence and type of the argument depend on the exception type.

The `except` clause may specify a variable after the exception name (or tuple). The variable is bound to an exception instance with the arguments stored in `instance.args`. For convenience, the exception instance defines `__getitem__` and `__str__` so the arguments can be accessed or printed directly without having to reference `.args`. But use of `.args` is discouraged. Instead, the preferred use is to pass a single argument to an exception (which can be a tuple if multiple arguments are needed) and have it bound to the message attribute. One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
>>> try:
... raise Exception('spam', 'eggs')
... except Exception as inst:
... print type(inst) # the exception instance
... print inst.args # arguments stored in .args
... print inst # __str__ allows args to
... # be printed directly
... x, y = inst # __getitem__ allows args
... # to be unpacked directly
... print 'x =', x
... print 'y =', y
...
<type 'instance'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

## Example Argument of an Exception

```
Define a function here.
def temp_convert(var):
 try:
 return int(var)
 except ValueError as e:
 print("Argument does not contain numbers\n", e)
Call above function here
temp_convert("xyz");
```

- **Result:**

```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```



[www.spiraltrain.nl](http://www.spiraltrain.nl)

Exception Handling

115

If an exception has an argument, it is printed as the last part ('detail') of the message for unhandled exceptions.

Exception handlers don't just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

```
>>> def this_fails():
... x = 1/0
...
>>> try:
... this_fails()
... except ZeroDivisionError as detail:
... print("Handling run-time error:", detail)
...
Handling run-time error: integer division or
modulo by zero
```

## Raising Exceptions

- **Exceptions can be raised with the `raise` statement:**

```
raise # re-raise current exception in except statement
raise Exception(args) # raise new anywhere
raise Exception(args) from e # include current in new
```

- **Explanation:**

- Exception is the type of exception, for example `NameError`
- Argument is a value for the exception argument
- The argument is optional; if not supplied, the exception argument is `None`.
- Final argument `traceback` is optional and is the `traceback` object for exception

- **Example:**

- An exception can be a string, a class, or an object
- Most of the exceptions that the Python core raises are classes
  - with an argument that is an instance of the class

You can raise exceptions using the `raise` statement by providing the name of the error/exception and the exception object that is to be thrown. The error or exception that you can raise should be a class which directly or indirectly must be a derived class of the `Exception` class.

```
class ShortInputException(Exception):
 '''A user-defined exception class.'''
 def __init__(self, length, atleast):
 Exception.__init__(self)
 self.length = length
 self.atleast = atleast

try:
 text = input('Enter something --> ')
 if len(text) < 3:
 raise ShortInputException(len(text), 3)
 # Other work can continue as usual here
except EOFError:
 print('Why did you do an EOF on me?')
except ShortInputException as ex:
 print('ShortInputException: The input was {0} long, expected at least {1}'.format(ex.length, ex.atleast))
else:
 print('No exception was raised.')
```

## Example Raising an Exception

```
try:
 raise NameError('HiThere')
```

In order to catch an exception:

- `except` should refer to same exception thrown as class object

- **To capture above exception, write `except` clause as follows:**

```
try:
 Business Logic here...
except NameError:
 print('An exception flew by!')
 # raise
else:
 Rest of the code here...
```



[www.spiraltrain.nl](http://www.spiraltrain.nl)

Exception Handling

117

### Output:

```
$ python raising.py
Enter something --> a
ShortInputException: Input was 1 long, expected at least 3
```

```
$ python raising.py
Enter something --> abc
No exception was raised.
```

Here, we are creating our own exception type. This new exception type is called `ShortInputException`. It has two fields - `length` which is the length of the given input, and `at least` which is the minimum length that the program was expecting.

In the `except` clause, we mention the class of error which will be stored as the variable name to hold the corresponding error/exception object. This is analogous to parameters and arguments in a function call. Within this particular `except` clause, we use the `length` and `at least` fields of the exception object to print an appropriate message to the user.

## User Defined Exceptions

- **Created by deriving classes from standard built-in exceptions:**
  - Exception class may created that is subclassed from `RuntimeError`
  - Useful when more specific information is needed when an exception is caught
- **In try: block:**
  - User-defined exception is raised and caught in the `except` block
  - The variable `e` is used to create an instance of the class `Networkerror`

```
class Networkerror(RuntimeError):
 def __init__(self, arg):
 self.args = arg
```

- **With above class exception is raised as follows:**

```
try:
 raise Networkerror("Bad hostname")
except Networkerror as e:
 print(e.args)
```



[www.spiraltrain.nl](http://www.spiraltrain.nl)

Exception Handling

118

Acquiring a resource in the try block and subsequently releasing the resource in the finally block is a common pattern. Hence, there is also a with statement that enables this to be done in a clean manner:

```
with open("poem.txt") as f:
 for line in f:
 print(line, end='')
```

The output should be same as the previous example. The difference here is that we are using the open function with the with statement - we leave the closing of the file to be done automatically by with open.

What happens behind the scenes is that there is a protocol used by the with statement. It fetches the object returned by the open statement, let's call it "thefile" in this case.

It always calls the `thefile.__enter__` function before starting the block of code under it and always calls `thefile.__exit__` after finishing the block of code.

So the code that we would have written in a finally block should be taken care of automatically by the `__exit__` method. This is what helps us to avoid having to use explicit try..finally statements repeatedly.

## Input and Output

- **Simplest way to produce output is `print` function:**
  - Allows you to pass expressions, separated by commas, enclosed in parentheses

```
print("Python is really a great language,", "isn't it? ")
```
- **Major difference between version 2 and 3 of the language:**
  - In version 3 `print` is a function
- **Standard input default comes from keyboard with `input` function:**
  - Reads line from standard input, converts it to a string and strips trailing newline
- **Python also has File Object methods:**
  - Most of the file manipulation is done using a file object
- **Python also has OS Object Methods:**
  - Methods to process files as well as directories



www.spiraltrain.nl

Python IO

119

The `print` statement in Python 2 becomes a `print()` function in Python 3.

- For basic `print` functionality the only difference is whether or not to use parentheses
- To format printed output, Python 2 uses special syntax while Python 3 uses The keyword arguments `sep` and `end`. `sep` determines the separator used Between arguments to the `print` function (default is space), and `end` determines the final character printed (default is newline)

Python 2: `print "The answer is", 42`  
 Python 3: `print("The answer is", 42)`  
 Output: The answer is 42

Python 2: `print`  
 Python 3: `print()`  
 Output: newline

Python 2: `print "The answer is", # comma suppresses newline`  
`print 42`  
 Python 3: `print("The answer is", end=" ")`  
`print(42)`  
 Output: The answer is 42  
 Python 3: `print("01", "12", "1981", sep="-")`  
 Output: 01-12-1981

## IO Module

- **Core of the I/O system is implemented in io library module**
- **Consists of a collection of different I/O classes:**
  - `FileIO`
  - `BufferedReader`
  - `BufferedWriter`
  - `BufferedRWPair`
  - `BufferedRandom`
  - `TextIOWrapper`
  - `BytesIO`
  - `StringIO`
- **Each class implements a different kind of I/O**
- **Classes get layered to add features**

[www.spiraltrain.nl](http://www.spiraltrain.nl)

Python IO

120

Python 3 fully embraces Unicode text, a change that affects almost every part of the language and library. For instance, all text strings are now Unicode, as is Python source code. When you open files in text mode, Unicode is always assumed—even if you don't specify anything in the way of a specific encoding (with UTF-8 being the usual default).

Borrowing from Java, Python 3 takes a completely different approach to file I/O. Although you still open files using the familiar `open()` function, the kind of “file” object that you get back is now part of a layered I/O stack. For example:

```
>>> f = open("foo")>>> f<io.TextIOWrapper object at 0x383950>
>>>
```

So, what is this `TextIOWrapper`? It's a class that wraps a file of type `BufferedReader`, which in turns wraps a file of type `FileIO`. Yes, Python 3 has a full assortment of various I/O classes for raw I/O, buffered I/O, and text decoding that get hooked together in various configurations. Although it's not a carbon copy of what you find in Java, it has a similar flavor.



## Opening Files

- **open function:**

- Opens a file to be able to read or write a file later
- Creates file object which can be used to call other support methods

```
file_object = open(file ,mode=r, buffering=-1, encoding=None)
```

- **file:**

- String value that contains the name of the file to be accessed

- **mode:**

- Determines mode in which the file has to be opened, read, write, append etc.
- `t` for text mode is default and optional, `b` is for byte mode

- **buffering:**

- If buffering value is set to 0, no buffering will take place
- If buffering value is 1, line buffering will be used (in text mode)
- If buffering > 1: buffering performed with indicated buffer size
- Default (-1): fixed-sized buffering (heuristic or `io.DEFAULT_BUFFER_SIZE`; line if tty)

www.spiraltrain.nl

Python I/O

121

Open file and return a corresponding stream. If the file cannot be opened, an `IOError` is raised.

`file` is either a string or bytes object giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped.

`mode` is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode. Other common values are `'w'` for writing (truncating the file if it already exists), and `'a'` for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent. (For reading and writing raw bytes use binary mode and leave encoding unspecified.)

Python distinguishes between binary and text I/O. Files opened in binary mode (including `'b'` in the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when `'t'` is included in the mode argument), the contents of the file are returned as `str`, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

### Note:

Python doesn't depend on the underlying operating system's notion of text files; all the the processing is done by Python itself, and is therefore platform-independent.

`buffering` is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works

## File Open Modes

- **Python distinguishes between binary and text I/O:**
  - In binary mode content is returned as bytes objects without any decoding
  - In text mode as `str`, bytes decoded using given or platform-specific encoding

| Modes            | Description                                                                                                                                                                                                  |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>r</code>   | Opens file for reading only. File pointer is placed at beginning of the file. Default mode                                                                                                                   |
| <code>rb</code>  | Opens file for reading only in binary format. File pointer is placed at beginning of file                                                                                                                    |
| <code>r+</code>  | Opens file for both reading and writing. File pointer will be at the beginning of the file                                                                                                                   |
| <code>rb+</code> | Opens file for both reading and writing in binary format. File pointer will be at the beginning of the file                                                                                                  |
| <code>w</code>   | Opens file for writing only. Overwrites file if file exists. If file does not exist, creates a new file for writing                                                                                          |
| <code>wb</code>  | Opens file for writing only in binary format. Overwrites file if file exists. If file does not exist, creates new file for reading and writing                                                               |
| <code>w+</code>  | Opens file for writing and reading. Overwrites existing file if file exists. If file does not exist, creates new file for reading and writing                                                                |
| <code>wb+</code> | Opens file for both writing and reading in binary format. Overwrites existing file if file exists. If the file does not exist, creates a new file for reading and writing                                    |
| <code>a</code>   | Opens a file for appending. File pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.               |
| <code>ab</code>  | Opens a file for appending in binary format. File pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates new file for writing |

www.spiraltrain.nl

Python I/O

122

mode is an optional string that specifies the mode in which the file is opened. It defaults to 'r' which means open for reading in text mode. Other common values are 'w' for writing (truncating the file if it already exists), and 'a' for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent. (For reading and writing raw bytes use binary mode and leave encoding unspecified.) The available modes are:

| Character | Meaning                                                         |
|-----------|-----------------------------------------------------------------|
| 'r'       | open for reading (default)                                      |
| 'w'       | open for writing, truncating the file first                     |
| 'a'       | open for writing, appending to the end of the file if it exists |
| 'b'       | binary mode                                                     |
| 't'       | text mode (default)                                             |
| '+'       | open a disk file for updating (reading and writing)             |

The default mode is 'rt' (open for reading text). For binary random access, the mode 'w+b' opens and truncates the file to 0 bytes, while 'r+b' opens the file without truncation.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn't. Files opened in binary mode (including 'b' in the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when 't' is included in the mode argument), the contents of the file are returned as unicode strings, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

## File Object Attributes

- **FileIO object represents raw unbuffered binary I/O**
- **File object has attributes with information about it**
- `file.closed` : Returns true if file is closed, false otherwise
- `file.mode` : Returns access mode with which file was opened
- `file.name` : Returns name of the file

- **Example:**

```
fo = open("test.txt", "wb")
print("Name of the file: ", fo.name)
print("Closed: ", fo.closed)
print("Opening mode: ", fo.mode)
```

- **Result:**

```
Name of the file: test.txt
Closed or not: False
Opening mode: wb
```



www.spiraltrain.nl

Python I/O

123

```
FileIO(name, mode='r', closed=True)
```

FileIO represents an OS-level file containing bytes data. It implements the RawIOBase interface (and therefore the IOBase interface, too).

The name can be one of two things:

a character string or bytes object representing the path to the file which will be opened;

an integer representing the number of an existing OS-level file descriptor to which the resulting FileIO object will give access.

The mode can be 'r', 'w' or 'a' for reading (default), writing, or appending. The file will be created if it doesn't exist when opened for writing or appending; it will be truncated when opened for writing. Add a '+' to the mode to allow simultaneous reading and writing.

The `read()` (when called with a positive argument), `readinto()` and `write()` methods on this class will only make one system call.

In addition to the attributes and methods from IOBase and RawIOBase, FileIO provides the following data attributes and methods:

`mode`

The mode as given in the constructor.

`name`

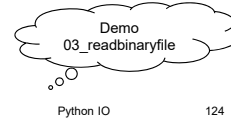
The file name. This is the file descriptor of the file when no name is given in the constructor.

## Reading Binary Files

- **Opening a file in binary mode is simple but subtle:**
  - Reading bytes not strings, so there's no conversion for Python to do
- **Difference from opening it in text mode:**
  - b mode parameter
  - No encoding attribute
- **read method reads whole file or a number of bytes:**
  - tell moves file pointer to the beginning of the file
  - seek moves file pointer to a certain position in the file

```
fr = open('trilobyte.jpg', mode='rb')
fr.tell()
data = fr.read(3) => read three bytes
print(type(data)) => type is bytes
fr.tell()
fr.seek(0)
data = fr.read()
print(data)
```

www.spiraltrain.nl



Opening a file in binary mode is simple but subtle. The only difference from opening it in text mode is that the mode parameter contains a 'b' character.

The stream object you get from opening a file in binary mode has many of the same attributes, including mode, which reflects the mode parameter you passed into the open() function.

Binary stream objects also have a name attribute, just like text stream objects.

Here's one difference, though: a binary stream object has no encoding attribute. That makes sense, right? You're reading (or writing) bytes, not strings, so there's no conversion for Python to do. What you get out of a binary file is exactly what you put into it, no conversion necessary.

Like text files, you can read binary files a little bit at a time. But there's a crucial difference.....you're reading bytes, not strings. Since you opened the file in binary mode, the read() method takes the number of bytes to read, not the number of characters.

That means that there's never an unexpected mismatch between the number you passed into the read() method and the position index you get out of the tell() method. The read() method reads bytes, and the seek() and tell() methods track the number of bytes read. For binary files, they'll always agree.

The io module defines the StringIO class that you can use to treat a string in memory as a file. To create a stream object out of a string, create an instance of the io.StringIO() class and pass it the string you want to use as your "file" data. Now you have a stream object, and you can do all sorts of stream-like things with it. Calling the read() method "reads" the entire "file," which in the case of a StringIO object simply returns the original string. Just like a real file, calling the read() method again returns an empty string. You can explicitly seek to the beginning of the string, just like seeking through a real file, by using the seek() method of the StringIO object. You can also read the string in chunks, by passing a size parameter to the read() method.

## Writing Binary Files

- **Open binary files for writing without encoding attribute:**

```
fw = open('binaryfile04.txt', mode='wb')
```

- **Use pack method of struct module to prepare buffer for writing:**

- Format string determines how packing takes place

```
nr1 = 1; nr2 = 2; result = nr1 + nr2
buffer = pack("iii", nr1, nr2, result)
fw.write(buffer)
```

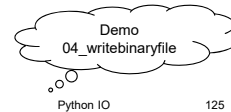
- **Data must be read like it is written using same format string:**

```
fr = open('binaryfile04.txt', mode='rb')
buffer = fr.read()
data = unpack("iii", buffer)
```

- **Returned data is a tuple that can be accessed as follows:**

```
num1 = data[0], num2 = data[1], res = data[2]
```

www.spiraltrain.nl



```
struct.pack(fmt, v1, v2, ...)
```

Return a bytes object containing the values *v1*, *v2*, ... packed according to the format string *fmt*. The arguments must match the values required by the format exactly.

```
struct.unpack(fmt, buffer)
```

Unpack from the buffer *buffer* (presumably packed by `pack(fmt, ...)`) according to the format string *fmt*. The result is a tuple even if it contains exactly one item. The buffer must contain exactly the amount of data required by the format (`len(bytes)` must equal `calcsz(fmt)`).

```
struct.pack_into(fmt, buffer, offset, v1, v2, ...)
```

Pack the values *v1*, *v2*, ... according to the format string *fmt* and write the packed bytes into the writable buffer *buffer* starting at position *offset*. Note that *offset* is a required argument.

```
struct.unpack_from(fmt, buffer, offset=0)
```

Unpack from *buffer* starting at position *offset*, according to the format string *fmt*. The result is a tuple even if it contains exactly one item. *buffer* must contain at least the amount of data required by the format (`len(buffer[offset:])` must be at least `calcsz(fmt)`).

```
struct.calcsize(fmt)
```

Return the size of the struct (and hence of the bytes object produced by `pack(fmt, ...)`) corresponding to the format string *fmt*.

## Writing Text Files

- **Open file for writing specifying mode and encoding:**

```
fw = open('textfilepolish.txt', mode='w', encoding='UTF-8')
```

- **write method writes any string to an open file:**
  - Parameter is the content to be written into the opened file
  - Does not add a newline character ('\n') to the end of the string

```
s = 'Polish text: aćęłńóśźżĄĆĘŁŃÓŚŹŻ'
fw.write(s)
```

- **Always close the file when you are done:**

```
fw.close()
```

- **File textfilepolish.txt with following content is created:**

```
Polish text: Ą...Ą+Ą™Ą, Ą,, Ą³Ą>Ą°Ą¼Ą,, Ą†ĄĄ♦ĄfĄ"ĄšĄ³Ą»
```

You can write to files in much the same way that you read from them. First you open a file and get a stream object, then you use methods on the stream object to write data to the file, then you close the file.

To open a file for writing, use the `open()` function and specify the write mode. There are two file modes for writing:

“Write” mode will overwrite the file. Pass `mode='w'` to the `open()` function.

“Append” mode will add data to the end of the file. Pass `mode='a'` to the `open()` function.

Either mode will create the file automatically if it doesn’t already exist, so there’s never a need for any sort of fiddly “if the file doesn’t exist yet, create a new empty file just so you can open it for the first time” function. Just open a file and start writing.

You should always close a file as soon as you’re done writing to it, to release the file handle and ensure that the data is actually written to disk. As with reading data from a file, you can call the stream object’s `close()` method, or you can use the `with` statement and let Python close the file for you. I bet you can guess which technique I recommend.

## Reading Text Files

- **Open file for reading specifying mode and encoding:**

```
fr = open('textfilepolish.txt', mode='r', encoding='UTF-8')
```

- **read method reads a string from an open file:**

- Parameter is the number of bytes to be read from the opened file

```
fileObject.read([count]);
```

- **read() starts reading from the beginning of the file:**

- If count is missing then it tries to read as much as possible, may be until end of file

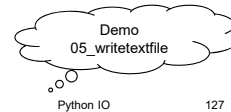
- **Example of reading and closing file** `textfilepolish.txt`:

```
data = fr.read()
print(data)
fr.close()
```

- **Result:**

Polish text: aćęłńóśźżĄĆĘŁŃÓŚŹŻ

[www.spiraltrain.nl](http://www.spiraltrain.nl)



127

Before you can read from a file, you need to open it. Opening a file in Python couldn't be easier:

```
a_file = open('examples/chinese.txt', encoding='utf-8')
```

Python has a built-in `open()` function, which takes a filename as an argument. Here the filename is `'examples/chinese.txt'`. There are five interesting things about this filename:

It's not just the name of a file; it's a combination of a directory path and a filename. A hypothetical file-opening function could have taken two arguments — a directory path and a filename — but the `open()` function only takes one. In Python, whenever you need a “filename,” you can include some or all of a directory path as well.

The directory path uses a forward slash, but I didn't say what operating system I was using. Windows uses backward slashes to denote subdirectories, while Mac OS X and Linux use forward slashes. But in Python, forward slashes always Just Work, even on Windows.

The directory path does not begin with a slash or a drive letter, so it is called a *relative path*. Relative to what, you might ask? Patience, grasshopper.

It's a string. All modern operating systems (even Windows!) use Unicode to store the names of files and directories. Python 3 fully supports non-ASCII pathnames.

It doesn't need to be on your local disk. You might have a network drive mounted. That “file” might be a figment of an entirely virtual filesystem. If your computer considers it a file and can access it as a file, Python can open it.

## Closing Files

- **close** method flushes unwritten data and closes file object:
  - No more writing can be done after calling `close`
  - File is closed automatically when reference file object is reassigned to another file
- **Syntax:**

```
fileObject.close();
```
- **Example:**

```
fo = open("foo.txt", "wb")
print("Name of the file: ", fo.name)
fo.close()
```
- **Result:**

```
Name of the file: foo.txt
```
- **Good practice to use** `close()` **method to close a file**

www.spiraltrain.nl

Python IO

128

Stream objects have an explicit `close()` method, but what happens if your code has a bug and crashes before you call `close()`? That file could theoretically stay open for much longer than necessary. While you're debugging on your local computer, that's not a big deal. On a production server, maybe it is.

Python 2 had a solution for this: the `try..finally` block. That still works in Python 3, and you may see it in other people's code or in older code that was ported to Python 3. But Python 2.6 introduced a cleaner solution, which is now the preferred solution in Python 3: the `with` statement.

```
with open('examples/chinese.txt', encoding='utf-8') as a_file:
 a_file.seek(17)
 a_character = a_file.read(1)
 print(a_character)
```

This code calls `open()`, but it never calls `a_file.close()`. The `with` statement starts a code block, like an `if` statement or a `for` loop. Inside this code block, you can use the variable `a_file` as the stream object returned from the call to `open()`. All the regular stream object methods are available — `seek()`, `read()`, whatever you need. When the `with` block ends, Python calls `a_file.close()` automatically.

Here's the kicker: no matter how or when you exit the `with` block, Python will close that file... even if you "exit" it via an unhandled exception. That's right, even if your code raises an exception and your entire program comes to a screeching halt, that file will get closed. Guaranteed.



## File Positions

- `tell` **method retrieves current position within the file:**
  - Next read or write will occur at that many bytes from the beginning of the file
- `seek(offset[, from])` **method changes the current file position:**
  - Offset argument indicates the number of bytes to be moved
  - From argument specifies reference position from where bytes are to be moved
- **If from is set to 0:**
  - Use beginning of the file as the reference position
- **If from is set to 1:**
  - Use current position as the reference position
- **If from is set to 2:**
  - Use end of the file as the reference position

After you open a file for reading, you'll probably want to read from it at some point.

```
>>> a_file = open('examples/chinese.txt', encoding='utf-8')
>>> a_file.read()
'Dive Into Python 是为有经验的程序员编写的一本 Python 书。'
>>> a_file.read()
''
```

Once you open a file (with the correct encoding), reading from it is just a matter of calling the stream object's `read()` method. The result is a string.

Perhaps somewhat surprisingly, reading the file again does not raise an exception. Python does not consider reading past end-of-file to be an error; it simply returns an empty string.

What if you want to re-read a file?

```
>>> a_file.read()
''
>>> a_file.seek(0)
0
>>> a_file.read(16)
'Dive Into Python'
>>> a_file.read(1)
' '
>>> a_file.read(1)
'是'
>>> a_file.tell()
20
```

## Example File Positioning

- Consider file `foo.txt` that was created before:

```
fo = open("foo.txt", "r+")
str = fo.read(10);
print("Read String is: ", str)
Check current position
position = fo.tell();
print("Current file position: ", position)
Reposition pointer at the beginning once again
position = fo.seek(0, 0);
str = fo.read(10);
print("Again read String is: ", str)
fo.close()
```

- Result:**

```
Read String is: Python is
Current file position: 10
Again read String is: Python is
```

[www.spiraltrain.nl](http://www.spiraltrain.nl)

Python IO

130

Since you're still at the end of the file, further calls to the stream object's `read()` method simply return an empty string. The `seek()` method moves to a specific byte position in a file.

The `read()` method can take an optional parameter, the number of characters to read. If you like, you can even read one character at a time.

$16 + 1 + 1 = \dots 20?$

Let's try that again.

```
>>> a_file.seek(17)
17
>>> a_file.read(1)
'是'
>>> a_file.tell()
20
```

Move to the 17th byte.

Read one character.

Now you're on the 20th byte.

Do you see it yet? The `seek()` and `tell()` methods always count bytes, but since you opened this file as text, the `read()` method counts characters. Chinese characters require multiple bytes to encode in utf-8. The English characters in the file only require one byte each, so you might be misled into thinking that the `seek()` and `read()` methods are counting the same thing. But that's only true for some characters.

## Renaming and Deleting Files

- **Files can be renamed and deleted:**
  - `os` module has methods for file-processing operations like renaming and deleting
  - `os` module must first be imported before methods can be called
- **rename method takes two arguments:**
  - Current filename and new filename

```
os.rename(current_file_name, new_file_name)
```

- **Rename an existing file from `test1.txt` to `test2.txt`:**

```
import os
os.rename("test1.txt", "test2.txt")
```

- **delete method deletes files supplied as argument:**

```
os.delete(file_name)
```

- **Delete an existing file `test2.txt`:**

```
import os
os.remove("text2.txt")
```

[www.spiraltrain.nl](http://www.spiraltrain.nl)

Python IO

131

This module provides a portable way of using operating system dependent functionality. If you just want to read or write a file see `open()`, if you want to manipulate paths, see the `os.path` module, and if you want to read all the lines in all the files on the command line see the `fileinput` module. For creating temporary files and directories see the `tempfile` module, and for high-level file and directory handling see the `shutil` module.

Notes on the availability of these functions:

The design of all built-in operating system dependent modules of Python is such that as long as the same functionality is available, it uses the same interface; for example, the function `os.stat(path)` returns stat information about path in the same format (which happens to have originated with the POSIX interface).

Extensions peculiar to a particular operating system are also available through the `os` module, but using them is of course a threat to portability.

All functions accepting path or file names accept both bytes and string objects, and result in an object of the same type, if a path or file name is returned.

Note:

If not separately noted, all functions that claim “Availability: Unix” are supported on Mac OS X, which builds on a Unix core.

An “Availability: Unix” note means that this function is commonly found on Unix systems. It does not make any claims about its existence on a specific operating system.

If not separately noted, all functions that claim “Availability: Unix” are supported on Mac OS X, which builds on a Unix core.

Note:

All functions in this module raise `OSError` in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system.

## Creating and Deleting Directories

- **Files are in directories and Python can handle directories too:**

- os module has methods to create, remove, and change directories

- **mkdir()** **method creates directory in the current directory:**

- Argument is name of the directory to be created

- **Syntax:** `os.mkdir("newdir")`

- **Example to create a directory test in current directory:**

```
import os
os.mkdir("test")
```

- **rmdir()** **method deletes the directory passed as an argument:**

- Before removing a directory, all the contents in it should be removed

- **Syntax:** `os.rmdir('dirname')`

- **Example to remove "/tmp/test" directory:**

```
import os
os.rmdir("/tmp/test")
```

www.spiraltrain.nl

Python IO

132

```
os.listdir(path='.')
```

Return a list containing the names of the entries in the directory given by path (default: '.'). The list is in arbitrary order. It does not include the special entries '.' and '..' even if they are present in the directory.

This function can be called with a bytes or string argument, and returns filenames of the same datatype.

```
os.makedirs(path, mode=0o777, exist_ok=False)
```

Recursive directory creation function. Like `mkdir()`, but makes all intermediate-level directories needed to contain the leaf directory. If the target directory with the same mode as specified already exists, raises an `OSError` exception if `exist_ok` is `False`, otherwise no exception is raised. If the directory cannot be created in other cases, raises an `OSError` exception. The default mode is `0o777` (octal). On some systems, mode is ignored. Where it is used, the current umask value is first masked out.

`makedirs()` will become confused if the path elements to create include `pardir`.

This function handles UNC paths correctly.

```
os.remove(path)
```

Remove (delete) the file path. If path is a directory, `OSError` is raised; see `rmdir()` below to remove a directory. This is identical to the `unlink()` function documented below. On Windows, attempting to remove a file that is in use causes an exception to be raised; on Unix, the directory entry is removed but the storage allocated to the file is not made available until the original file is no longer in use.

```
os.rename(src, dst)
```

Rename the file or directory `src` to `dst`. If `dst` is a directory, `OSError` will be raised. On Unix, if `dst` exists and is a file, it will be replaced silently if the user has permission. The operation may fail on some Unix flavors if `src` and `dst` are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement). On Windows, if `dst` already exists, `OSError` will be raised even if it is a file; there may be no way to implement an atomic rename when `dst` names an existing file.

Availability: Unix, Windows.

## Directory Methods

- `chdir()` **changes the current directory:**
  - Argument is the name of the directory that you want to make the current directory

- **Syntax:**

```
os.chdir("newdir")
```

- **Example changing a directory to `"/home/newdir"`:**

```
import os
os.chdir("/home/newdir")
```

- `getcwd()` **displays the current working directory**

- **Syntax:**

```
os.getcwd()
```

- **Example to give location of the current directory:**

```
import os
os.getcwd()
```

[www.spiraltrain.nl](http://www.spiraltrain.nl)

Python IO

133

```
os.system(command)
```

Execute the command (a string) in a subshell. This is implemented by calling the Standard C function `system()`, and has the same limitations. Changes to `sys.stdin`, etc. are not reflected in the environment of the executed command. If command generates any output, it will be sent to the interpreter standard output stream.

On Unix, the return value is the exit status of the process encoded in the format specified for `wait()`. Note that POSIX does not specify the meaning of the return value of the C `system()` function, so the return value of the Python function is system-dependent.

On Windows, the return value is that returned by the system shell after running command. The shell is given by the Windows environment variable `COMSPEC`: it is usually `cmd.exe`, which returns the exit status of the command run; on systems using a non-native shell, consult your shell documentation.

The `subprocess` module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using this function. See the [Replacing Older Functions with the subprocess Module](#) section in the `subprocess` documentation for some helpful recipes.

Availability: Unix, Windows.

```
os.chdir(path)
```

Change the current working directory to *path*.

Availability: Unix, Windows.

```
os.fchdir(fd)
```

Change the current working directory to the directory represented by the file descriptor *fd*. The descriptor must refer to an opened directory, not an open file.

Availability: Unix.

```
os.getcwd()
```

Return a string representing the current working directory.

## Iteration

- **Python for statement can iterate over many kinds of objects**
- **Iterate over a collection of items:**

```
for x in [1,4,5,10]:
 print(x) # 1 4 5 10
```

- **Iteration over dictionary given the keys:**

```
prices = { 'GOOGLE': 490.10, 'IBM': 145.23, 'YAHOO': 21.71 }
for key in prices:
 print(key) # GOOGLE IBM YAHOO
```

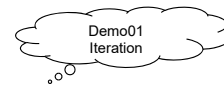
- **Iteration over a string gives characters in string :**

```
s = "Mars!"
for c in s:
 print(c) # M a r s !
```

- **Iteration over a file gives the lines:**

```
for line in open("realprogrammers.txt"):
 print(line)
```

[www.spiraltrain.nl](http://www.spiraltrain.nl)



Generators

134

In computer science, an iterator is an object that allows a programmer to traverse through all the elements of a collection regardless of its specific implementation.

Iterators in Python are a fundamental part of the language and in many cases go unseen as they are implicitly used in the for (foreach) statement, in list comprehensions, and in generator expressions.

"Iterators are the secret sauce of Python 3. They're everywhere, underlying everything, always just out of sight. Comprehensions are just a simple form of iterators. Generators are just a simple form of iterators. A function that yields values is a nice, compact way of building an iterator without building an iterator." - Mark Pilgrim

All of Python's standard built-in sequence types support iteration, as well as many classes which are part of the standard library.

The for loop works on any iterable object. Actually, this is true of all iteration tools that scan objects from left to right in Python including for loops, the list comprehensions, and the map built-in function, etc.

Though the concept of iterable objects is relatively recent in Python, it has come to permeate the language's design. Actually, it is a generalization of the sequences. An object is iterable if it is either a physically stored sequence or an object that produces one result at a time in the context of an iteration tool like a for loop.

## Consuming Iterables

- **Many functions can consume an iterable object:**

- **Reductions:**

```
s = [1,4,5,10]
sum(s)
min(s)
max(s)
```

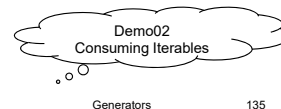
- **Constructors:**

```
l = list(s)
t = tuple(s)
st = set(s)
```

- **in operator:**

```
item in s
```

- **And many others in the library**



www.spiraltrain.nl

Generators

135

As a way of understanding the file iterator, we'll look at how it works with a file. Open file objects have `readline()` method. This reads one line each time we call `readline()`, we advance to the next line. At the end of the file, an empty string is returned. We detect it to get out of the loop.

```
>>> f = open('C:\\workspace\\Masters.txt')
>>> f.readline()
'Michelangelo Buonarroti \n'
>>> f.readline()
'Pablo Picasso\n'
>>> f.readline()
'Rembrandt van Rijn \n'
>>> f.readline()
'Leonardo Da Vinci \n'
>>> f.readline()
'Claude Monet \n'
>>> f.readline()
'\n'
Returns an empty string at end-of-file
>>> f.readline()
''
>>>
```

But files also have a `__next__()` method that has an identical effect. It returns the next line from a file each time it is called. The only difference is that `__next__()` method raises a built-in `StopIteration` exception at end-of-file instead of returning an empty string

## Iteration Protocol

- **Protocol implemented by objects that you can iterate over:**

```
items = [1, 4, 5]
it = items.__iter__()
print(next(it))
print(next(it))
print(next(it))
print(next(it))
Traceback (most recent call last):
File "D:\..\src\demo03_iteration_protocol.py", line 7, in <module>
 print(next(it))
StopIteration
```

- **Underneath the covers of for statement:** for x in obj:

```
_iter = obj.__iter__() # Get iterator object
while True:
 try:
 x = next(_iter) # Get next item
 except StopIteration: # No more items
 break
 statements
```



- **Object supporting \_\_iter\_\_() and \_\_next\_\_() is iterable**

www.spiraltrain.nl

Generators

136

We have a built-in next() function for manual iteration. The next() function automatically calls an object's \_\_next\_\_() method. For an object X, the call next(X) is the same as X.\_\_next\_\_() but simpler.

```
>>> f = open('C:\\workspace\\Masters.txt')
>>> f.__next__()
'Michelangelo Buonarroti \n'
>>> f.__next__()
'Pablo Picasso\n'
>>>
>>> f = open('C:\\workspace\\Masters.txt')
>>> next(f)
'Michelangelo Buonarroti \n'
>>> next(f)
'Pablo Picasso\n'
>>>
```

When the for loop begins, it obtains an iterator from the iterable object by passing it to the iter built-in function. This object returned by iter has the required the \_\_next\_\_() method. Let's look at the internals of this through for loop with lists. For Python versions < 3, we may want to use next(iterObj) instead of iterobj.\_\_next\_\_().



## Supporting Iteration

- **User-defined objects can support iteration:**

- Object implement should implement `__iter__()` and `__next__()`

```
class Countdown(object):
 def __init__(self, start):
 self.count = start
 def __iter__(self):
 return self
 def __next__(self):
 if self.count <= 0:
 raise StopIteration
 r = self.count
 self.count -= 1
 return r
```

- **Now Countdown can be iterated as follows:**

```
c = Countdown(5)
for i in c:
 print(i)
```



www.spiraltrain.nl

Generators

137

We've looked at the iterators for files and lists. How about the others such as dictionaries? To step through the keys of a dictionary is to request its keys list explicitly:

```
>>> D = {'a':97, 'b':98, 'c':99}
>>> for k in D.keys():
 print(k, D[k])

a 97
c 99
b 98
```

Dictionaries have an iterator that automatically returns one key at a time in an iteration context:

```
>>> iterObj = iter(D)
>>> next(iterObj)
'a'
>>> next(iterObj)
'c'
>>> next(iterObj)
'b'
>>> next(iterObj)
Traceback (most recent call last):
 File ...,
 next(iterObj)
StopIteration
>>>
```

## Generators

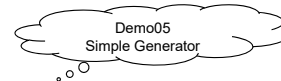
- **Functions producing sequence of results instead of single value:**

```
def countdown(n):
 while n > 0:
 yield n
 n -= 1
for i in countdown(5):
 print(i)
```

- **Instead of returning a value:**
  - Generates series of values using `yield` statement
- **Behavior is quite different from normal function:**
  - Calling a generator function creates an generator object
  - However, it does not start running the function

```
x = countdown(10)
print(x) # <generator object at 0x58490>
```

- **Notice that no output was produced**



[www.spiraltrain.nl](http://www.spiraltrain.nl)

Generators

138

In computer science, a generator is a special routine that can be used to control the iteration behavior of a loop.

A generator is very similar to a function that returns an array, in that a generator has parameters, can be called, and generates a sequence of values. However, instead of building an array containing all the values and returning them all at once, a generator yields the values one at a time, which requires less memory and allows the caller to get started processing the first few values immediately. In short, a generator looks like a function but behaves like an iterator.

Python provides tools that produce results only when needed:

### Generator functions

They are coded as normal `def` but use `yield` to return results one at a time, suspending and resuming.

### Generator expressions

These are similar to the list comprehensions. But they return an object that produces results on demand instead of building a result list.

Because neither of them constructs a result list all at once, they save memory space and allow computation time to be split by implementing the iteration protocol.

### Generator Functions: `yield` vs. `return`

We can write functions that send back a value and later be resumed by picking up where they left off. Such functions are called generator functions because they generate a sequence of values over time.

Generator functions are not much different from normal functions and they use `def`s. When created, however, they are automatically made to implement the iteration protocol so that they can appear in iteration contexts.

## Generator Functions

- **Function only executes on `next()`:**


```
x = countdown(6)
print(x) # <generator object countdown at 0x00D5B4E0>
```

- **`yield` produces a value, but suspends the function:**
  - Function resumes on next call to `next()`

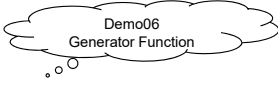
- **Counting down from 10:**

```
print(next(x)) #6
print(next(x)) #5
print(next(x)) #4
print(next(x)) #3
print(next(x)) #2
print(next(x)) #1
print(next(x)) # Exception => StopIteration
```

Function starts  
executing here



Demo06  
Generator Function



Normal functions return a value and then exit. But generator functions automatically suspend and resume their execution. Because of that, they are often a useful alternative to both computing an entire series of values up front and manually saving and restoring state in classes. Because the state that generator functions retain when they are suspended includes their local scope, their local variables retain information and make it available when the functions are resumed.

The primary difference between generator and normal functions is that a generator yields a value, rather than returns a value. The `yield` suspends the function and sends a value back to the caller while retains enough state to enable the function immediately after the last `yield` run. This allows the generator function to produce a series of values over time rather than computing them all at once and sending them back in a list.

Generators are closely bound up with the iteration protocol. Iterable objects define a `__next__()` method which either returns the next item in the iterator or raises the special `StopIteration` exception to end the iteration. An object's iterator is fetched with the `iter` built-in function.

The `for` loops use this iteration protocol to step through a sequence or value generator if the protocol is suspended. Otherwise, iteration falls back on repeatedly indexing sequences.

To support this protocol, functions with `yield` statement are compiled specially as generators. They return a generator object when they are called. The returned object supports the iteration interface with an automatically created `__next__()` method to resume execution. Generator functions may have a return simply terminates the generation of values by raising a `StopIteration` exceptions after any normal function exit.

## Convenient Iterator

- **Generator function mainly more convenient to write an iterator:**
  - Don't worry about iterator protocol with `__next__()` and `__iter__()`
  - It just works
- **Generators versus iterators:**
  - Generator function is slightly different than an object that supports iteration
  - Generator is a one-time operation:
    - Can iterate over the generated data once
    - If you want to do it again, you have to call the generator function again
  - This is different than a list:
    - You can iterate over a list as many times as you want

The net effect is that generator functions, coded as `def` statements containing `yield` statement, are automatically made to support the iteration protocol and thus may be used any iteration context to produce results over time and on demand. Let's look at the interactive example below:

```
>>> def create_counter(n):
 print('create_counter()')
 while True:
 yield n
 print('increment n')
 n += 1

>>> c = create_counter(2)
>>> c
<generator object create_counter at 0x03004B48>
>>> next(c)
create_counter()
2
>>> next(c)
increment n
3
>>> next(c)
increment n
4
>>>
```

## Generator Expression

- **Generated version of a list comprehension:**

```
li = [1,2,3,4]
generator expression, watch the parenthesis
ge = (2*x for x in li)
print(ge) # <generator object <genexpr> at 0x0085B4E0>
print(next(ge)) # prints 2
for i in ge: # prints 4, 6 and 8
 print(i)
```

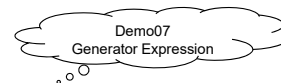
- **Loops over sequence of items and applies operation to each:**

- However, results are produced one at a time using a generator

- **Important differences from a list comprehension:**

- Generator Expression does not construct a list
- Its only useful purpose is iteration and once consumed it can't be reused

```
list comprehension, watch the block brackets
lc = [2*x for x in li]
print(lc) # [2, 4, 6, 8]
```



www.spiraltrain.nl

Generators

141

Here are the things happening in the code:

The presence of the `yield` keyword in `create_counter()` means that this is not a normal function. It is a special kind of function which generates values one at a time. We can think of it as a resumable function. Calling it will return a generator that can be used to generate successive values of `n`.

To create an instance of the `create_counter()` generator, just call it like any other function. Note that this does not actually execute the function code. We can tell this because the first line of the `create_counter()` function calls `print()`, but nothing was printed from the line:

```
>>> c = create_counter(2)
```

The `create_counter()` function returns a generator object.

The `next()` function takes a generator object and returns its next value. The first time we call `next()` with the counter generator, it executes the code in `create_counter()` up to the first `yield` statement, then returns the value that was yielded. In this case, that will be 2, because we originally created the generator by calling `create_counter(2)`.

Repeatedly calling `next()` with the same generator object resumes exactly where it left off and continues until it hits the next `yield` statement. All variables, local state, &c. are saved on `yield` and restored on `next()`. The next line of code waiting to be executed calls `print()`, which prints increment `n`. After that, the statement `n += 1`. Then it loops through the while loop again, and the first thing it hits is the statement `yield n`, which saves the state of everything and returns the current value of `n` (now 3).

The second time we call `next(c)`, we do all the same things again, but this time `n` is now 4.

Since `create_counter()` sets up an infinite loop, we could theoretically do this forever, and it would just keep incrementing `n` and spitting out values.

## Generator Expression Syntax

- **General syntax:**

```
(expression for i in s if cond1
 for j in t if cond2
 ...
 if condfinal)
```

- **Resolves to:**

```
for i in s:
 if cond1:
 for j in t:
 if cond2:
 ...
 if condfinal: yield expression
```

- **If used as a single function argument:**

- Parenthesis on a generator expression can be dropped

```
sum(x*x for x in s)
```

↑  
Generator expression

www.spiraltrain.nl

Generators

142

The generator function in the following example generated the cubics of numbers over time:

```
>>> def cubic_generator(n):
 for i in range(n):
 yield i ** 3
```

```
>>>
```

The function yields a value and so returns to its caller each time through the loop. When it is resumed, its prior state is restored and control picks up again after the yield statement. When it's used in a for loop, control returns to the function after its yield statement each time through the loop:

```
>>> for i in cubic_generator(5):
 print(i, end=': ') # Python 3.0
 #print i, # Python 2.x
0: 1: 8: 27: 64:
>>>
```

If we use return instead of yield, the result is:

```
>>> def cubic_generator(n):
 for i in range(n):
 return i ** 3
>>> for i in cubic_generator(5):
 print(i, end=': ') #Python 3.0
Traceback (most recent call last):
 File "", line 1, in
 for i in cubic_generator(5):
TypeError: 'int' object is not iterable
```

## Building Blocks

- **Generator functions:**

```
def countdown(n):
 while n > 0:
 yield n
 n -= 1
```

- **Generator expressions:**

```
squares = (x*x for x in s)
```

- **In both cases:**

- Get object that generates values which are typically consumed in a for loop

Here is an example of using generator and yield.

```
>>> # Fibonacci version 1
>>> def fibonacci():
 Limit = 10
 count = 0
 a, b = 0, 1
 while True:
 yield a
 a, b = b, a+b
 if (count == Limit):
 break
 count += 1

>>>
>>> for n in fibonacci():
 print(n, end=' ')

0 1 1 2 3 5 8 13 21 34 55
>>>
```

Because generators preserve their local state between invocations, they're particularly well-suited for complicated, stateful iterators, such as **fibonacci numbers**. The generator returning the Fibonacci numbers using Python's yield statement can be seen below.

Here is another version of Fibonacci:

## Programming Problem

- **Find out how many bytes of data were transferred:**

- Sum up last column of data in huge Apache web server log

```
81.107.39.38 - ... "GET /ply/ HTTP/1.1" 200 7587
81.107.39.38 - ... "GET /favicon.ico HTTP/1.1" 404 133
81.107.39.38 - ... "GET /ply/bookplug.gif HTTP/1.1" 200 23903
81.107.39.38 - ... "GET /ply/ply.html HTTP/1.1" 200 97238
81.107.39.38 - ... "GET /ply/example.html HTTP/1.1" 200 2359
66.249.72.134 - ... "GET /index.html HTTP/1.1" 200 4447
```

- **Each line of log looks like this:**

```
81.107.39.38 - ... "GET /ply/ply.html HTTP/1.1" 200 97238
```

- **Number of bytes is the last column:**

```
bytestr = line.rsplit(None,1)[1]
```

- **Is either a number or a missing value (-):**

```
81.107.39.38 - ... "GET /ply/ HTTP/1.1" 304 -
```

www.spiraltrain.nl

Generators

144

```
>>> # Fibonacci version 2
>>> def fibonacci(max):
 a, b = 0, 1 (1)
 while a < max:
 yield a (2)
 a, b = b, a + b (3)
```

Simple summary for this version:

It starts with 0 and 1, goes up slowly at first, then more and more rapidly. To start the sequence, we need two variables: a starts at 0, and b starts at 1.

a is the current number in the sequence, so yield it.

b is the next number in the sequence, so assign that to a, but also calculate the next value  $a + b$  and assign that to b for later use. Note that this happens in parallel; if a is 3 and b is 5, then  $a, b = b, a + b$  will set a to 5 (the previous value of b) and b to 8 (the sum of the previous values of a and b).

```
>>> for n in fibonacci(500):
 print(n, end=' ')
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>>
>>> list(fibonacci(500))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
>>>
```



## Problem Solutions

- **Non-Generator Solution:**

- Do a simple for-loop, read-by-line and update a sum:

```

wwwlog = open("access-log")
total = 0
for line in wwwlog:
 bytestr = line.rsplit(None,1)[1]
 if bytestr != '-':
 total += int(bytestr)
print("Total", total)

```

- **Generator Solution:**

- Use some generator expressions

```

wwwlog = open("access-log")
bytecolumn = (line.rsplit(None,1)[1] for line in wwwlog)
bytes = (int(x) for x in bytecolumn if x != '-')
print("Total", sum(bytes))

```

- **Notice lesser code and a completely different programming style**

www.spiraltrain.nl

Generators

145

As we can see from the output, we can use a generator like `fibonacci()` in a for loop directly. The for loop will automatically call the `next()` function to get values from the `fibonacci()` generator and assign them to the for loop index variable (`n`). Each time through the for loop, `n` gets a new value from the yield statement in `fibonacci()`, and all we have to do is print it out. Once `fibonacci()` runs out of numbers (a becomes bigger than max, which in this case is 500), then the for loop exits gracefully.

This is a useful idiom: pass a generator to the `list()` function, and it will iterate through the entire generator (just like the for loop in the previous example) and return a list of all the values.

To end the generation of values, functions use either a return with no value or simply allow control to fall off the end of the function body.

To see what's happening inside the for, we can call the generator function directly:

```

>>> x = cubic_generator(5)
>>> x
<generator object cubic_generator at 0x000000000315F678>
>>>

```

## Functional Programming

- **Programming style that is focused on expressions:**
  - Also called expression oriented programming
- **Python has functions and features enabling functional approach:**
  - `map, filter, reduce, lambda`
  - `list comprehension`
  - `map(aFunction, aSequence):`
    - Applies operation to each item and collects the result
  - `filter(aFunction, aSequence):`
    - Extracts each element in the sequence for which function returns True
  - `reduce(aFunction, aSequence):`
    - Reduces list to single value by combining elements via supplied function
    - In `functools` in Python 3.0
  - `lambda:`
    - Small anonymous function consisting of single line

[www.spiraltrain.nl](http://www.spiraltrain.nl)

Comprehensions

146

```
>>> items = [1, 2, 3, 4, 5]
>>>
>>> def sqr(x): return x ** 2
>>> list(map(sqr, items))
[1, 4, 9, 16, 25]
>>>
```

We passed in a user-defined function applied to each item in the list. `map` calls `sqr` on each list item and collects all the return values into a new list. Because `map` expects a function to be passed in, it also happens to be one of the places where `lambda` routinely appears:

```
>>> list(map((lambda x: x **2), items))
[1, 4, 9, 16, 25]
>>>
```

In the short example above, the `lambda` function squares each item in the `items` list.

As shown earlier, `map` is defined like this:

```
map(aFunction, aSequence)
```

While we still use `lambda` as a `aFunction`, we can have a list of functions as `aSequence`:

```
def square(x):
 return (x**2)

def cube(x):
 return (x**3)

funcs = [square, cube]
for r in range(5):
 value = map(lambda x: x(r), funcs)
 print value
```

## Map and Filter

- **Higher-order functions** `map` and `filter` that operate on lists:

```
def square(x):
 return x*x
def even(x):
 return 0 == x % 2
some_list = list(map(square, range(10,20)))
print(some_list)
```

- **Output:** [100, 121, 144, 169, 196, 225, 256, 289, 324, 361]

```
some_list = list(filter(even, range(10,20)))
print(some_list)
```

- **Output:** [10, 12, 14, 16, 18]

```
some_list = list(map(square, filter(even, range(10,20))))
print(some_list)
```

- **Output:** [100, 144, 196, 256, 324]

- **More Pythonic are list comprehensions**



[www.spiraltrain.nl](http://www.spiraltrain.nl)

Comprehensions

147

Because using `map` is equivalent to for loops, with an extra code we can always write a general mapping utility:

```
>>> def mymap(aFunc, aSeq):
 result = []
 for x in aSeq: result.append(aFunc(x))
 return result
>>> list(map(sqr, [1, 2, 3]))
[1, 4, 9]
>>> mymap(sqr, [1, 2, 3])
[1, 4, 9]
>>>
```

Since it's a built-in, `map` is always available and always works the same way. It also has some performance benefit because it is usually faster than a manually coded for loop. On top of those, `map` can be used in more advance way. For example, given multiple sequence arguments, it sends items taken from sequences in parallel as distinct arguments to the function:

```
>>> pow(3,5)
243
>>> pow(2,10)
1024
>>> pow(3,11)
177147
>>> pow(4,12)
16777216
>>>
>>> list(map(pow,[2, 3, 4], [10, 11, 12]))
[1024, 177147, 16777216]
>>>
```

As in the example above, with multiple sequences, `map()` expects an N-argument function for N sequences. In the example, `pow` function takes two arguments on each call.

## Reduce and Lambda

- **reduce** accepts iterator to process and returns single result:

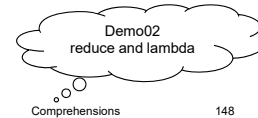
```
from functools import reduce
var1 = reduce((lambda x, y: x * y), [1, 2, 3, 4])
print(var1) # 24
```

- **Above code works as follows:**

- **reduce** passes current product along with next item from list to **lambda** function
- Default first item in sequence initialized starting value

- **Same functionality with for loop:**

```
li1 = [1, 2, 3, 4]
result = li1[0]
for x in li1[1:]:
 result = result * x
print(result)
```



www.spiraltrain.nl

As the name suggests filter extracts each element in the sequence for which the function returns True. The reduce function is a little less obvious in its intent. This function reduces a list to a single value by combining elements via a supplied function. The map function is the simplest one among Python built-ins used for functional programming.

These tools apply functions to sequences and other iterables. The filter filters out items based on a test function which is a filter and apply functions to pairs of item and running result which is reduce.

Because they return iterables, range and filter both require list calls to display all their results in Python 3.0. As an example, the following **filter** call picks out items in a sequence that are less than zero:

```
>>> list(range(-5,5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>>
>>> list(filter((lambda x: x < 0), range(-5,5)))
[-5, -4, -3, -2, -1]
>>>
```

Items in the sequence or iterable for which the function returns a true, the result are added to the result list. Like map, this function is roughly equivalent to a for loop, but it is built-in and fast:

```
>>>
>>> result = []
>>> for x in range(-5, 5):
>>> if x < 0:
>>> result.append(x)
>>>
>>> result
[-5, -4, -3, -2, -1]
>>>
```

## List Comprehensions

- **Construct for creating new list based on existing lists:**
  - Languages like Haskell, Erlang, Scala and Python have them
- **Comprehension term:**
  - Comes from math's set comprehension notation to define sets in terms of other sets
- **Powerful and popular feature in Python:**
  - Generate new list by applying operation or function to every member of original list

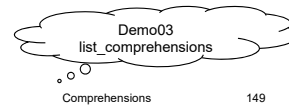
```
slist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(slist)
```

```
list_str = [str(x) for x in slist]
print(list_str)
```

- **Output:** ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10']

```
new_list = [x * 2 for x in slist]
print(new_list)
```

- **Output:** [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]



www.spiraltrain.nl

Comprehensions

149

Python features functional programming tools like map and filter for mapping operations over sequences and collecting results. Since this is such a common task in Python coding, Python made a new expression: the list comprehension which is more flexible than map and filter. List comprehensions apply an arbitrary expression to items in an iterable rather than applying function. It provides a compact way of mapping a list into another list by applying a function to each of the elements of the list.

### List Comprehension vs. map

Python's built-in ord returns the ASCII integer code of a character:

```
>>> ord('A')
65
```

If we want to collect the ASCII codes of all characters in a string, the most straightforward method is using a for loop and append the results to a list:

```
>>> result = []
>>> for x in 'Dostoyevsky':
>>> result.append(ord(x))
>>> result
[68, 111, 115, 116, 111, 121, 101, 118, 115, 107, 121]
>>>
```

If we use map, we can get the same result with a single function call:

```
>>> result = list(map(ord, 'Dostoyevsky'))
>>> result
[68, 111, 115, 116, 111, 121, 101, 118, 115, 107, 121]
>>>
```

## Syntax List Comprehensions

- **Syntax of a list comprehension may involve:**
  - for loop, an in operation and an if statement
- **Python's notation:**
  - `[ expression for name in list ]`
  - Expression:
    - Some calculation or operation acting upon the variable name
  - For each member of the list, the list comprehension
    - Sets name equal to that member
    - Calculates a new value using expression
  - Collects these new values into a list which is return value of list comprehension
 

```
li = [3, 6, 2, 7]
li_new = [elem*2 for elem in li] => [6, 12, 4, 14]
```
- **if statement acts as a filter and is optional:**

```
[x-10 for x in grades if x>0]
```

But, we can get the similar result from a list comprehension expression. While map maps a function over a sequence, list comprehensions map an expression over a sequence:

```
>>> result = [ord(x) for x in 'Dostoyevsky']
>>> result
[68, 111, 115, 116, 111, 121, 101, 118, 115, 107, 121]
>>>
```

List comprehensions collect the result of applying an arbitrary expression to a sequence and return them in a new list. The effect is similar to that of the for loop and the map call. List comprehensions become more convenient when we need to apply an arbitrary expression to a sequence:

```
>>> [x ** 3 for x in range(5)]
[0, 1, 8, 27, 64]
```

If we had to use map, we would need to write a function to implement the square operation, probably, lambda instead of using a def:

```
>>> list(map((lambda x: x ** 2), range(5)))
[0, 1, 4, 9, 16]
```

This does the job. It's only a little bit longer than the list comprehension. For more advanced kinds of expressions, however, list comprehensions will often require considerably less typing.

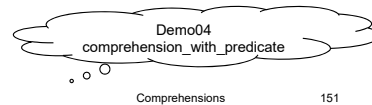
## Filtered List Comprehension

- **Filter determines if expression is performed on list member:**
  - For each element of list, checks if it satisfies the filter condition
- **If filter condition returns False:**
  - Element is omitted from the list before the list comprehension is evaluated
- **Traditional syntax:**

```
slist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for x in slist:
 if (x % 2) == 0:
 new_list.append(str(x))
print(new_list) => ['2', '4', '6', '8', '10']
```

- **Pythonic syntax using list comprehension:**

```
new_list = [str(x) for x in slist if (x % 2) == 0]
print(new_list) => ['2', '4', '6', '8', '10']
```



www.spiraltrain.nl

Comprehensions

151

## List Comprehension with filter

If we use if with list compression, it is almost equivalent to the filter built-in.

Let's make examples using both schemes.

```
>>>
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]
>>> list(filter((lambda x: x % 2 == 0), range(10)))
[0, 2, 4, 6, 8]
>>> result = []
>>> for x in range(10):
 if x % 2 == 0:
 result.append(x)
>>> result
[0, 2, 4, 6, 8]
>>>
```

All of these use the modulus operator %, to extract even numbers. The filter call here is not much longer than the list comprehension either. But we can combine an if and a map, in a single expression:

```
>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
>>>
```

We collect the squares of the even numbers from 0 to 9. The for loop skips numbers which the if on the right is false. The expression on the left computes the squares. The equivalent map requires a lot more work. We have to combine filter with map iteration:

```
>>> list(map((lambda x: x ** 2), filter((lambda x: x % 2 == 0), range(10))))
[0, 4, 16, 36, 64]
```

## Syntactic Sugar

- **List comprehensions are syntactic sugar for higher-order functions:**

```
[expression for name in list]
map(lambda name: expression, list)
```

- **Example using higher order function:**

```
lm = list(map(lambda x: 2*x+1, [10, 20, 30]))
print(lm) # [21, 41, 61]
```

- **Example using list comprehension:**

```
lc = [2*x+1 for x in [10, 20, 30]]
print(lc) # [21, 41, 61]
```

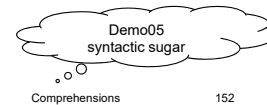
- **Example using higher order function:**

```
ls1 = list(map(lambda x: 2*x+1, filter(lambda x: x > 0, [10, 20, 30, -10])))
print(ls1)
```

- **Example using list comprehension:**

```
ls2 = [2*x+1 for x in [10, 20, 30, -10] if x > 0]
print(ls2)
```

www.spiraltrain.nl



Including if clause shows benefits of sweetened form:

- [ expression for name in list if filt ]
- map( lambda name . expression, filter(filt, list) )

Actually, list comprehensions are more general. We can code any number of nested for loop in a list comprehension, and each may have an optional associated if test. When for loop are nested within a list comprehension, they work like equivalent for loop statement:

```
>>> result = []
>>> result = [x ** y for x in [10, 20, 30] for y in [2, 3, 4]]
>>> result
[100, 1000, 10000, 400, 8000, 160000, 900, 27000, 810000]
>>>
```

More verbose version is:

```
>>> result = []
>>> for x in [10, 20, 30]:
 for y in [2, 3, 4]:
 result.append(x ** y)
>>> result
[100, 1000, 10000, 400, 8000, 160000, 900, 27000, 810000]
>>>
```

Though list comprehensions construct lists, they can iterate over any sequence:

```
>>> [x + y for x in 'ball' for y in 'boy']
['bb', 'bo', 'by', 'ab', 'ao', 'ay', 'lb', 'lo', 'ly', 'lb', 'lo', 'ly']
>>>
```



## Nested List Comprehensions

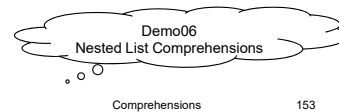
- **List comprehensions can easily be nested:**
  - Take list as input and produce list as output
- **In following inner comprehension produces:** `[4, 3, 5, 2]`

```
li = [3, 2, 4, 1]
[elem*2 for elem in [item+1 for item in li]] #[8, 6, 10, 4]
```
- **Another example is:**

```
list_a = ['A', 'B']
list_b = ['C', 'D']
list_c = [x+y for x in list_a] for y in list_b]
print(list_c)
```
- **Produces:**

```
[['AC', 'BC'], ['AD', 'BD']]
```

www.spiraltrain.nl



Here is a much more complicated list comprehension example:

```
>>> [(x,y) for x in range(5) if x % 2 == 0 for y in range(5) if y % 2 == 1]
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

The expression permutes even numbers from 0 through 4 with odd numbers from 0 through 4. Here is an equivalent version which is much more verbose:

```
>>> result = []
>>> for x in range(5):
 if x % 2 == 0:
 for y in range(5):
 if y % 2 == 1:

 result.append((x,y))
>>> result
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

### List Comprehension with Matrixes

Let's see how the list comprehension works with Matrixes. For example, we have two 3 x 3 matrixes as lists of nested list:

```
>>> M1 = [[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
>>> M2 = [[9, 8, 7],
 [6, 5, 4],
 [3, 2, 1]]
```

We can index rows and columns within rows:

```
>>> M1[2]
[7, 8, 9]
>>> M1[2][2]
```

9

## Comprehension Features

- **If list contains elements of different types:**
  - Expression must operate correctly on the types of all of list members.
- **If elements of list are other containers:**
  - Name can consist of container of names matching type and “shape” of list members

```
list_d = [('a' , 1), ('b', 2), ('c', 7)]
list_e = [n * 3 for (x, n) in list_d]
print(list_e) # [3, 6, 21]
```

- **Containers are objects that contain references to other objects:**
  - Lists, types, dictionaries
- **Expression can also contain user-defined functions:**

```
def subtract(a, b):
 return a - b
oplist = [(6, 3), (1, 7), (5, 5)]
list_f = [subtract(y, x) for (x, y) in oplist]
print(list_f) # [-3, 6, 0]
```



www.spiraltrain.nl

Comprehensions

154

List comprehensions are powerful tools for processing such structures because they automatically scan rows and columns for us. For example, though this structure stores the matrix by rows, to collect the second column we can simply iterate across the rows and pull out the desired column. We can also iterate through positions in the rows and index as we go:

```
>>> [r[2] for r in M1]
[3, 6, 9]
```

Here, we pulled out column 3 from M1. We can get the same result from the following list comprehension.

```
>>> [M1[r][2] for r in (0, 1, 2)]
[3, 6, 9]
```

### When to Use List Comprehension

We typically should use simple for loops when getting started with Python, and map. Use comprehension where they are easy to apply. However, there is a substantial performance advantage to use list comprehension. The map calls are roughly twice as fast as equivalent for loops. List comprehensions are usually slightly faster than map calls. This speed difference is largely due to the fact that map and list comprehensions run at C language speed inside the interpreter. It is much faster than stepping through Python for loop code within the Python Virtual Machine (PVM).

However, for loops make logic more explicit, we may want to use them on the grounds of simplicity. On the other hand, map and list comprehensions are worth knowing and using for simpler kinds of iterations if the speed of application is an important factor.

## Zip Function

- `zip(*iterables):`
  - Makes an iterator that aggregates elements from each of the iterables
  - iterator stops when the shortest input iterable is exhausted

- **Used to combine two lists in list of pairs:**

```
ls1 = [1,2,3]
ls2 = [4,5,6]
ls3 = list(zip(ls1,ls2))
print(ls3) # [(1, 4), (2, 5), (3, 6)]
```

- **Original lists can also be restored (as tuples) using same function:**

```
b1, b2 = zip(*ls3)
print(b1) -> (1,2,3)
print(b2) -> (4,5,6)
```



www.spiraltrain.nl

Comprehensions

155

## Dictionary construction with zip

We can use `zip` to generate dictionaries when the keys and values must be computed at runtime. In general, we can make a dictionary by typing in the dictionary literals:

```
>>> D1 = {'a':1, 'b':2, 'c':3}
>>> D1
{'a': 1, 'c': 3, 'b': 2}
>>>
```

Or we can make it by assigning values to keys:

```
>>> D1 = {}
>>> D1['a'] = 1
>>> D1['b'] = 2
>>> D1['c'] = 3
>>> D1
{'a': 1, 'c': 3, 'b': 2}
```

However, there are cases when we get the keys and values in lists at runtime. For instance like this:

```
>>> keys = ['a', 'b', 'c']
>>> values = [1, 2, 3]
```

How can we construct a dictionary from those lists of keys and values? Now it's time to use `zip`. First, we `zip` the lists and loop through them in parallel like this:

```
>>> list(zip(keys,values))
[('a', 1), ('b', 2), ('c', 3)]
```

```
>>> D2 = {}
>>> for (k,v) in zip(keys, values):
... D2[k] = v
...
>>> D2
{'a': 1, 'b': 2, 'c': 3}
```

## Dictionary Construction with zip

- **zip can be used to generate dictionaries:**
  - When keys and values must be computed at runtime
- **Dictionaries are traditionally created with:**

```
dict1 = {'a':1, 'b':2, 'c':3}
print(dict1) # {'a': 1, 'c': 3, 'b': 2}
```
- **Or we can make it by assigning values to keys:**

```
dict2 = {}
dict2['a'] = 1; dict2['b'] = 2; dict2['c'] = 3
print(dict2) # {'a': 1, 'c': 3, 'b': 2}
```
- **Sometimes keys and values are only know at runtime:**
  - Can then use zip to create lists and loop through them in parallel like this:

```
dict3 = {}
for (k,v) in zip(keys, values):
 dict3[k] = v
print(dict3) # {'a': 1, 'b': 2, 'c': 3}
```
- **Alternative is using constructor with:**

```
dict4 = dict(zip(keys, values))
```



www.spiraltrain.nl

Comprehensions

156

Python 3.x introduced dictionary comprehension, and we'll see how it handles the similar case.

### Dictionary Comprehension

As in the previous section, we have two lists:

```
>>> keys = ['a', 'b', 'c']
>>> values = [1, 2, 3]
```

Now we use dictionary comprehension (Python 3.x) to make dictionary from those two lists:

```
>>> D = { k:v for (k,v) in zip(keys, values)}
>>> D
{'a': 1, 'b': 2, 'c': 3}
```

It seems require more code than just doing this:

```
>>> D = dict(zip(keys, values))
>>> D
{'a': 1, 'b': 2, 'c': 3}
```

However, there are more cases when we can utilize the dictionary comprehension. For instance, we can construct dictionary from one list using comprehension:

```
>>> D = {x: x**2 for x in [1,2,3,4,5]}
>>> D
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

```
>>> D = {x.upper(): x*3 for x in 'abcd'}
>>> D
{'A': 'aaa', 'C': 'ccc', 'B': 'bbb', 'D': 'ddd'}
```

When we want initialize a dict from keys, we do this:

```
>>> D = dict.fromkeys(['a','b','c'], 0)
>>> D
{'a': 0, 'c': 0, 'b': 0}
```

## Dictionary Comprehensions

- **Dictionary comprehension is like a list comprehension:**
  - Constructs a dictionary instead of a list
- **Dictionary comprehension makes dictionary from two lists:**

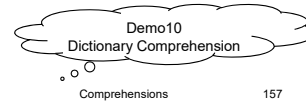
```
keys = ['a', 'b', 'c']
values = [1, 2, 3]
dict5 = { k:v for (k,v) in zip(keys, values)}
print(dict5) # {'a': 1, 'b': 2, 'c': 3}
```

- **Requires actually more code then constructor function:**

```
dict6 = dict(zip(keys, values))
print(dict6) # {'b': 2, 'c': 3, 'a': 1}
```

- **Many more occasions to use dictionary comprehension:**

```
dict7 = {x: x**2 for x in [1,2,3,4,5]}
print(dict7) # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
dict8 = {x.upper(): x*3 for x in 'abcd'}
print(dict8) # {'A': 'aaa', 'C': 'ccc', 'B': 'bbb', 'D': 'ddd'}
```



www.spiraltrain.nl

We can use dictionary comprehension to do the same thing ;

```
>>> D = {k: 0 for k in ['a','b','c']}
>>> D
{'a': 0, 'c': 0, 'b': 0}
```

We sometimes generate a dict by iterating each element:

```
>>> D = dict.fromkeys('dictionary')
>>> D
{'a': None, 'c': None, 'd': None, 'i': None, 'o': None, 'n': None,
'r': None, ...}
```

If we use comprehension:

```
>>> D = {k:None for k in 'dictionary'}
>>> D
{'a': None, 'c': None, 'd': None, 'i': None, 'o': None, 'n': None,
'r': None}
```

### Simple zip()

The following example calculates Hamming distance. Hamming distance is the number of positions at which the corresponding symbols are different. It's defined for two strings of equal length.

```
def hamming(s1,s2):
 if len(s1) != len(s2):
 raise ValueError("Not defined,unequal lenght sequences")
 return sum(c1 != c2 for c1,c2 in zip(s1,s2))
if __name__ == '__main__':
 print(hamming('toned', 'roses')) # 3
 print(hamming('2173896', '2233796')) # 3
 print(hamming('0100101000', '1101010100')) # 6
```

## Dictionary from Keys

- We can initialize dictionary from keys as follows:

```
dict9 = dict.fromkeys(['a','b','c'], 0)
print(dict9) # {'a': 0, 'c': 0, 'b': 0}
```

- Can use dictionary comprehension to do the same thing:

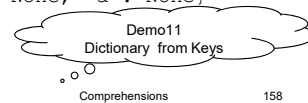
```
dict10 = {k: 0 for k in ['a','b','c']}
print(dict10) # {'a': 0, 'c': 0, 'b': 0}
```

- Generate dictionary by iterating each element:

```
dict11 = dict.fromkeys('dictionary')
print(dict11) # {'d': None, 'o': None, 'i': None,
't': None, 'a': None, 'c': None, 'y': None, 'n': None, 'r': None}
```

- Now using comprehension:

```
dict12 = {k:None for k in 'dictionary'}
print(dict12) # {'t': None, 'y': None, 'c': None,
'd': None, 'i': None, 'r': None, 'o': None, 'n': None, 'a': None}
```



www.spiraltrain.nl

### Dictionary comprehensions:

Dictionary comprehensions operate the same way, but use two variables, the key and its value, instead of the typically single value for list iteration.

```
abs_paths = { k: os.path.join(os.environ['HOME'], v) for (k,v) in
rel_paths.iteritems() }
```

The only differences in syntax are:

1. We use braces instead of brackets. We are specifying the container type here
2. We specify the key:value pair in our expression
3. We identify the key and value variable names as a tuple following the for clause. in reality, there is no syntactic difference with this part of the comprehension syntax. Lists can have tuples to define the values used in the expression. Example:

```
a = [(1,2), (3,4), (5,6)]
b = [x*y for (x,y) in a]
```

## Set Comprehensions

- **Sets have their own comprehension syntax as well:**
  - Quite similar to the syntax for dictionary comprehensions
  - Difference is that sets just have values instead of `key:value` pairs
- **Set comprehensions can take set as input:**
  - Following set comprehension calculates squares of set of numbers from 0 to 9

```
my_set = set(range(10))
print(my_set) # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
my_set2 = {x ** 2 for x in my_set}
print(my_set2) # {0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
```

- **Set comprehensions can contain `if` clause to filter items:**

```
my_set3 = {x for x in my_set if x % 2 == 0}
print(my_set3) # {0, 8, 2, 4, 6}
```

- **Set comprehensions do not have to take set as input:**

```
• They can take any sequence
my_set4 = {2**x for x in range(10)}
print(my_set4) # {32, 1, 2, 4, 8, 64, 128, 256, 16, 512}
```



www.spiraltrain.nl

Comprehensions

159

As many of you are familiar with list comprehension in Python, let me inform you about set comprehension. You can create a set from a list or a dictionary. Example:

```
#create set from list
>>> s = set([1, 2, 3])
>>> s
set([1, 2, 3])
#create set from dictionary (keys)
>>> dt = {1: 10, 2: 20, 3: 30}
>>> dt
{1: 10, 2: 20, 3: 30}
>>> type(dt)
<type 'dict'>
>>> s = set(dt)
>>> s
set([1, 2, 3])
```

Now we can create a set without using the `set()` function: (The feature is available in Python 3.x and Python 2.7)

```
>>> s = {1, 2, 3, 'Bangladesh', 'python', 1.15}
>>> type(s)
<type 'set'>
>>>
```

And here is an example of set comprehension (similar to list comprehension):

```
>>> s = { x for x in range(10) }
>>> s
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

## Course Schedule

- Python Intro
- Variables and Data Types
- Data Structures
- Control Flow
- Functions
- Modules
- Classes and Objects
- Exception Handling
- Python IO
- Comprehensions
- Generators
- **Decorators**
  - Function as Objects
  - Passing and Returning Functions
  - What is a Decorator?
  - Decorator Syntax
  - Types of Decorators
  - Passing Arguments
  - Multiple Decorators
  - Class Decorators
  - Syntax Class Decorators
  - Singleton Class
  - Why Decorators
  - Need for AOP
  - Crosscutting Concerns
- Python Database Access
- Python and XML
- Python Libraries

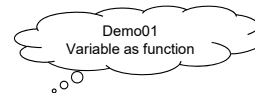


## Functions as Objects

- **Functions are first class citizens in Python**
- **Functions can be assigned to variables:**

```
def greet(name):
 return "hello " + name

greet_someone = greet
print(greet_someone("Albert"))
=> Outputs: hello Albert
```

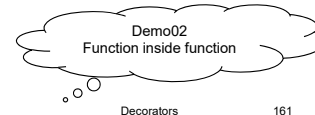


- **Function can be defined inside another function:**

```
def greet(name):
 def get_message():
 return "Hello "

 result = get_message()+name
 return result

print(greet("Albert"))
=> Outputs: Hello Albert
```



www.spiraltrain.nl

Decorators

161

But when it comes to technical topics, everyone has his or her own style of learning and one size of explanation does not fit all.

So I thought I'd try my hand at writing an introduction to Python decorators. My goal is not to explain everything about decorators. Instead I want to try to explain just the basics, just enough to give you a workable mental model of what decorators are and how to use them. Just enough to get started on doing useful work with decorators.

As Aristotle said, "Let us begin at the beginning", which is to say, we begin by looking at functions.

### Functions

When the Python interpreter encounters this code:

```
def hello():
 print("Hello, world!")
```

- It compiles the code to create a function object
- it binds the name "hello" to that function object.

Then, to run the function object, you can code:

```
hello()
```

which causes this to be printed:

```
Hello, world!
```

If you code:

```
print(hello)
```

you will get something like:

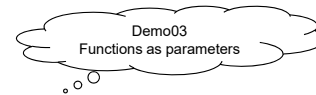
```
<function hello at 0x02D021E0>
```

which is the string representation of the hello function object.

## Passing and Returning Functions

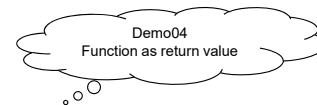
- Python supports functional programming style
- Function can take other function as an argument:

```
def greet(name):
 return "Hello " + name
def call_func(func):
 other_name = "Albert"
 return func(other_name)
print(call_func(greet)) => Outputs: Hello Albert
```



- Function can return another function:

```
def compose_greet_func():
 def get_message():
 return "Hello there!"
 return get_message
greet = compose_greet_func()
print(greet()) => Outputs: Hello there!
```



www.spiraltrain.nl

Decorators

162

To understand decorators, you must first understand that functions are objects in Python. This has important consequences. Let's see why with a simple example:

```
def shout(word="yes"):
 return word.capitalize()+"!"
print shout()
outputs: 'Yes!'
```

As an object, you can assign the function to a variable like any other object

```
scream = shout
```

Notice we don't use parentheses: we are not calling the function, we are putting the function "shout" into the variable "scream". It means you can then call "shout" from "scream":

```
print scream()
outputs: 'Yes!'
```

More than that, it means you can remove the old name 'shout', and the function will still be accessible from 'scream'

```
del shout
try:
 print shout()
except NameError, e:
 print e
#outputs: "name 'shout' is not defined"
print scream()
outputs: 'Yes!'
```

Okay! Keep this in mind. We'll circle back to it shortly.

## Closure

- **Closure is combination of code and scope:**
  - Python functions combine code to be executed and scope in which to do that
- **Closure is mostly about nested function and scope of function:**
  - Function can retain value when it was created even though scope cease to exist

```
def startAt(start):
 def incrementBy(inc):
 return start + inc
 return incrementBy

f = startAt(10)
g = startAt(100)
print f(1), g(2) # print 11 102
```

- **Inner function can access outer scope:**

```
def compose_greet_func(name):
 def get_message():
 return "Hello there "+name+"!"
 return get_message

greet = compose_greet_func("Albert")
print(greet()) # Outputs: Hello there Albert
```



www.spiraltrain.nl

163

Short answer may be like this: a closure is a combination of code and scope. However, most of the time when we speak about closure, it's about nested function and the scope of the function. Sometimes we want a function to retain a value when it was created even though the scope cease to exist. This technique of using the values of outer parameters within a dynamic function is another way of defining the closure.

```
def startAt(start):
 def incrementBy(inc):
 return start + inc
 return incrementBy

f = startAt(10)
g = startAt(100)
print f(1), g(2)
```

Closures in python are created by function calls. In the code above, the call to `startAt()` creates a binding for `start` that is referenced inside the function `incrementBy()`. Each call to `startAt()` creates a new instance of this function, but each instance has a link to a different binding of `start`. If we run it:

```
11 102
```

So, it looks like the call objects `f` and `g` retain their states at the time they were created. When we create `f`, the outer function `startAt()` uses the nested function `incrementBy()` as a return value. Note that it's the function itself that is returned, not the return value of that function. The inner function is not called within the `startAt()` function. So, the `startAt()` is a function that returns a function when called. In that way, our program can have an external reference to the nested function, and the nested function retains its reference to the call object of the outer function. In that way, the call object for a particular invocation of the outer function continues to live.

In summary, a closure is a function (object) that remembers its creation environment (enclosing scope).

## What is a Decorator?

- **Decorator is used to wrap a function:**
  - Gives new functionality without changing the original function
- **Following decorator wraps function output with bold tags:**

```
def get_text(name):
 return "Hello {0}, how are you".format(name)

def p_decorate(func):
 def func_wrapper(name):
 return "<p>{0}</p>".format(func(name))
 return func_wrapper

my_get_text = p_decorate(get_text)

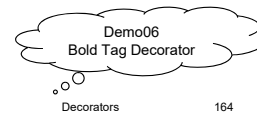
print(my_get_text("Albert"))

=> Outputs <p>Hello Albert, how are you</p>

get_text = p_decorate(get_text)

print(get_text("Albert"))

=> Outputs <p>Hello Albert, how are you</p>
```



www.spiraltrain.nl

Another interesting property of Python functions is they can be defined... inside another function!

```
def talk():
 # You can define a function on the fly in "talk" ...
 def whisper(word="yes"):
 return word.lower()+"..."
 # ... and use it right away!
 print whisper()
```

```
You call "talk", that defines "whisper" EVERY TIME you
call it, then "whisper" is called in "talk".
```

```
talk()
outputs:
"yes..."
But "whisper" DOES NOT EXIST outside "talk":
```

```
try:
 print whisper()
except NameError, e:
 print e
#outputs: "name 'whisper' is not defined"*
#Python's functions are objects
```

## Decorator Syntax

### Syntax:

```
def decorator_function():

@decorator_function
def my_func:


```

### This is equivalent to:

```
my_func =
decorator_function(my_func)
```

- **Decorator function:**

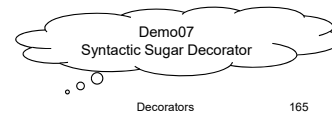
- Takes function being decorated as parameter
- Wraps it in a wrapper function and returns wrapper function

```
def decorator_function(decorated_function):
 def wrapper_function():

 some code -----
 decorated_function()

 return wrapper_function
```

www.spiraltrain.nl



Decorators

165

## Annotations

Many discussions of decorators use the word “decorator” rather loosely, to refer to different decorator-related concepts. This kind of ambiguity is disconcerting at best, and confusing at worst.

To help avoid this ambiguity I will use the term “annotation” in this discussion to refer to lines of code that begin with “@”.

Here is a snippet of Python code that begins with two annotations:

```
@helloGalaxy
@helloSolarSystem
def hello():
 print("Hello, world!")
```

We can say that the definition of the hello function is “decorated” with these two annotations. Since there are multiple annotations, we say that the annotations are “stacked”. When the interpreter sees these lines of code, here is what it does.

- It pushes helloGalaxy onto the annotation stack.
- It pushes helloSolarSystem onto the annotation stack.

then it does the standard processing for a function definition ...

It compiles the code for hello into a function object (lets call it functionObject1)

It binds the name “hello” to functionObject1.

then...

It pops helloSolarSystem off of the annotation stack,

passes functionObject1 to helloSolarSystem ...

helloSolarSystem returns a new function object (lets call it functionObject2), and...

the interpreter binds the original name “hello” to functionObject2

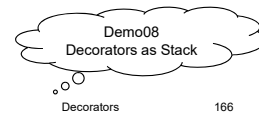
then...

## Types of Decorators

- **There are two types of decorators:**
  - Built-in decorators and User Defined decorators
- **Built in decorators are** `staticmethod` **and** `classmethod`:
  - Used in classes to define class or hierarchy specific methods
  - Also adressable with `@staticmethod` or `@classmethod` annotations
- **User defined decorators:**
  - Can be more that one decorator for one function and executed using stack
- **First all decorators are pushed into a stack:**
  - Then they are popped one by one

```
@helloGalaxy
@helloSolarSystem
def hello():
 print("Hello World")
hello()
```

www.spiraltrain.nl



166

then...

It pops `helloGalaxy` off of the annotation stack,

passes `functionObject2` to `helloGalaxy` ...

`helloGalaxy` returns a new function object (lets call it `functionObject3`), and...

the interpreter binds the original name "hello" to `functionObject3`

As you can see, this process could be repeated for indefinitely many annotations.

I've been vague about what kind of thing that `helloSolarSystem` and `helloGalaxy` are. For now, think of them as a special kind of function — a kind of function that takes one function object as an argument, and returns another function object as a result. The annotations:

```
@helloGalaxy
```

```
@helloSolarSystem
```

are calls to these functions. So this snippet of Python code:

```
@helloGalaxy
```

```
@helloSolarSystem
```

```
def hello():
 print("Hello, world!")
```

is functionally equivalent to this:

```
def hello():
 print("Hello, world!")
hello = helloSolarSystem(hello)
hello = helloGalaxy(hello)
```

## Passing Arguments

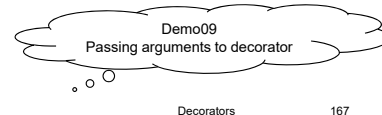
- **Wrapper function should accept arguments:**
  - If arguments which being passed to the function being decorated

```
def helloSolarSystem(old_function):
 def wrapper_function(planet=None):
 print("Hello Solar System")
 old_function(planet)
 print("Leaving Solar System")
 return wrapper_function

@helloSolarSystem
def hello(planet=None):
 if planet:
 print("Hello "+planet)
 else:
 print("Hello World")

hello("Mars")
hello()
```

- **Decoration can take place:**
  - Before, after or around the original function call



www.spiraltrain.nl

Decorators

167

## Arguments to functions

Now lets look at decorating functions that take arguments. Let's modify the *hello* function so it accepts an argument, like this:

```
def hello(targetName=None):
 if targetName:
 print("Hello, " + targetName + "!")
 else:
 print("Hello, world!")
```

If we were to run an undecorated version of the hello function, we'd get a nice greeting, like this:

```
>>> hello("Earth")
Hello, Earth!
```

But if we run the decorated version of the hello function, we get this:

```
TypeError: new_function() takes no arguments (1 given)
```

What's the problem?

Remember that we wrapped functionObject1 (created from hello) in functionObject2 (created from helloSolarSystem) and then in functionObject3 (created from Galaxy), and then bound the name "hello" to functionObject3. So when we use the "hello" function, we are calling functionObject3.

FunctionObject3 was created by the code for new\_function in helloGalaxy, and it accepts no arguments. Which is why we get the error message:

```
TypeError: new_function() takes no arguments (1 given)
```

## Variable Arguments

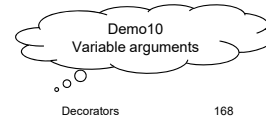
- **Decorator wrapper function must support variable arguments:**
  - To be able to work with different functions
  - Also need to add code to `new_function` so that it will accept arguments
- **Define general wrapper function using `*args` and `**kwargs`:**

```
def helloSolarSystem(original_function):
 def new_function(*args, **kwargs):
 original_function(*args, **kwargs)
 print("Hello, galaxy!")
 return new_function
```

- **Possible annotations:**

```
@helloSolarSystem
def hello(planet=None):
 @helloSolarSystem
 def goodbye(planets, moons, species):
```

- **Can also define wrapper for class method:**
  - But their first argument should be `self`



[www.spiraltrain.nl](http://www.spiraltrain.nl)

The solution is to add support for arguments to the function objects that our decorators create. We need to add code to `new_function` so that it will accept arguments, and we need to add code to `original_function` so that it will accept the arguments that its wrapper (`new_function`) makes available to it.

```
def helloSolarSystem(original_function):
 def new_function(*args, **kwargs):
 original_function(*args, **kwargs)
 print("Hello, solar system!")
 return new_function
```

```
def helloGalaxy(original_function):
 def new_function(*args, **kwargs):
 original_function(*args, **kwargs)
 print("Hello, galaxy!")
 return new_function
```

And now:

```
>>>hello("Earth")
Hello, Earth!
Hello, solar system!
Hello, galaxy!
```



## Multiple Decorators

```
def p_decorate(func):
 def func_wrapper(name):
 return "<p>{0}</p>".format(func(name))
 return func_wrapper

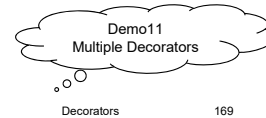
def strong_decorate(func):
 def func_wrapper(name):
 return "{0}".format(func(name))
 return func_wrapper

def div_decorate(func):
 def func_wrapper(name):
 return "<div>{0}</div>".format(func(name))
 return func_wrapper

@div_decorate
@p_decorate
@strong_decorate
def get_text(name):
 return "Hello {0} how are you".format(name)

print(get_text("Mamaloe"))
```

www.spiraltrain.nl



169

```
def p_decorate(func):
 def func_wrapper(name):
 return "<p>{0}</p>".format(func(name))
 return func_wrapper

def strong_decorate(func):
 def func_wrapper(name):
 return "{0}".format(func(name))
 return func_wrapper

def div_decorate(func):
 def func_wrapper(name):
 return "<div>{0}</div>".format(func(name))
 return func_wrapper

def get_text(name):
 return "Hello {0} how are you?".format(name)

get_text = div_decorate(p_decorate(strong_decorate(get_text)))

print(get_text("Pipo"))
```

## Class Decorators

- **Classes can also be decorated in Python:**
  - Just like functions can be decorated with other functions
- **Class decorators add required functionality:**
  - External to class implementation
- **Class decorators:**
  - Run at end of class statement to rebind a class name to a callable
  - Used to manage classes when they are created
  - Can be used to insert a layer of wrapper logic to manage instances

### Decorator definition:

```
def class_decorator(cls):

```

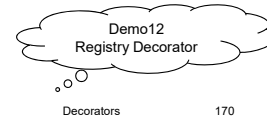
### Class definition:

```
@class_decorator
class My_Class:

```

### This is equivalent to :

```
My_Class = class_decorator(My_Class)
```



www.spiraltrain.nl

170

Just like functions can be decorated with other functions, classes can also be decorated in python. We decorate classes to add required functionality that maybe external to the class implementation; for example we may want to enforce the singleton pattern for a given class. Some functions implemented by class decorators can also implemented by metaclasses but class decorators sometimes make for a cleaner implementation to such functionality.

The most popular example used to illustrate the class decorators is that of a registry for class ids as they are created.

```
registry = {}

def register(cls):
 registry[cls.__clsid__] = cls
 return cls

@register
class Foo(object):
 __clsid__ = "123-456"

def bar(self):
 pass
```

## Singleton Class

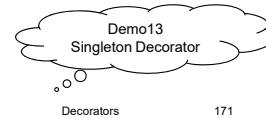
```
instances = {} #dictionary to hold class and their only instance

def getInstance(klass,*args): #this function is used by decorator
 if klass not in instances:
 instances[klass]=klass(*args)
 return instances[klass]

def singleton(klass): #decorator function
 def onCall(*args):
 return getInstance(klass,*args)
 return onCall

@singleton
class Star:
 def __init__(self,name):
 self.name=name

#A solar system should have only one star
sun = Star('Sun')
print sun.name
taurus = Star('Taurus')
print taurus.name
```



www.spiraltrain.nl

Decorators

171

Another example of using class decorators is for implementing the singleton pattern as shown below:

```
def singleton(cls):
 instances = {}
 def get_instance():
 if cls not in instances:
 instances[cls] = cls()
 return instances[cls]
 return get_instance
```

The decorator defined above can be used to decorate any python class forcing that class to initialize a single instance of itself throughout the life time of the execution of the program.

```
@singleton
class Foo(object):
 pass

>>> x = Foo()
>>> id(x)
4310648144
>>> y = Foo()
>>> id(y)
4310648144
>>> id(y) == id(x)
True
>>>
```