



Close

Python 3 Programming

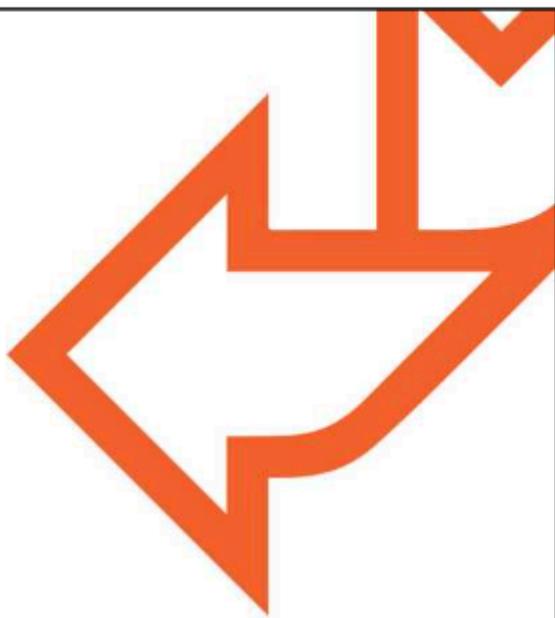
Learner Guide





Python 3 Programming

Introduction to Python 3



QA

INTRODUCTION TO PYTHON 3



Contents

- What is Python?
- Why Python?
- Zen of Python
- Python Interactive Session
- IPython and Jupyter
- Python scripts
- Python help
- Example of a Python script
- Modules
- Functions and built-ins

Summary

- Python built-in functions



This chapter introduces version 3 of the Python language and explains why Python remains popular after so many years, its benefits, and the 19 guiding principles that influence the design of the Python programming language.

We will discover that Python is a general purpose, object-oriented scripting language and has several interfaces including online web-based interfaces such as Jupyter. As an extensible language, it supports libraries of modules that extend the core language giving a helping hand in developing quick and efficient solutions.

We will only have time to look at basics in this chapter, but that should be enough to get started.

QA What is Python?

Python is an object-oriented scripting language

- First published in 1991 by Guido van Rossum
- Designed as an OOP language from day one
- Does not need knowledge of OO to use

It is powerful

- General purpose, fully functional, rich
- Many extension modules

It is free

- Open source: Python licence is less restrictive than GPL

It is portable

- UNIX, Linux, Windows, OS X, Android, etc...
 - Ported to the Java and .NET virtual machines



Python is a scripting language with a clean and Object Oriented (OO) based interface.

Python was invented by Guido van Rossum. However, a much wider community has been involved in its development. Although a free and open source, Python version 2 is copyright – BeOpen.com and Stichting Mathematisch Centrum Amsterdam. The Python 3 copyright is owned by the Python Software Foundation, which was formed in 2001. It is not controlled by the Free Software Foundation and does not contain any GNU code (although there are optional links), although the licence is "GPL compatible".

Selected links:

Python: <http://www.python.org>

Jython: <http://www.jython.org>

Iron Python:

<http://www.codeplex.com/Wiki/View.aspx?ProjectName=Iron+Python>

QA

WHAT IS PYTHON 3?



Python 3 was released in December 2008

- Also known as Python 3000 or Py3k
- New version of the language

Not backward compatible with Python 2

- `2to3.py` tool distributed from Python 2.6 and 3.n

Most language features are the same

- Some detail has changed
- Syntax cleaned up
- Many deprecated features have been removed

The first thing everyone notices about Python 3 is that the `print` statement is no longer a statement: it is a built-in function (see later). The effect is that we now must put parentheses around the thing we wish to print. That took a while to get used to after years of missing them out.

Many Python 3 scripts will run on earlier versions of Python 2, particularly on version 2.6 or later. This should be a coincidence rather than a feature and should not be relied on. The same may be said of Python 2 scripts which coincidentally runs on Python 3.

From Python 2.6, many Python 3 features have been added to Python 2. Python 2.6 has a `-3` option to warn about differences between that version and Python 3, and there is a conversion utility `2to3.py`. All versions of Python 2 have now reached end-of-life with final release of 2.7 on July 4th, 2010, and maintenance (bug fixes) ended in 2019.

Guido van Rossum says 'If you're starting a brand-new thing, you should use 3.0'. Mostly because of performance problems, 3.0 only

had a short life and was replaced by 3.1 within a few months.

The King is dead. Long live the King!

QA Why Python?

The most common version is written in C

- Known as CPython
- Java based version: Jython
- C# (.Net) based version: Iron Python

Python based version: PyPy

- Just-in-time compiler
- Remarkable performance improvements x4
- Fewer threading issues
- Latest version is 3.9.10
- Might be the default Python one day
- Compile it yourself!

Platform Agnostic

- No platform specific functions in the base language
- Greater portability for applications



There are a number of versions of Python, and they all differ in subtle ways. The most common version, and the one assumed for this course, is known as CPython. It is written in C, for speed and portability, and runs on many platforms. You can even download the source code and compile it yourself. The latest version being 3.12.1 as of January 2024.

The Java based version, Jython, uses Java native services for many of its features. It is particularly suitable for interfacing with Java systems, since it can use Java classes. Iron Python does a similar task with .Net objects. Jython is still awaiting a version 3 release, but the latest release of Iron Python is 3.4.1 (December 2023).

All these implementations, including CPython, have developers who talk to each other and exchange ideas, and sometimes even code. A pure Python VM, called PyPy, is in development, yet the other VMs have all gained new insights from it.

One of the features of the Python language is that operating specific features are separated out into a module called os. Nothing in that module is guaranteed to be portable - that is the

point of it. Despite that we can never be 100% sure that our scripts will run across platforms without change, filename syntaxes, for example, vary between UNIX, Windows, and Apple filesystems.

Q^ Why Python?

Power

- Garbage collector
- Native Unicode objects

Supports many component libraries

- GUI libraries such as wxPython and PyQt
- Scientific libraries such as NumPy, DISLIN, SciPy, etc.
- Mature JSON support
- Web template systems such as Jinja2, Mako, etc.
- Web frameworks such as Django, Pyjamas, and Zope
- Relational Databases support
- Libraries may be coded in C or C++



Like most modern languages, Python has a garbage collector. This means we do not have to worry too much about tidying up memory (we said 'too much', not 'at all'). Unicode is the term used to describe multi-byte character sets, i.e., those containing more than 256 characters. Western Europeans may consider these 'foreign', but Unicode is required to store and display the Euro currency symbol € and is web-ready including all the characters in all the human languages, and emojis!

One of the strengths of community-based products like Python is the huge range of add-on libraries, most of them free. They include numeric libraries like NumPy (<http://numpy.scipy.org>), and DISLIN - a library for data visualization (<http://www.mps.mpg.de/dislin>) .

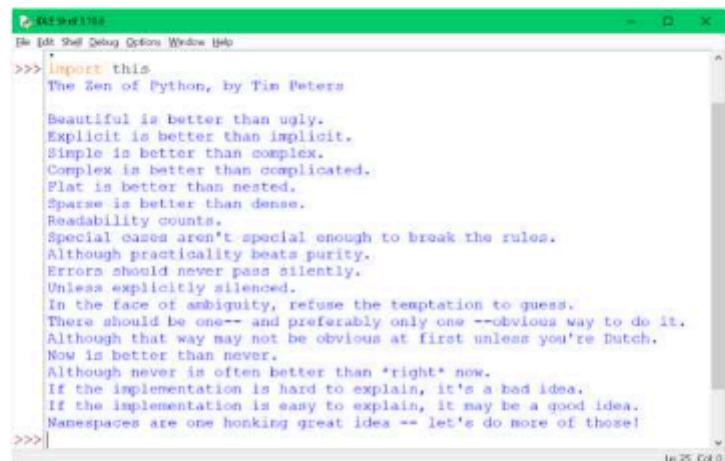
JSON (JavaScript Object Notation) is considered an important part of Python and is heavily used. It is used with Ajax and is an alternative to XML. There are several web development frameworks, not just those listed, and many other web interfaces, including at least four to the Twitter API (most use JSON).

RDBMS support includes PostgreSQL, Oracle, DB2, Sybase, SAPDB, Informix, Ingres, MySQL, SQLite, ODBC, and native Python databases buzhug, and SnakeSQL. Although it must be said there have been some issues with Microsoft's SQL Server in earlier versions.

The add-ons themselves would not be possible without low-level hooks into the C programming language. Most APIs (Application Program Interfaces) are designed around C, and Python can have libraries written in C. To interface to a 3rd-party product therefore, one 'merely' writes a C-Python wrapper around it.

QA Zen of Python

19 guiding principles



```
File Edit Shell Debug Options Window Help
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
10:25 Ed 0
```



...and some entertainment

- pygame – set of libraries for writing games
- Many games use Python as a scripting language
- A fun way to learn

It would be remiss not to mention the module *this* – a collection of 19 guiding principles by Tim Peters for writing computer programs to influence the design of the Python language.

To view these Zen-like principles, try this Easter egg at the IDLE prompt:

```
>>> import this
```

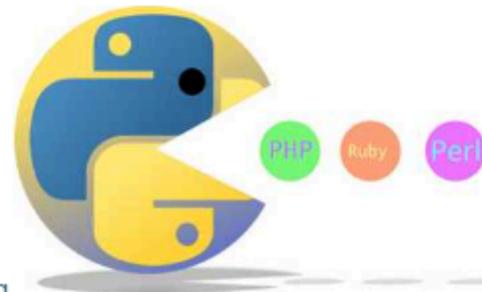
And if you are game for some more entertainment, download pygame and try out Flappy Bird, Super Potato Bruh, Sudoku and of course snake! And then develop your own games!

See also <http://www.pygame.org> "Takes the C++ out of Game Development"

Q A Performance downsides

Performance

- Python programs are compiled at runtime into 'byte code'
- Byte code does not compile down to PE or ELF
- Cannot be as fast as well written C
- Worse CPU usage than Java and C# on most benchmarks
- Better memory management



But...

- Roughly equivalent to Perl and Ruby - depending on the benchmark
- Better than PHP, much better than Tcl
- Packages often use compiled extensions
- Google *Unladen Swallow* project improved the base

Like many scripting languages – such as Java, C#, Perl, Ruby, and PHP - Python is compiled at runtime into an internal format known as byte-code. It is unrealistic to expect this to run as quickly as C code which has been compiled into a native executable format such as PE COFF (on Windows) or ELF (on UNIX). Despite this, some remarkable benchmarks have been produced where the performance is sometimes comparable.

It is worth pointing out that it requires a very good C programmer to produce fast, efficient code which is safe. Achieving safe code in Python is much easier, and quicker. As we shall see later, the byte-code produced by a Python module is usually saved, so it only is compiled once. This byte-code is not portable.

Further information on benchmarks is available at:
<http://shootout.alioth.debian.org/gp4/python.php>.
Iron Python and Jython perform about the same as CPython, however on all these VM implementations there is continual work on improvements and optimisations.

There are several projects working on speeding up Python. Psyco was an old Python 2 package that has maintenance issues and will not run on 64-bit machines. The development effort on Psyco has moved to PyPy (Python written in Python), which has recently been ported to Python 3.

Google's 'Unladen Swallow' project was merged into the base and has improved the base performance ('Unladen Swallow' comes from a memorable line in 'Monty Python and the Holy Grail'), see PEP 3146.

QA The community

Python Software Foundation (PSF)

- Holds the Python intellectual property rights
- The starting point: www.python.org



Many newsgroups and forums

- Main download and docs: <https://python.org>
- Main newsgroup for Python users: <comp.lang.python>
- Web forum: <python-forum.org/pythonforum/index.php>
- Blog: <http://planet.python.org/>

Python Conference: PyCon

- EuroPython moves around European cities

Python Interest Groups

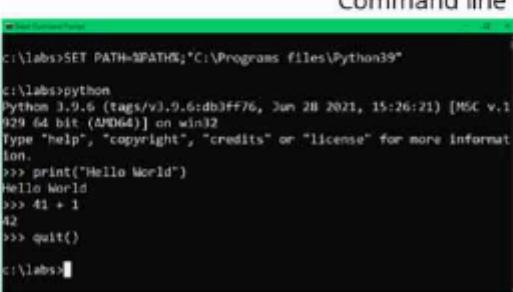
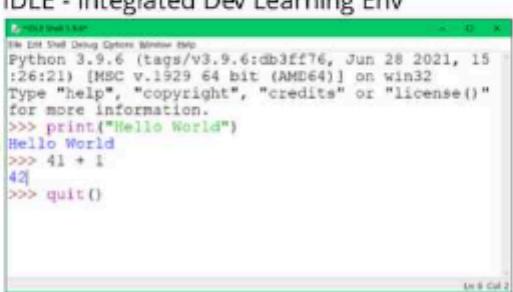
- U.S.A. - PIGgies

The starting point for anything Python related should be the Python Software Foundation's (PSF) website. There you will find many links, including downloads, tutorials, and modules.

There are several Python Conferences around the world each year, including a EuroPython. See the PyCon link from python.org.

'Python', and the Python logo, are registered trademarks of the PSF.

Q4 Python interactive session

<p>Command line</p>  <p>Arrow Keys</p> <p>VS</p> <p>Benefits</p> <ul style="list-style-type: none">• Interactive• Available on Windows or Linux Shell• Limited debugging features• Interactive help() and documentation• quit() to end session	<p>IDLE - Integrated Dev Learning Env</p>  <p>Alt/Ctrl p</p> <p>Alt/Ctrl n</p> <p>Benefits</p> <ul style="list-style-type: none">• Interactive• Portable• Limited debugging features• Interactive help() and full documentation• GUI and syntax highlighting• Built-in editor• quit() to end session
---	---

When python is called interactively from a shell, whether it is a UNIX style shell such as the Korn shell or Bash, or Microsoft's cmd.exe.

The Python shell prompt `>>>` is sometimes called the REPL prompt as it Reads, Evaluates, Prints, and Loops back to the prompt. It allows the user to run or evaluate any Python statement or syntax before committing the syntax to a script. It will continue until the functions `exit()` or `quit()` are run, or `<CTRL>Z` on Windows or `<CTRL>D` on UNIX/Linux.

One of the more useful commands at first is the built-in `help()` function.

Python can use some optional environment variables to configure the way it runs, but these are rarely needed. See the online documentation for details.

IDLE is a simple GUI useful for debugging and development and is bundled with Python. And Eric would approve , 'Nudge Nudge,

'Wink, Wink, Say no more!'. Note that it might have problems with a personal firewall. On Linux, IDLE requires that Tk is installed.

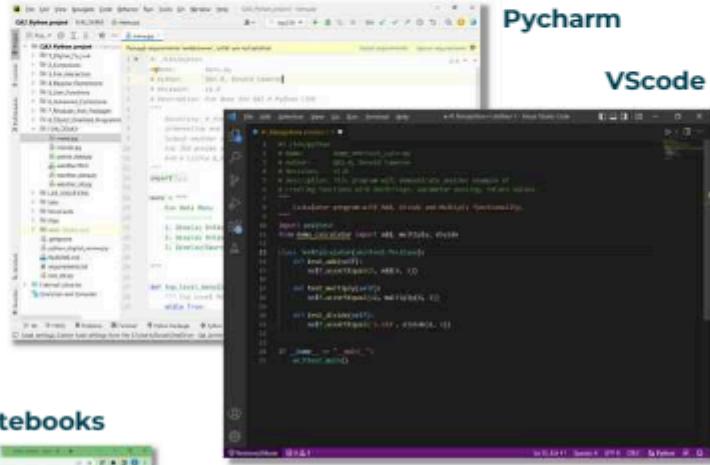
IDLE controls on MAC OS X are slightly different:

<CTRL>P	Previous statements
<CTRL>N	Next statements

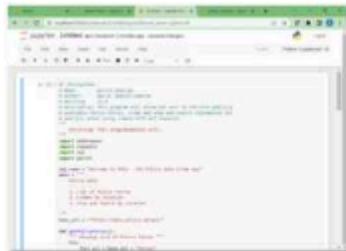
QA IDEs

Integrate Development Environments

- User Friendly GUI
- Built-in text editor
- Built-in debugger
- Integrated console
- Coloured Coded highlighting
- Code Search
- Integrate with Online Repository
- Support CI/CD



Jupyter Notebooks



Integrate Development Environments

- Web-based and shareable
- Supports 100+ languages
- Data Sciences
- Machine Learning

The Python REPL prompt and IDLE are very basic interfaces with limited features.

Even beginner Pythonistas will move quickly onto using more feature rich IDEs such as Pycharm or VSCode. These IDEs provide a more powerful interface, console, editor, and debugging features as well as integration with Version Control and online repositories like GitHub and BitBucket.

We even have web-based interfaces such as Jupyter Notebooks which is more focussed to the Data Science community.

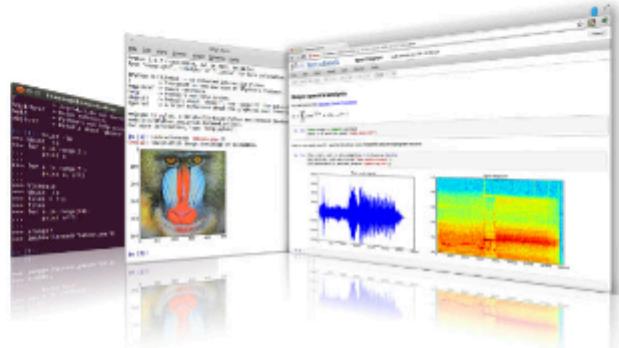
Q4 IPython and Jupyter

Interactive computing and data visualisation

- Powerful Interactive Shell
- Kernel for Jupyter
- Interactive data visualisation
- Support for Parallel programming

Jupyter Notebooks

- Fork of IPython in 2014
- Name derived from Julia, Python, and R
- Computational notebook for Literate Programming
- Support for multiple kernels



Source: <https://ipython.org/>

'Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.' - Donald Knuth

IPython or Interactive Python is a command line shell for interactive computing. It offers introspection (can examine the type of an object), rich media support (text, images, graphs etc.), shell syntax, tab completion and history.

It also provides a browser-based notebook style interface with support for code, text, math expressions, and interactive data visualisation such as plots. It is often used with Math and Data Science libraries such as NumPy, SciPy, and matplotlib.

In 2014, the IPython creator Fernando Pérez developed a spin-off project called Project Jupyter. IPython now exists as the kernel and shell for the Jupyter notebook inspired interface. Jupyter is language agnostic and gets its name to the core programming languages it supports - Julia, Python and R.

As such, Jupyter Notebook (formerly IPython Notebook) is a web-based interactive computational environment for creating, executing, and visualising data and supports over 100 kernels – IPython being one of them. The programs are written in a Literate

Programming style first introduced by Donald Knuth in 1984.

QA Python scripts

Python command line options

```
python [-options]... [-c cmd| -m mod|script file|-] [script arguments]...
```

Python scripts are compiled into byte-code

- Like Java, Perl, .NET, etc...
- Script files should have **.py** suffix
- Compiled modules have **.pyc** suffix
- Ensure script names do not conflict with standard modules names
- Use Sh-Bang (`#!/usr/bin/python`) in UNIX and Linux scripts

```
#!/usr/bin/python  
print('Hello World!')  
hello.py
```

```
chmod u+x hello.py  
./hello.py
```

Python programs (often called scripts) and modules usually have the file name suffix **.py**. Python modules (program components) are automatically saved in a semi-compiled format, with the suffix **.pyc**, to speed loading. The byte-code is automatically updated if the source file has a later timestamp but is not portable.

For command line options to python, see `python -h`.

When choosing your script names, ensure they do not have the same name as a Python module. If in doubt, use a naming convention, for example by prefixing the script name with a code. We shall discuss modules in more detail later.

It is good practice for Python scripts on UNIX and Linux to have a special comment called the shebang consisting of a `#!` followed by the pathname of the interpreter (of course Python) to execute the script. It means that users can execute the script on the command line using just the name of the script. The comment is ignored on Windows, but normal practice is to keep it in. It is also usually necessary to assign execute permission to the script on Unix and

Linux command line (`chmod u+x scriptname`). Modules do not require this shebang line and only require read permission.

The `-c` option is not used much but can be handy for small jobs. Specifying a dash instead of `-c` or a script file will read from standard input.

The `-i` option allows you to run a script but then enter the interactive interpreter. Any functions or variables declared in the script will be visible to the command-line (following the usual rules of scope).

The `-m` option runs an external module as a script. You can check the version of Python you are using with `python -V` (uppercase V). This course assumes Python 3.0 or later.

Q^ Python help

Online

- <http://docs.python.org>
- <http://docs.python.org/lib>

UNIX/Linux man pages

```
$ man python
```

Python pydocs

```
$ pydoc sys
```

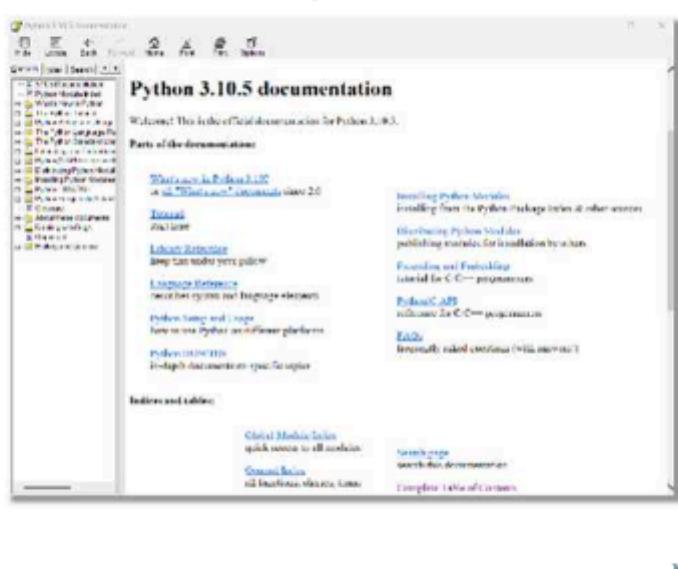
Python Shell – interactive help()



The screenshot shows the Python 3.10.5 documentation page. At the top, it says "Welcome! This is the official documentation for Python 3.10.5." Below that is a sidebar titled "Parts of the documentation" with links like "What's new in Python 3.10", "Index", "Library Reference", "Language Reference", "Python Setup and Usage", "Python Modules", and "Editor". The main content area has sections for "Index", "Global Modules Index", "General Index", and "Search this documentation".

```
Welcome to Python 3.10's help utility!
If this is your first time using Python, you should definitely check out
the tutorial on the internet at https://docs.python.org/3.10/tutorial/.
Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".
To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contains a given string such as "spam", type "modules spam".
help [
```

IDLE – Press <F1> for help



Python documentation is available online from the site shown, for example <http://www.python.org/doc/current/lib/modindex.html> is the online module reference. <http://docs.python.org/3.0/> is the Python 3 language reference.

IDLE has the documentation built-in, and on UNIX/Linux systems the man pages for python come bundled (try `man -k python` for extensions).

On UNIX/Linux the pydoc tool is usually available as a script, so just call `pydoc` keyword from the shell, for example:

```
pydoc sys
```

On Windows cmd.exe use the `pydoc.py` script in the Lib folder, for example:

```
C:\Python32\Lib\pydoc.py sys
```

There is also a useful TK GUI version usually bundled with the Windows release (in the Tools folder), but you may have to setup a shortcut to that.

QA PEP – Python Enhancement Proposals

Language design document

- Evolution
- Information
- New features

PEP 8 – Style guide for Python

- <https://peps.python.org/pep-0008/>
- Code layout
- Readability
- Consistency
- Indentation – four spaces!
- Dos and don'ts

The screenshot shows the Python Enhancement Proposals website. On the left, there's a sidebar with a 'Contents' section listing various PEPs. The main area is titled 'PEP 8 – Style Guide for Python Code'. It includes author information (Guido Van Rossum), status (Proposed), creation date (2001-03-01), and last update (2021-08-21). Below that is a 'Table of Contents' and an 'Introduction' section. A quote at the bottom reads: 'Code is read much more often than it is written.' - Guido Van Rossum.

'Code is read much more often than it is written.'
- Guido Van Rossum

Python Enhancement Proposals, or PEPs, are a design document detailing the evolution, features and general information of the Python language to the Python community. It should describe new features and a rationale for their inclusion and is a great resource to explain the reason for a new feature and when it was implemented and released.

The (large) index is here: <http://www.python.org/dev/peps/>, but is also available in the Python 3 documentation, and there are many links to PEPs within the help text.

One of the most important PEP documents is PEP 8. It is the Style Guide for Python code and details best practice guidelines in layout, readability, consistency and general Do's and Don'ts when writing Python code. It should be READ and REFERENCED often.

QA Example of a Python program

```
#!/usr/bin/python

# Example Python script

import sys

argc = len(sys.argv)

if argc > 1:
    print('Too many args')
else:
    where = 'World'
    print("Hello", where)

print('Goodbye from ' +
      sys.argv[0])
```

#! line for UNIX/Linux, ignored on Windows

Comment line

Load an external module

Variable assignment and function call

Condition is terminated by a colon:
Blocks require consistent indentation

print *inserts a space between parameters*

Statements are terminated by a new-line
unless inside brackets. Note the + used to
join strings

Can enclose string literals in either single or double quotes, but be consistent.

When a Python script is run, the code is first compiled then executed - much the same way as Java, C#, and many other languages. The compilation phase is very fast and there is (usually) no discernible delay. All the lines shown are optional, and right now we do not show the whole story – for example, functions and global variables are omitted.

The first line indicates to UNIX the name of the Python interpreter. It is preceded by a #, which also marks the start of a comment, so that on non-UNIX platforms the line is seen as a comment.

The **import** statement loads an external Python module. You will usually find this statement near the beginning of the script. There are a many external modules available, this one, sys, gives us details of the runtime environment. More on modules later.

A variable assignment is shown - we do not formally declare the variable type. The if condition is terminated by a colon and membership is by indentation. This also applies to other conditions such as while loops.

We are using the **print()** function. Prior to Python 3, print was a statement and parentheses were not required around the argument list. It writes text to the standard output stream (STDOUT), which is usually the terminal screen, and prints a new-line by default.

Statements are usually terminated by a new-line, but there are a few alternatives. A new-line which appears inside brackets (parentheses, square brackets or braces) is just considered to be another white-space character. A new-line may be 'escaped' by placing a \ in front. If you must, several statements may be placed on the same line terminated by semi-colons. But we can C what you are doing and will disapprove!

QA Modules

Most Python programs load other Python code

- Standard Library modules bundled with Python
- Downloaded extensions, or modules written locally

```
>>> import sys  
>>> print(sys.platform)
```

**Find & compile 'sys'
Must specify the module name**

```
>>> from sys import *  
>>> print(platform)
```

**Find & compile 'sys', and import all
names to our namespace**

Examples:

- Operating system specific code - os
- Interface to the runtime environment - sys
- Scientific libraries like NumPy, DISLIN, SciPy, and many others
- Python's built-in functions are in the "built-ins" module
- Automatically imported

```
>>> help("builtins")
```

Modules are files of Python code compiled and run as part of the main program. In fact, even the main program is really a module - called 'main'. Each module has its own namespace, which means that variables and functions created within them will not be confused with others of the same name in different modules.

Modules are bundled with Python, and often used. A list of those issued with the system can be found in the online help. Modules can also be downloaded, usually from <http://pypi.python.org/pypi> (note the side-bar for modules specifically for Python 3). Python usually finds its modules among directories listed in the environment variable PYTHONPATH, or in Python's lib directory.

Help text for modules can be obtained in Idle by typing `help(module-name)` or `help()` to get the `help>` prompt and enter `modules` to list the installed modules. You can then enter the name of the module to see documentation on that module.

We will be looking into writing our own modules later...

QA Functions and built-ins

A function is a named block of program code

- It can be passed values, which might be altered
- We can write our own functions
- It can return a value

```
lhs = function_name(arg1, arg2,...)
```

Python includes many built-in functions in the compiler

- Called (remarkably) "builtins" or `__builtins__` and viewed as a module
- Always available - no need to import
- Examples: `print()`, `len()`, `str()`, `list()`, and `set()`
- See the on-line documentation
 - The Python Standard Library > Built-in Functions
 - Summary list at the end of this chapter

Functions are blocks of code provided to carry out a specific task. There are many supplied with Python, which adds greatly to the functionality of the language. A list is given after the chapter summary but you can use the online documentation for details. The difference between built-in functions and those in a module is that they do not require any external file to be included - they are always available in your program and are very fast.

To view a list of the built-in functions, use `help("builtins")` or `help(__builtins__)`.

In Python 2.6 the Python 3.0 built-ins were available in a module called `future_built-ins`.

QA

SUMMARY



Python is a free fully functional language

- Extensive on-line documentation

Python Scripts can be run:

- From scripts
- Interactively
- Or from an IDE

Python syntax is different!

Python syntax requires good indentation!

- And is intentional!

Using external modules is commonplace

- Load a module by using `import`

Built-in functions are:

- fast, always available, and useful!

»

QA

PYTHON BUILT-IN FUNCTIONS (1)



```
abs(x)
all(iterable)
aiter(iterator)
any(iterable)
anext(iterator)
ascii(object)
bin(x)
bool([x])
breakpoint(function)
bytearray([arg[, encoding[, errors]]])
bytes([arg[, encoding[, errors]]])
callable(object)
chr(i)
classmethod(function)
compile(source, filename,
        mode[, flags[, dont_inherit]])
complex([real[, imag]])
delattr(object, name)
dict([arg])
```

```
dir([object])
divmod(a, b)
enumerate(iterator[, start=0])
eval(expression[, globals[, locals]])
exec(object[, globals[, locals]])
filter(function, iterable)
float([x])
format(value[, format_spec])
frozenset([iterable])
getattr(object, name[, default])
globals()
hasattr(object, name)
hash(object)
help([object])
hex(x)
id(object)
input([prompt])
int([number | string[, radix]])
isinstance(object, classinfo)
```

20

This slide, and the one which follows, is meant for reference - you are not expected to remember these! You will probably only regularly use about a quarter of them anyhow - not many experienced Python programmers could produce this list from memory.

This is not a substitute for the main online documentation.

QA

PYTHON BUILT-IN FUNCTIONS (2)



```
issubclass(class, classinfo)
iter(o[, sentinel])
len(s)
list([iterable])
locals()
map(function, iterable, ...)
max(iterable[, args...], *[, key])
memoryview(obj)
min(iterable[, args...], *[, key])
next(iterator[, default])
object()
oct(x)
open(file[, mode='r'], buffering=None
      [, encoding=None],
      errors=None
      [, newline=None,
      closefd=True]]))])
ord(c)
pow(x, y[, z])
```

21

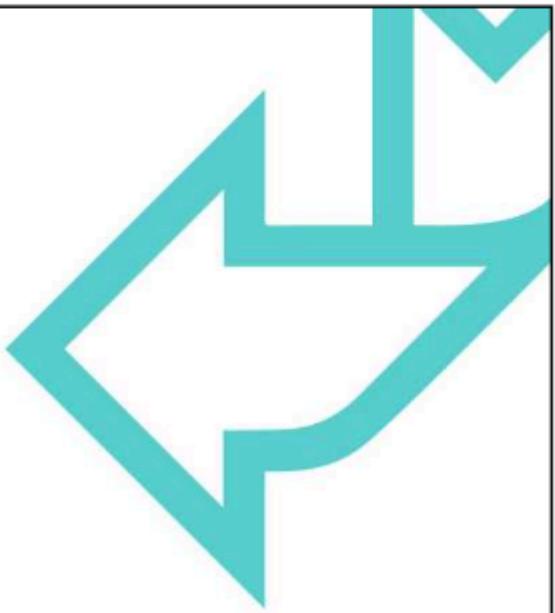
This slide, and the one which follows, is meant for reference - you are not expected to remember these! You will probably only regularly use about a quarter of them anyhow - not many experienced Python programmers could produce this list from memory.

This is not a substitute for the main online documentation.



Python 3 Programming

Fundamental variables



QA

FUNDAMENTAL VARIABLES



Contents

- Python objects
- Python variables
- Type specific methods
- Augmented assignments
- Python types
- Python lists
- Python tuples
- Python dictionaries

Summary

- Python operators
- Python reserved words

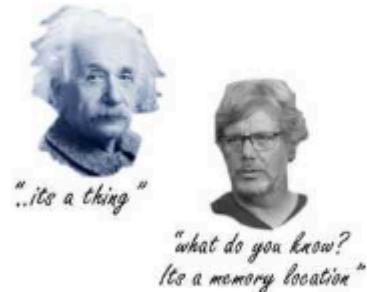
2

This chapter discusses the basic building blocks of a Python program - its object types and variables.

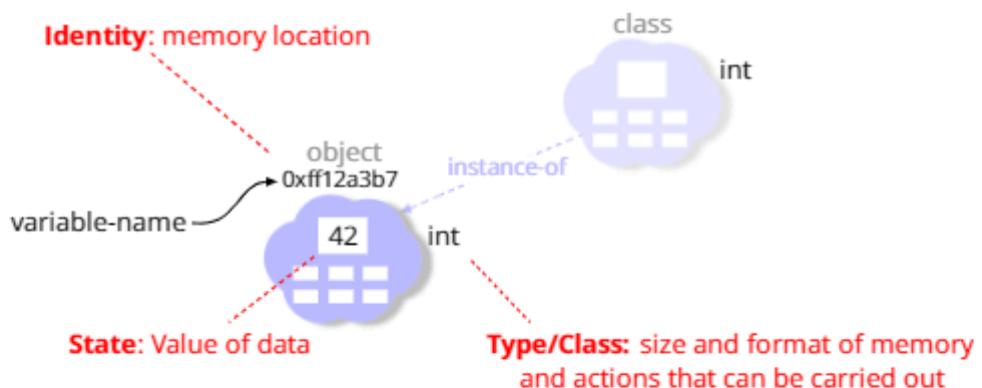
QA Python is object oriented

So, what is an object?

- To a mathematician, the term describes a 'thing'.
- To a programmer, the term describes a specific area of memory.



Objects have type, state, and identity



Object orientation is inescapable if you wish to understand Python and have even more fun with it. You do not need to write OO code, but it is important to understand the principles, which are not that complicated.

Many programming languages have the concept of variable types, which is roughly analogous to a class - it describes the object to which the variable is referring. How that object is laid out in memory is not particularly important to us, but it is important to Python - it must know the size and format of memory required, and it is up to us to describe it.

Fortunately, the common object types (classes), like strings, files, and exceptions, are already defined.

When we create an object, it is 'instance of' an existing class (built-in or user defined) and will inherit the structure of the class it's based on. It will be assigned a memory location, and we can assign a suitable value to be stored within. It is normal to store the object location in a named variable for reuse. So, a simple definition of a

variable in Python, is a named identifier referencing an object in memory.

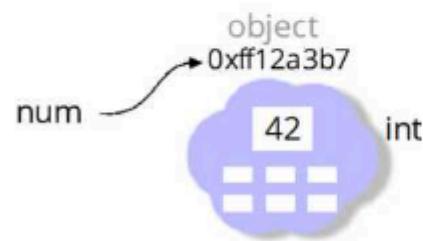
You can test the class of an object by using the `isinstance()` or `type()` functions:

```
>>> name = "Guido"
>>> isinstance(name, str)
True
>>> type(name) == str
True
```

In Python, an object's identity can be obtained using the `id()` built-in function, although this is rarely needed. Are classes types? Not really, we shall discuss this later (hint: duck-typing).

QA Python variables

Python variables are references to objects



Variables are defined automatically

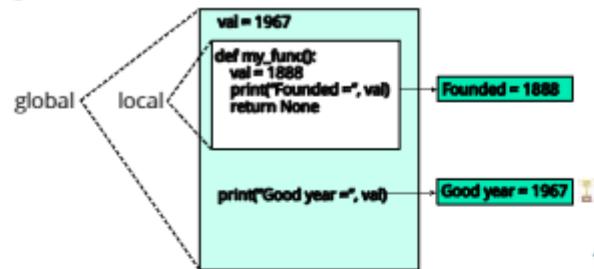
- An undefined variable refers to a special object called None

Variables can be deleted with del

- An object's memory can be reused when it is no longer referenced

Variables are local by default

- If created within a user written function
- More on global variables and scope later...
- Display local variables with `print(locals())`
- Display global variables with `print(globals())`



Like most scripting languages, Python variables are defined automatically, and are untyped until assigned. Variables are actually references to objects, so the assignment of a string to a variable makes that variable reference a string object. Uninitialised variables reference an object called None (NULL or undef in other languages).

Being objects, class specific functions, like altering the case of a string, are implemented as methods calls on the object. If you are not familiar with this terminology, a method call is like a conventional function call, with a reference to the variable passed automatically. We shall see examples shortly.

Where the objects themselves are immutable (cannot be altered), then several variables may reference the same physical value.

Unlike most scripting languages, variables defined in a function are local by default - and must be specifically marked as global if required. This is a good thing!

When you delete a variable, then that removes the name. That will

decrement the object's reference count, and when the count reaches zero then the memory can be reused.

QA Variable names

Naming rules

- CaSe sensitive.
- Must start with an underscore or a letter.
- Followed by any number of letters, digits, or underscores.

Conventions with underscores

- Names beginning with one underscore are private to a module/class.
`_private_to_module`
- Names beginning with two underscores are mangled.
`_private_to_class`
- Names beginning and ending with two underscores are special.
`_itsakindamagic_`
- The character `_` represents the result of the previous command.

The names given to variables, and to other symbols like functions, follow the usual rules common to many programming languages. For example, names are case sensitive. Variable names must not clash with Python keywords: these may be listed with help ('keywords') or consult the list at the end of this chapter.

In addition to the rules, we have a number of conventions which are followed by the interpreter and programmers concerning underscores.

A name (variable, function, method) prefixed by a single underscore indicates the name is meant for internal use only in a module or class. It's not really private like other languages (Java), but merely a hint to programmers to not access the names. Additionally, the interpreter will not import the names when using a wildcard import (`from moduleA import *`) but this should be avoided in PEP008 compliant code.

A name prefixed with a double leading underscore (dunders) has its name mangled (changed) by the Python interpreter to avoid naming conflicts in subclasses. It will have `_modulename` or `_class` prefixed to its name. By default, all variables and methods are virtual in that they are inheritable and can be overridden by a subclass.

Using the dunders makes it private to the specific class in which they are used and not accessible to its inherited child classes.

The upshot of these is that you should never have names of your own with both leading and trailing underscores - these should be reserved for system use. A single underscore prefix means that the name is not imported from a module, and names with two leading underscores are mangled to stop unintentional access. We shall be seeing examples of these conventions later.

QA Type specific methods

Actions on objects are done by calling *methods*

- A method is implemented as a *function* - a named code block.
`object.method ([arg1[,arg2...]])`
- *object* need not be a variable.

Which methods may be used?

- Depends on the Class (type) of the object.
- `dir(object)` lists the methods available.
- `help(class.method)` for help on specific method.

Examples:

<code>name.upper()</code>	<code>names.pop()</code>
<code>name.isupper()</code>	<code>mydict.keys()</code>
<code>names.count()</code>	<code>myfile.flush()</code>

Just about everything in Python is an object, and carrying out an operation on an object, or querying an attribute, is often done by calling a function - more appropriately termed a method - from the object itself. There are several advantages to this system compared to just calling a general function. For example, you know that you are operating on the correct type - there is no other way.

In the examples, we have made up some variables:

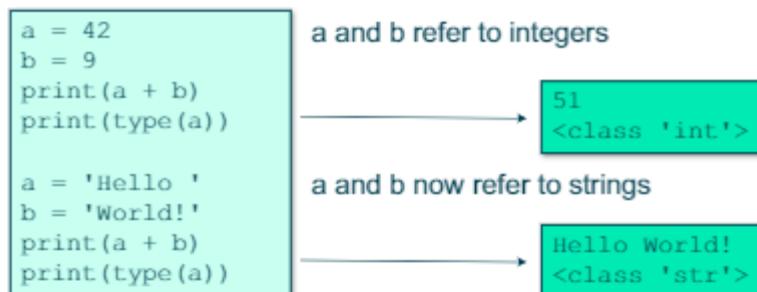
`name` is a string object
`names` is a list object
`mydict` is a dictionary object
`myfile` is a file object

The method names shown are mostly self-explanatory, but full documentation for them (and all the others) is available in the online help. The `help()` function prints help text (docstrings) for the class and all the methods in that class. An alternative is `print(object.__doc__)` which displays the docstring for the class only.

QA Operators and type

An operator carries out an operation on an object

- Produces a result which does not (usually) alter the object.
- The operation depends on the Class (type) of the object.
- List the Class using the `type` built-in function.



- A list of Python operators is given after the chapter summary.

7

An operator is usually a symbol (see the slide after the summary) which carries out an operation on a single object (unary operator) or between two operators (binary operator). The result has a value, which may be passed to a function (like `print`) or used on the right-hand side of an assignment.

Arithmetic operators like `+` (plus), `-` (minus), `*` (multiply), `/` (divide), and `%` (modulus) have familiar meanings when used with numbers, but what do they mean when used with strings?

Operators have different meanings depending on the type of object they are operating on, some of which are not at all obvious. For example, `%` does string formatting - something that you would not have guessed! Therefore, it is imperative that we know what type (class) of object we are dealing with, and the `type` function will tell us that.

QA Augmented assignments

A convenient shorthand for some assignments

```
stein = 1  
pint  = 1
```

```
stein = stein + pint
```



```
stein += pint
```

Use any arithmetic operator

```
lhs = lhs + rhs  
lhs = lhs - rhs  
lhs = lhs * rhs  
lhs = lhs / rhs  
lhs = lhs % rhs
```



```
lhs += rhs  
lhs -= rhs  
lhs *= rhs  
lhs /= rhs  
lhs %= rhs
```

*Augmented assignment is an assignment!
It has a result, which is usually ignored.*

Augmented assignments are called Compound assignments in some languages and come from an ancient language now lost in the mists of the distant past (C).

The expression `a += b` can be read as 'a is incremented by the value of b'. Therefore, in the code above, the variable `stein`, initially at 1 is incremented by the value of `pint`, which is 1. After the assignment, the value of `stein` becomes 2 and the value of `pint` remains 1.

Augmented assignment operators are more succinct than the long hand approach, so programmers tend to write expressions like this:

```
total += subtotal  
geometric *= progression
```

Rather than like this:

```
total = total + subtotal  
geometric = geometric * progression
```

Usually, the resulting code generated will be the same. Like other

operators, their meaning depends on the class of the object, for example `+=` on a string means append.

```
a = 'Hello'  
b = 'World!'  
a += b
```

`print(a)` gives Hello World! and a new string object is created. This operation is optimised on CPython. Python does not support post and pre increment `++` and decrement `-` operators like C and Perl, as it doesn't need them and would just add bloat to the core language. To increment and decrement in Python simply use `+1` and `-1`.

QA Python 3 types



Some of these types are obvious, but some require an explanation. Note the notation for octal (base 8) numbers. In old releases of Python, any number starting with a 0 (zero) would be an octal value, in Python 2.6 the prefix 0o (zero, lowercase oh) was introduced, and at Python 3 the leading zero no longer means octal. Old Python also used a trailing L to mean 'long' and a trailing 'U' to mean 'unsigned' - both are now removed.

Strings of text are objects, and once a variable of this type has been created, a string method can be called on it.

Numbers, strings, and Tuples are immutable, that is they cannot be altered. References to them can change to refer to different values, but the values themselves cannot. This enables Python to save space by storing just one instance of literals used in a program, regardless of how many references there are to it.

Strings, Lists and Tuples are ordered collections of objects, also known as sequences. We have a chapter on string handling later followed by a chapter on lists and tuples.

Dictionaries are collections of objects accessed by a key and are like associative arrays in awk and PHP, and hashes in Perl and Ruby.

Sets were introduced into Python 2.4 and are described in a later chapter. There is also an immutable set: frozenset. Lists, Tuples, Dictionaries, and Sets are known as collections, and are discussed in more detail in the Collections chapter.

A byte object is an immutable array of 8-bit values, whereas a bytearray is a mutable array of 8-bit values.

QA Switching types

Sometimes Python switches types automatically

```
num = 42
pi = 3.142
num = 42/pi
print(num)
```

num gets automatic promotion

13.367281986

Sometimes you have to encourage it

- This avoids unexpected changes of type.

```
port = 80
print("Unused port: " + port)
TypeError: Can't convert 'int' object to str implicitly
```

- Use the `str()` function to return an object as a string.
- Use `int()` or `float()` to return an object as a number.
- Other functions available to return lists and tuples from strings.

```
print("Unused port: " + str(port))
```

Unused port: 80

Like most modern languages, Python converts between numeric types automatically, generally from the narrow to the wider if there is a choice. If you really want an integer division (so the result is truncated), then use the `//` operator. However, with other types things are not so clear-cut.

A common error when beginning Python is the type error shown. Python does not know if the `+` means string concatenation or arithmetic - mixing objects of a different type is bad coding. We must explicitly tell Python what to convert and when - there is no hidden magic here.

One aid to typing is to use a naming system for your variables, for example a modified Hungarian notation which prefixes a string variable with '`s`', an integer with an '`i`', a List with an '`l`', and so on. For example: `iCount`, `sName`, `lNames`. This is, effectively, inventing your own sigil system. But this is not commonly used in Python.

Where we appear to call functions like `str()`, `int()`, and `float()` [and `tuple()`, `list()` and `dict()` which we will see later] they are really the

name of the class, so what we are really doing is constructing an object.

You can find out the memory size of an object by using the `sys.getsizeof()` call, for example:

```
import sys
print("Size of count", sys.getsizeof(count), "bytes")
print("Size of str(count)",
      sys.getsizeof(str(count)), "bytes")
```

Q Python lists introduced

Python lists are similar to arrays in other languages

- Items may be accessed from the left by an index starting at 0.
- Items may be accessed from the right by an index starting at -1.
- Specified as a comma-separated list of objects inside [].

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg']
print(cheese[1])
cheese[-1] = 'Red Leicester'
print(cheese)
```

Stilton
['Cheddar', 'Stilton', 'Red Leicester']

Multi-dimensional lists are just lists containing others:

```
cheese = ['Cheddar', ['Camembert', 'Brie'], 'Stilton']
print(cheese[1][0])
```

Camembert

Lists are objects containing a sequential collection of other objects, commonly called elements. Elements may be accessed by a position (counting from zero) specified within [] which is common in other languages. Indexing from zero is a convention for almost all languages, likely inherited from BCPL and C, and likely chosen due to pointer memory addressing starting from zero and counting from zero is more efficient. Mathematicians would say it's just natural!

Lists are Mutable, or able to be changed, so they are like arrays in some languages. They are dynamic in that they may be extended or shrunk. New items may be added or removed anywhere with the list.

We discuss lists in more detail in the Collections chapter.

Q^ Python tuples introduced

Tuples are *immutable* (read-only) objects

- Specified as a comma-separated list of objects, often inside ().
- Can be specified inside () sometimes required for precedence.
- The comma makes a tuple, not the ().
- Can be indexed in the same way as lists.
- Starting from 0 on the left or -1 on the right.

```
mytuple = 'eggs', 'bacon', 'spam', 'tea'
print(mytuple)
print(mytuple[1])
print(mytuple[-1])
```

('eggs', 'bacon', 'spam', 'tea')
bacon
tea

- Can be reassigned, but not altered.

```
mytuple[2] = 'spam'
TypeError: 'tuple' object does not support item assignment
```

Tuples are Immutable (read only), for example if you attempt to append:

```
TupleVar.append('Vikings')
Traceback (most recent call last):
  File "liststuples.py", line 6, in ?
    TupleVar.append('Vikings')
AttributeError: 'tuple' object has no attribute 'append'
```

The process of assigning values to a tuple is called packing. And likewise, the process of extracting values from a tuple to variables on the lhs (of an assignment) is called unpacking. But you cannot assign to elements within a tuple as it is immutable.

It may seem strange that Python has two seemingly similar types, tuples and lists. While lists are more flexible than tuples, there is a penalty to pay in performance overhead. In most cases, where either could be used, tuples are most efficient than lists.

Although parentheses are often associated with tuples, these are usually optional. So, in the example on the slide, the following is

equally valid and will produce the same result:

```
mytuple = ('eggs', 'bacon', 'spam', 'tea')
```

We discuss tuples further in the Collections chapter.

Q A Python dictionaries introduced

A Dictionary object is an "ordered" collection of objects [see notes about ordering]

- Constructed from {} or dict().

```
mydict = {'key1':object1, 'key2':object2, 'key3':object3}
```

- A key is a text string, or anything that yields a text string.

```
mydict['key4'] = object4
```

- Example:

```
mydict = {'Australia':'Canberra', 'Eire':'Dublin',
          'France':'Paris', 'Finland':'Helsinki',
          'UK':'London', 'US':'Washington'
        }
print(mydict['UK'])

country = 'Iceland'
mydict[country] = 'Reykjavik'
```

London

Dictionaries are just like associative arrays in awk and PHP, or hashes in Perl and Ruby. They are constructed from lists of key:object pairs, inside curly braces, although you may also assign them using the dict() function.

For example:

```
mydict = dict(Sweden='Stockholm', Norway='Oslo')
```

The key is a text string, while the value is an object of any valid class, including a list, tuple, or dictionary. No special syntax is required to access them.

Traditionally in computing, this type of structure is considered an unordered collection, but becomes an ordered collection in CPython 3.6 and a language feature across all Python platforms from Python 3.7. It is ordered in 'insertion order'.

A list of the keys may be extracted using the keys() method, and values with the values() method.

We discuss dictionaries, and their related type sets, in the Collections chapter.

QA

SUMMARY



A Python variable is a reference to an object.

Python variable names are case-sensitive

- Watch out for leading underscores.

Variables are accessed using operators and methods.

- `dir(object)` lists the methods available.

Lists are like arrays in other languages.

Tuples are "immutable".

- But can contain variables.

Dictionaries store objects accessed by keys:

- Keys are unique.
- Keys are not ordered.

QA

PYTHON OPERATORS



<code>or</code>	logical OR
<code>and</code>	logical AND
<code>not</code>	logical NOT
<code>< <= > >=</code>	comparison operators
<code>== !=</code>	equality operators
<code>is</code>	object identity test
<code>in</code>	object membership test
<code> ^</code>	binary OR, XOR
<code>&</code>	binary AND
<code><< >></code>	binary shift
<code>- +</code>	subtract, add
<code>* / // %</code>	multiply, divide, integer-divide, modulo
<code>@</code>	matrix multiplication (3.5)
<code>~ **</code>	complement, exponentiation
<code>await</code>	await expression (3.5)

The operators listed in reverse order of precedence (or is the lowest precedence).

Difference between / and // is best shown as an example:

<code>x = 2</code>	
<code>y = x/3</code>	gives 0.666666666667
<code>y = x//3</code>	gives 0

Await is used with coroutines and becomes a keyword in 3.7, along with `async`. Both were introduced at 3.5 and require the `asyncio` module. This module is current provisional and may include changes that are not backward compatible.

The `@` operator (`matmul`) is intended for matrix multiplication and has the same precedence as multiplication. No built-in types currently support this operator, it is intended for third-party modules.

QA

PYTHON RESERVED WORDS



The following are illegal as variable or function names in Python:

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

Briefly, the meanings of these reserved words are:

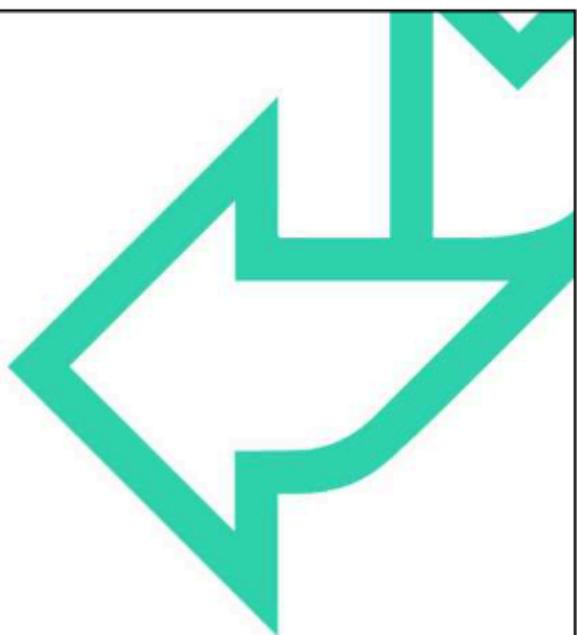
and, not, or	logical operators
assert, raise	trigger an exception
async, await (from 3.5, reserved at 3.7)	used with async coroutines
break	exit the current loop
class	create a class object
continue	do the next iteration of the
current loop	
def	create a function object
del	delete an item from a list
except	indicates an
exception handler for a try block	
exec	execute code
finally	statements always executed
after a try block	
for	sequence iteration loops
from	used with 'import' to specify
imported names	
global	declare variable as

global	
if, else, elif	conditional clauses
import	find and load a
module	
in	tests sequence membership
is	identity test
lambda	create an anonymous
function	
pass	empty statement (no-op)
print	write to stdout, appending a
"\n"	
return	return a value
from a function	
try	catches exceptions
while	conditional loop statement
with, as	a context resource manager
yield	creates a generator function



Python 3 programming

Flow control



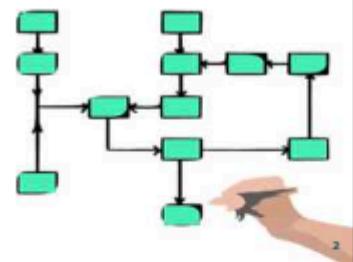
QA

FLOW CONTROL



Contents

- Python conditionals
- What is truth?
- Boolean and logical operators
- Chained comparisons
- Sequence and collection tests
- Object types
- While loops
- For loops
- Conditional expressions
- Unconditional closedown



Now we look at decision making, if statements, and loops. Finally, we look at how to give-up and exit our program.

Q^ Python conditionals

Conditional membership is by *indentation*

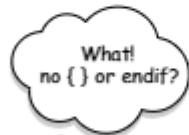
- Designed for readability.

```
if condition:  
    statements  
elif condition:  
    statements  
else:  
    statements
```

- Boolean operators are overloaded by type.
- No need for different text or numeric operators.

```
if lista == listb:  
    print("Same!")  
  
if "eggs" in lista:  
    print("It eggists!")
```

```
temp = 0  
  
if temp < 10:  
    print("That's cold")  
elif temp <= 15:  
    print("That's mild")  
else:  
    print("That's warm")
```



The Python syntax for conditional statements is very simple. The condition is terminated by a colon, and membership of the block, which follows, is shown by consistent indentation. The indentation of the first line of the block is expected in lines which follow. A change in indentation, marks the end of the block (or an error if Python cannot figure out what you mean). The white-space used can be any amount but must be consistent.

The usual set of comparison operators are supported:

> < == >= <= or and not in

No special operators are required for built-in types (like comparing strings), the standard operators are overloaded.

Q&Python conditionals

Testing for System and Platform information

- Common to test for system script it is running on.
- Can be used to setup different file paths and environment variables.
- Can use std library modules sys and platform.

```
import sys
if sys.platform == "win32":
    home_dir = os.environ["HOMEDIR"]
else:
    home_dir = os.environ["HOME"]
```

Capture env variables from shell

- Platform module also provides other information.

```
import platform
if platform.system() == "Windows":
    print("Machine -", platform.machine())
    print("Hostname -", platform.node())
    print("OS -", platform.platform())
    print("Python -", platform.platform())
```

AMD64
DESKTOP-A600EVV
Windows-10-10.0.1.19045-SP0
CPython

A common test in a script is to test which platform the script is executing on. Useful for setting up different paths or environment variables in the script and therefore making the script more portable.

The sys module, so called as it provides access to system specific variables and functions that interact with the Python interpreter. There is a string object called 'platform' that holds the current platform. Possible values include 'linux', 'posix', 'win32', 'cygwin', 'Darwin', and 'openbsd6'.

Alternatively, we could use the *platform* module, and the *platform.system()* function to return the current platform, for example 'Windows'. It also provides additional functions for finding the architecture, machine type, hostname, processor and which Python platform we are using – for example, 'Cpython'.

The os module provides miscellaneous operating system interfaces and can also return the operating system with *os.name*, for example 'nt'. But if the sys and platform equivalent provide more

granular information and are preferred.

All three modules are part of the Python Standard Library.

Q\Python conditionals

Match Case (introduced in 3.10)

- When if /elif /else statements become too vast, you can use a Python match in its place.
- Considered to be easier to read than multiple elif statements.

```
http_code = "418"

match http_code:
    case "200":
        print("OK")
    case "404":
        print("Not Found")
    case "418":
        print("Some other problem")
    case _:
        print("Code not found")
```

Some other problem

Python was considered behind the times as this structure existed in many languages for a long time. Introduced in python 3.10, this was the first official method that allowed python coders to use a match, although there were many documented 'get arounds' for solutions.

By too vast, we mean any more than 3 elifs!

QA Indentation

Python? That is for children. A Klingon Warrior uses only machine code, keyed in on the front panel switches in raw binary.

Klingon Programmer

The Future

We will perhaps eventually be writing only small modules which are identified by name as they are used to build larger ones, so that devices like *indentation*, rather than delimiters, might become feasible for expressing local structure in the source language.

Donald Knuth 1974

Guidelines from PEP008:

- Spaces are the preferred indentation method.
- Use 4 spaces per indentation level.
- Python does not allow mixing tabs and spaces.



The Donald Knuth quote is from 'Structured Programming with go to Statements', written in 1974.

How much indentation should be used? It is generally thought that 4 spaces is the optimum, but it really does not matter if you disagree - pick the indentation that you feel is the best and stick with that. Tabs are usually a bad idea because their size varies between editors. What looks good in one, may be terrible in another.

In addition to the guidelines from PEP008 shown:

Indentation

 Use 4 spaces per indentation level.

Tabs or Spaces?

 Never mix tabs and spaces

 Tabs should be used solely to remain consistent with code that is already indented with tabs.

QA What is truth?

Built-in function `bool()` tests an object as a Boolean

- False : 0, None, empty string, tuple, list, dictionary, set
- True : everything else
- Constants True and False are defined

Use double equal signs (`==`) to compare values

- Overloaded for built-in types
- Use `is` to compare identities of two objects

Sequence types and dictionaries also support `in`

- Tests membership of the container

```
lang = ['Perl', 'Python', 'PHP', 'Ruby']
if 'Python' in lang:
    print('Python is there')
```

Unlike C based languages, not every statement in Python has a Boolean value. For example, assignments cannot be mis-read as a comparison in Python, unless you specifically wrap it inside a `bool`.

The comparison operator `==` is overloaded for different classes and compares the values of two objects. If you wish to test if two variables refer to the same object, use `is`.

A useful operator (from awk) is `in`. This tests for membership of any sequence, including strings, lists, and dictionaries (where it tests for a key). For example:

```
>>> slang = "We luv Python"
>>> if 'Python' in slang:
>>>     print("Python is in slang")

>>> dlang = {'Perl': 'sigils', 'Python':
'indentation', 'PHP': 'functions', 'Ruby': 'Rails'}
>>> if 'Python' in dlang:
>>>     print("Python is there: " + dlang['Python'])
```

Strangely, in Python 2 True and False are variables? So, this was legal, if silly:

```
>>> True = False
```

Fortunately, that gives a SyntaxError in Python 3.

Q\Comparison and Boolean operators

Comparison operators

<code><</code>	value less than	<code>expression < expression</code>
<code><=</code>	value less than or equal	<code>expression <= expression</code>
<code>></code>	value greater than	<code>expression > expression</code>
<code>>=</code>	value greater than or equal	<code>expression >= expression</code>
<code>==</code>	value equality	<code>expression == expression</code>
<code>!=</code>	value inequality	<code>expression != expression</code>
<code>is</code>	object identity is the same	<code>object is object</code>

Boolean operators

<code>not</code>	logical NOT	<code>not expression</code>
<code>and</code>	logical AND	<code>expression and expression</code>
<code>or</code>	logical OR	<code>expression or expression</code>

The Boolean (true/false) operators in Python are conventional in most ways. Unlike some languages, however the same operators are overloaded for different types. For example, these operators may be used to test numbers, strings, lists, or dictionaries. What happens when you mix incompatible types is discussed later.

The odd operator out in the list is 'is'. All the other operators test the values placed either side of them, is tests the identity of the references themselves to see if they refer to the same object.

Python 2 not only had `<>`, but also a `cmp` built-in function which is now deprecated.

QA Chained comparisons

Consider a test with multiple related expressions:

- Number must be between 0 and 41 and distance must be above 42:

```
if 0 < number and number < 42 and 42 < distance:  
    print("number and distance are within range")  
else:  
    print("number and distance are out of range")
```

- Could be rewritten as a chained comparison:

```
if 0 < number < 42 < distance:  
    print("number and distance are within range")  
else:  
    print("number and distance are out of range")
```

- And can be combined with other tests:

```
if 0 < number < 42 and distance != 20:  
    ...
```

A useful shortcut for complex tests (in 'if' statements or 'while' loops) are chained comparisons. These enable the programmer to write a more readable test when a range is concerned.

They can become fairly complex, as our first example shows (and can get worse than that). In principle, you should write code that is easy to understand. Only use a chained comparison, if it makes the code simpler - remember the Zen of Python.

Q& Sequence and collection tests

An empty string, tuple, list, dictionary, set returns False:

```
mylist = [0, 1, 2, 3]
if mylist:
    print("mylist is True")
```

mylist is True

Sequences also support built-in functions **all()** and **any()**:

- **all()** returns True if all items in the sequence are true.
- **any()** returns True if *any* of the items in the sequence are true.

```
mylist = [0, 1, 2, 3]
if not all(mylist):
    print("mylist: not all are True")
if any(mylist):
    print("mylist: at least one item is True")
```

mylist: not all are True
mylist: at least one item is True

An empty list is not the same as a list which has not been defined. A list, and any sequence (like a string), or dictionary, can exist but be empty, in which case it also has the Boolean value False.

Sequences also support the built-in functions **all()** and **any()**, which are occasionally useful. The **all()** function returns True if all its objects evaluate to True, whilst **any()** returns True if any evaluate to True.

QA Object types

Beware of comparing objects of different types.

- Comparison operators may be overloaded.
- Do not expect automatic conversion.

```
num = 42
txt = '3'

if txt < num:
    print('Wow!')
else:
    print('Doh!')

if int(txt) < num:
    print('Wow!')
else:
    print('Doh!')
```

TypeError: unorderable
types: str() < int()

Before Python 3 this
was not an error, we
just got the wrong
answer!

Wow!

A drawback of Python's use of overloaded operators is when a binary operator has two dissimilar objects on each side. In old releases of Python, it guessed which object's operator to use, and its guess was not always what the programmer intended. Fortunately, that has been fixed in Python 3, and we get an error message (Exception) as expected.

It is important to force conversion to the correct type, often using the `int()` or `str()` functions.

This is the price we have to pay for simpler syntax, the alternative would be to have multiple operators (Perl) or a plethora of built-in functions for comparisons (PHP).

QA A note on exception handling...

An exception is Python's way of telling you something

- Unless handled, it will halt the program.

Many Python built-in functions can raise an exception

- When they wish to indicate some condition.

Exceptions do not necessarily indicate failure

- For example:
 - Search for something which does not exist.
 - Unable to open a file.

At this point in the course, we will just live with them.

Later, we will discuss how to handle exceptions



The `TypeError` shown on the previous slide was actually an Exception. It can be trapped, and the error message changed, as follows:

```
try:  
    if txt < num:  
        print('Wow!')  
    else:  
        print('Doh!')  
    except TypeError:  
        print("Invalid types compared", file=sys.stderr)
```

The `try` block contains code to be tested. Should any of that code (which is often a function call) raise an exception, then it can be trapped, and code executed in the `except` block to handle it. If an exception list is specified then the handler code will only be executed if the exception is in the list, otherwise the stack will be unwound until a handler is found. Unhandled exceptions terminate the program. Python exception handling also supports a `finally` and `else` block.

We discuss exception handling in more detail later.

QA While loops

Loop while a condition is true

- Python only supports entry condition loops.
- There is no *do...while* loop.
- Membership is by indentation.

while condition:

loop body

```
line = None  
  
while line != 'done':  
    line = input('Type "done" to complete: ')  
    print('<', line, '>')
```

```
myl = [23, 67, 32, 9, 77]  
  
while myl:  
    print(myl.pop() * 2)
```

```
i = 0  
  
while i < 10:  
    print(i)  
    i += 1
```

154
18
64
134
46

pop() on a list
removes and
returns the last
item

Loops follow a similar syntax to the if statements, membership of the body of the loop is by consistent indentation. The usual while loops and list processing for statement are available.

Somewhat unusually, in Python there is no do.. while loop.

The first example is fairly simple but indicates that the tested variable must already exist. Missing out that initialisation would result in a NameError exception.

The second example loops while the list is true and not empty. We are using the pop() list method (which we shall see later) which removes the last (right-most) item from a list and returns it.

QA Loop control statements

Loop control statements

- `continue` perform next iteration
- `break` exit the loop at once
- `pass` empty placeholder (do nothing)

The else: clause

- Indicates code to be executed when the while condition is false, or when the for list expires...
 - Including when the loop condition is false on entry.

```
i = 1
j = 120
while i < 42:
    i *= 2
    if i > j: break
else:
    print("Loop expired: ", i)
print("Final value: ", i)
```

The else clause is not executed if
the loop exits using a break

```
Loop expired: 64
Final value: 64
```

The loop control statements 'continue' and 'break' may be familiar from other languages such as C, C++, C#, Java, PHP, awk, ksh, and Bash. Their use is often frowned upon, and it is true that redesigning conditionals can often make them unnecessary. However, this can make code more difficult to follow, so our guideline is to use them when they make the code easier to follow.

The 'do nothing' pass statement is unusual in languages (COBOL has NEXT SENTENCE) and is a consequence of Python's use of white-space to denote membership. Its use is not confined to loops - pass may be used in if statements, and usefully in an exception handler where we just want to ignore the exception.

The else: clause in a loop is also unusual. Statements that are part of this clause are executed when the loop exits, except for a break. In the example, the break is not executed because i will never exceed j, but if j was lower than 42, for example 12, then the break would be executed and the else: clause would not.

QA For loops

To iterate through a sequence

- Often a list or tuple.
- Loop variable holds a copy of each element in turn.
- Membership is by indentation.

```
for variable in object:  
    loop body
```

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

```
import sys  
for arg in sys.argv:  
    print("Cmd line argument:", arg)
```

sys.argv is a list of
the command-line
arguments

```
C:\Python>for.py Monday Tuesday Wednesday  
Cmd line argument: C:\Python\for.py  
Cmd line argument: Monday  
Cmd line argument: Tuesday  
Cmd line argument: Wednesday
```

Accessing a list or tuple sequentially (iterating) is a common enough requirement, and many languages have a for construct for this purpose. A difference with Python is that the iteration might not be done using a simple integer count, the class may have its own iterator (implemented in the class through `__iter__`). The for loop is preferable to using your own iterator and counting each element yourself - the class is much better at that sort of thing.

The loop variable (arg in the example) holds a copy of each element, so altering the variable value will not alter the list (see later for a solution).

QA For loops

To iterate through files

- In the filesystem using the glob module.
- Globbing (global matching) derived from Unix shells.
- Matches file and directory names using wildcards.
- os.path.getsize() returns the size of a file.

```
import glob
import os

home_dir = "\user\john\"

for file in glob.glob(home_dir, "*.txt"):
    size = os.path.getsize(file)
    print(file, size, "bytes")
```

spam.txt 0 bytes
bacon.txt 192 bytes
eggs.txt 56 bytes
tea.txt 200 bytes

Iterator for loops can be combined with iterable objects, which is an object that contains multiple values. The glob.glob() function returns a list (iterable object) of files that match the given pattern. The loop can then iterate through the list of filenames (and directories) returned.

In this example we also use the os.path.getsize() function to get the size of each file.

One note to consider is that the glob() function returns a list which could be large if there is many files matched. An alternative would be to use the os.scandir() function that returns an iterator object (it is a generator function – more of that in a later chapter) and will yield one pathname at a time.

Q\enumerate

Use in loops over any sequence

- Returns a two-item tuple, which contains a count and the item at that position in the sequence.

```
for idx, arg in enumerate(sys.argv):  
    print('index:', idx, 'argument:', arg)
```

Or other object type which supports iteration

- For example, open will open a file and return an iterator.
- Enumerate() also takes an optional *start* parameter.

```
for nr, line in enumerate(open('brian.txt'), start=1):  
    print(nr, line, end="")
```

line numbers
start from 1,
sequences
start at 0

```
1 Some things in life are bad  
2 They can really make you mad  
3 Other things just make you swear and curse.
```

The enumerate() function (added at Python 2.3) enables us to obtain the current position in a sequence, as well as the data item. This function can be used on any sequence: a list, tuple, string, or bytearray - or any object that supports iteration.

Two items are returned from enumerate, the sequential number (starting from zero) and the data item at that position. We can also specify a different start number (introduced at Python 2.6).

On the slide, we show a less obvious use of enumerate, from a file open - we shall be describing file IO in more detail later.

QA Counting 'for' loops

Using the builtin `range()` function:

```
range([start], stop[, step])
```

```
for i in range(0, len(some_list)):
    if some_list[i] > 42: some_list[i] += 1
```

- But this maintains its own iterator:

```
for i in range(0, len(some_list)):
    print(some_list[i])
```

- Use a system generated one instead:

```
for num in some_list:
    print(num)
```

- But an **index** is needed to alter the sequence:

```
for idx, num in enumerate(some_list):
    if num > 42: some_list[idx] += 1
```

The built-in `range()` (previously called `xrange()` in Python2) is often used to produce a list of values for counting. All parameters must be integers, but they can be positive or negative. Notice that the value of the `stop` parameter is never reached.

Counting loops are popular in most primitive languages for iterating through lists, but often they are not required in Python - it is easier and faster to use a system generated iterator than to maintain your own.

We mentioned earlier that the loop variable only holds a copy of the item in the loop, so altering it will not alter the sequence. We need an index to be able to do that and `enumerate()` comes to the rescue. To be fair, `enumerate()` cannot do everything - it does not have `stop` or `step` parameters.

Q A Zipping through multiple related lists

The `zip` built-in returns an iterator of tuples:

- Can convert to a list using `list()`.
- Can consume a lot of memory.
- Useful for stepping through parallel lists.

```
farms    = ['Home Farm', 'Muckworthy',
           'Scales End', 'Brown Rigg']
squirls = [42, 12, 2, 0]
rabbits = [395, 68, 57, 32]
moles   = [12, 8, 0, 29]

for f, s, r, m in zip(farms, squirls, rabbits, moles):
    print ('Total for', f, ':', s + r + m)
```

*A squirl is a truncated squirrel

```
Total for Home Farm : 449
Total for Muckworthy : 88
Total for Scales End : 59
Total for Brown Rigg : 61
```

The `zip()` built-in function returns an iterator of tuples, and the most common use is shown. When used with a `for` loop and a tuple of loop variables, each loop variable is set to an item from the corresponding tuple (or list). It avoids having to create our own iterator, as in the classic C-style `for` loop.

The function is useful in other scenarios. For example, here is a quick way of constructing a dictionary from lists of keys and values:

```
keys = ['Australia', 'Eire', 'France', 'Finland', 'UK', 'US']
vals = ['Canberra', 'Dublin', 'Paris', 'Helsinki', 'London',
       'Washington']
mydict = dict(zip(keys,vals))
```

In Python 2 `zip` returned a list of tuples, in Python 3 `zip` returns an iterator of tuples, so Python 2 `zlist = zip(keys,vals)` becomes `zlist = list(zip(keys,vals))` in Python 3.

It is worth noting that the `zip()` function has nothing to do with archiving like the similarly named tools on UNIX.

Q4 Conditional expressions

Shorthand for conditionals

`expr1 if boolean else expr2`

```
i= 42  
j = 3  
if i > j:  
    print("i gt j")  
else:  
    print("i lt j")
```

```
print("i gt j") if i > j else print ("i lt j")  
print("i gt j" if i > j else "i lt j")
```

These 'if' statements all do the same thing

- No : and elif not allowed

```
-1 if a < b else (+1 if a > b else 0)
```

- Beware of precedence and readability

```
a = 54  
answer = a + 5 if a < 42 else 0  
answer = a + (5 if a < 42 else 0)
```



20

Conditional expressions replace the ternary conditional (?:) in C-style languages.

The extra parentheses are required in the last example because otherwise if would add 5 to a only if $a < 42$, otherwise it would set answer to zero.

You may think that conditional expressions contravene Python's clean style, however they are useful for certain more advanced statements, such as lambda functions.

QA Unconditional closedown

`os._exit(integer_expression)`

- Cannot be trapped.
- Returns *integer_expression* to the caller (usually the shell).

`os.abort()`

- Raises a SIGABRT signal (trappable on UNIX).
- Causes a core dump on UNIX, an exit 3 on Windows.

`sys.exit(expression)`

- Raises a SystemExit exception which can be trapped.
- Returns expression to the caller (usually the shell) if it is an integer.
- Prints to other objects to stderr and returns 1 to caller.

```
sys.exit ("Goodbye")
```



The `sys.exit` method will shut down the current process, even if called from a function (covered later). The argument to exit is returned to the caller, which is often a shell program, but could be another application. Many environments (e.g., UNIX) only support a single byte for the return code, so do not return numbers outside the range 0 - 255. For portability reasons, it is wise to stay within that range, even on Windows. By convention, zero is success. Some standard exit codes are defined in the UNIX version of the `os` module (prefixed `os.EX_`), but these are not universally adopted.

The main advantage of `sys.exit`, over the more primitive `os._exit`, is that it can be trapped by exception handling. It is recommended that `os._exit` is only used in special circumstances, such as immediately after a fork (UNIX specific way of creating a new process).

There appears to be a built-in called `exit` which has the same behaviour as `sys.exit`, but appearances can be deceptive. In fact, `exit` is not a built-in function but a site.Quitter callable object which raises a `SystemExit` exception. In early versions of Python (prior to

2.4), the difference was significant, but not anymore. The site module is usually automatically loaded on start-up but can be suppressed with the -S command-line option, in which case exit will not work.

If exit or sys.exit are not used, then the Python program will return zero.

Python has an atexit module which enables one or more user-written functions to be run on exit to the program (early versions of Python also had exitfunc, which is now deprecated in favour of atexit). These are not run by os._exit.

QA Unconditional flow control (2)

But what about `exit()` and `quit()`

- At start-up, the `site` module is automatically loaded.
- Unless the `-S` command-line option is given.
- Several objects are created, including `exit` and `quit`.

When called they:

- output a message:

```
>>> exit
Use exit() or Ctrl-Z plus Return to exit
>>> quit
Use quit() or Ctrl-Z plus Return to exit
```

- raise a `SystemExit` exception and close `stdin`.
- IDLE ignores `SystemExit` but closes when `stdin` is closed.

Only use in an interpreter session, not in production code!

- Because of the side-effect of closing `stdin`.

xx

The `exit()` and `quit()` functions are actually objects of `site.Quitter` class, and the magic behaviour is obtained by the class implementing the special methods `__call__` and `__repr__` (see later).

Other objects loaded by the `site` module include `copyright`, `license`, and `credits`, these are objects of class `site._Printer`.

Quote from the Python documentation: 'They are useful for the interactive interpreter shell and should not be used in programs.'

Quite apart from the fact that the `site` module might not be loaded, the side effect of closing standard input could upset programs that trap the `SystemExit` exception.

QA

SUMMARY



Python has the usual Boolean and logical operators

- Be careful of types.

Basic flow control statements :

if condition:

indented statements

while condition:

indented statements

for target in object:

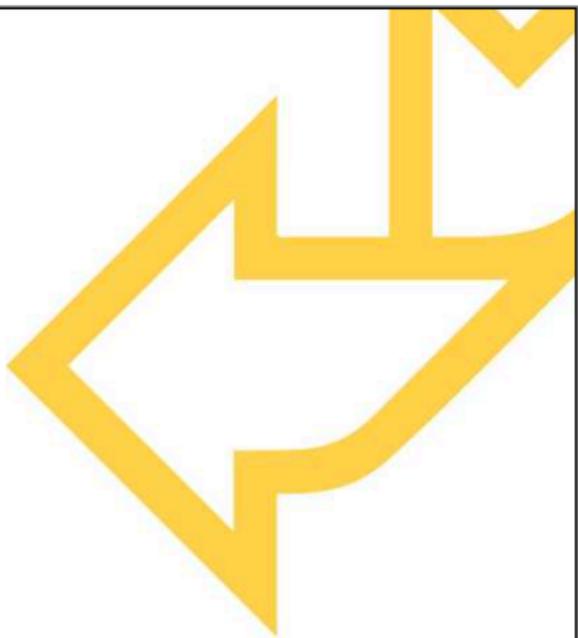
indented statements

Terminate a process using `sys.exit()`.



Python 3 Programming

String handling



QA

STRING HANDLING



Contents

- Python 3 strings
- The print function
- Cooking strings
- String concatenation
- "Quotes"
- String methods
- String tests
- String formatting
- Literal string interpolation
- Slicing a string
- String methods - split and join

Summary

- String formatting - old style

2

Handling text is important to most Python programs and is a fundamental object type in Python. They went through an import change between Python 2 and 3...

QA Python 3 strings

Strings in Python 3 are Unicode

- Multi-byte characters.
- `\unnnn` for a two-byte Unicode character.
- `\unnnnnnnn` for a four-byte Unicode character.
- `\N{name}` for a named Unicode character.

```
>>> euro="\u20ac"
>>> euro
'e'
>>> euro="\N{euro sign}"
>>> print(euro)
e
```

For low-level interfaces we have `bytes()` and `bytearray()`

```
chars_as_bytes = b"single-byte string"
```

- Conversion between strings and bytes:
`string.encode() ←→ bytes.decode()`

String types went through a major revision at Python 3, they are now Unicode strings, which are usually two bytes for each character. In 3.3 strings are more flexible and are stored internally in as few bytes as possible. Therefore, ASCII characters only use one byte each, despite being part of a unicode object (this optimisation reduces memory usage significantly).

The default encoding used is UTF-8, but this can be altered by a pragma comment at the start of the script, for example:

```
# -*- coding: latin-1 -*-
```

See the Python help text for a list of standard encodings - it is a long list! The current encoding can be obtained from
`sys.getdefaultencoding()`.

The old Python 2 `u"..."` prefix was not supported in 3.0 to 3.2, but did reappear at 3.3. It has no effect on Python 3 and is only there for backward compatibility.

You will see **bytes** and **bytearray** mentioned in the

documentation, and we will briefly see them later in the course. They map to the old single-byte character sets and are mostly used for interfacing with low-level primitives written in C, and for compatibility with Python 2.

You may find other, specialised, string types. For example, the GUI package PyQt has a `QString` type.

Q^ The print function

One of the most commonly used functions:

- Used for displaying a comma separated list of objects.
- Objects are *stringified*.

```
print(object1, object2, ... )
```

Has several named arguments, specified in any order

- end**= characters to be appended, default is '\n' (newline).
- file**= file object to be written to, default is sys.stdout.
- sep**= separator used between list items, default is a space.
- flush**=to flush or not to flush (Boolean), default is False (3.3).

```
print("The answer is", 42, "always", sep=':', end='')
```

```
print("(I think)")
```

```
The answer is: 42: always(I think)
```

4

The built-in **print()** function is one of the most used functions, and we have seen it before. The **print()** function was introduced at Python 2.6 and replaced the old **print** statement at Python 3.0 (so, 2.6 and 2.7 has both). The named arguments are new, as are the parentheses which are required now that **print** is a function.

The **flush** argument was added at Python 3.3.

When we say that objects are *stringified*, we mean that they are converted to strings into the same format as the **str** built-in function. As we shall see later, this can be overridden by a class method to stringify an object.

We shall see how to use **print** with file objects in a later chapter.

Q^ Escaping a character

Adds a meaning to a normal character

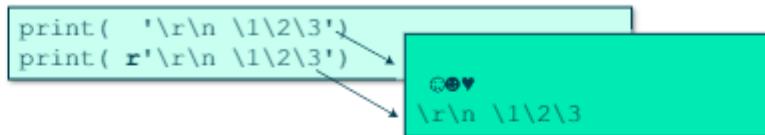
- \n becomes a new-line
- \t becomes a tab
- and so on

Removes a meaning from a special character

- \\ removes the special meaning of \
- \' removes the special meaning of '
- \" removes the special meaning of "

Raw strings do not treat \ as a special character

```
print( '\r\n \1\2\3')
print( r'\r\n \1\2\3')
```



The diagram illustrates the behavior of raw strings. On the left, two print statements are shown: one with a regular string and one with a raw string (prefixed by 'r'). Arrows point from each print statement to its corresponding output on the right. The regular string's output is '\r\n \1\2\3', where the backslash is treated as a literal character. The raw string's output is '\r\n \1\2\3', where the backslash is not treated as a special character.

Escape codes:

\newline	newline is ignored
\\"	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	Bell (BEL)
\b	Backspace (BS)
\f	Formfeed (FF)
\n	Linefeed (LF)
\N{name}	Character named <i>name</i> in the Unicode database
\r	Carriage Return (CR)
\t	Horizontal Tab (TAB)
\uxxxx	Character with 16-bit hex value xxxx
\Uxxxxxxxxx	Character with 32-bit hex valuexxxxxxxx
\v	Vertical Tab (VT)
\ooo	Character with octal value ooo(3,5)
\0	Null
\hh	Character with hex value hh

Triple quoted strings """ are discussed later.
Raw strings are particularly useful when we do not want these translations. For example when using regular expressions \1, \2, etc. are used as back-references (see later) and if you do not use raw strings these will be translated as \x01, \x02, etc. The last character inside a raw string may not be a \.

QA String concatenation

- Adjacent literals are concatenated.

```
>>> name = 'fred' 'bloggs'  
>>> name  
'fredbloggs'  
>>> name = 'fred' \  
... 'bloggs'
```

line continuation

- But that does not work with variables.
- Use the overloaded + operator instead.

```
>>> name = first + 'bloggs'
```

- But remember that strings are **immutable**.

```
s = ""  
for item in alist:  
    s = s + str(item) + " "
```



Very inefficient
code

Use join() str method instead

Much like *awk*, string literals may be 'joined' merely by placing them next to each other - the intervening white-space is optional. This is useful when specifying long strings that go over one line, but we must 'escape' the new-line first. This removes the special meaning of a new-line, which normally terminates the statement.

Unlike *awk* we cannot use this technique with variables, an operator must be used. For this, much like C++ and Java, Python overloads the + operator for strings for concatenation.

Although string concatenation using the + operator is easy, it should not be over-used, particularly in a loop. The problem is that we think we are adding data onto the end of a string, but we are not! We are creating new string objects and copying the entire contents on each iteration. There is a better way of joining all the items in a list - a string method called `join()` which we see later.

QA "Quotes"

Single and double quotes have the same effect

- Use " when you have embedded ', and vice versa.

```
print('hello\nworld')    ⇔    print("hello\nworld")
```

With embedded quotes or new-lines, use triple quotes

```
>>> html = """  
<tr>  
    <td><font color="#690000"><b>Username :</b></font></td>  
    <td><input type='textbox' name='username'></td>  
</tr>  
"""
```

```
'\n<tr>\n\n\t<td><font color="#690000"><b>Username :</b></font></td>\n\t<td><input type=\\'textbox\\\' name=\\'username\\\'></td>\n</tr>\n'
```

wrapped around

In Python, double quotes have no special meaning over single quotes - and both may contain special characters. If you have embedded single quotes (such as with SQL), then use double quotes around your string. When you embed double quotes in the data (such as with HTML), then use single quotes. If you have both or embedded white-space such as new-lines or tabs, or \ (such as Windows path names), then triple quotes (single or double) will handle them. Notice that new-lines and quotes are all 'escaped'. This makes it particularly useful when data has come from a user and may be a hacking attempt, for example SQL injection.

With triple quotes, be careful of trailing special characters, including other quotes. For example:

```
file = """C:\\QA\\Python\\PYTHB\\""""
```

does not work, since the final \\ 'escapes' the following quote.
Adding another " will unfortunately add it to the end of the data.
However, escaping the \\ does work:

```
file = """\"C:\\\\QA\\\\Python\\\\PYTHB\\\\\""""
```

QA String methods

The `string` module is now mostly replaced by methods

- Some useful string functions and methods:

String to a number	<code>int</code>	<code>int("42")</code>
Object to a string	<code>str</code>	<code>str(42)</code>
Number of characters	<code>len</code>	<code>len(name)</code>
Convert to lower case	<code>lower</code>	<code>str.lower()</code>
Replace a sub-string	<code>replace</code>	<code>str.replace('old', 'new')</code>
Remove trailing chars	<code>rstrip</code>	<code>str.rstrip()</code>
Search for a sub-string (returns the offset)	<code>find</code>	<code>str.find('cheese')</code>

- Overloaded * operator:

```
>>> 'Spam' * 4
      Spam Spam Spam Spam' ← Mandatory Monty Python reference
```

In Python's early days, string handling was done using the `string` module. Thankfully, this is no longer necessary, and we have several built-in methods. The `string` module still works but is only there for backward compatibility.

Both `str` and `repr` can convert a number to a string, generally `str` will produce prettier output. Both these functions can be implemented in a user written class, and their expected result is rather different. The `str` function is supposed to return a human-readable form of the object. In contrast, `repr` is supposed to return a Python readable form, or *representation*, specifically for the built-in function `eval` (which compiles code at runtime and is generally pronounced *evil*). So, use `str` unless you are doing particularly eval things.

Note that in early versions of Python, `back-ticks` could be used instead of `repr`, but this was removed at Python 3.0.

The `replace` and `find` methods often overlap in functionality with regular expressions (discussed later). Generally, these methods are

more efficient than REs (Regular Expressions), and easier to code.

There are other string methods, around 40 in all, some are discussed later.

QA String tests

Remember the `in` operator

```
if substr in string:
```

Testing a string type can often be done with a method:

- Regular Expressions can also be used, but can be slow.

```
count  
endswith  
isalnum  
isalpha  
isdigit  
islower  
isspace  
istitle  
isupper  
startswith
```

```
text = 'hello world'  
print(text.count('o'))  
  
if text.startswith('hell'):  
    print("It's hell in there")  
  
if text.isalpha():  
    print('string is all alpha')  
  
text = '\t\r\n'  
if text.isspace():  
    print('string is whitespace')
```

```
2  
It's hell in there  
string is whitespace
```

Several methods are available with strings to test their type. Many return a Boolean (True or False) but some, like `count`, do not. Most of these methods can be coded in other ways and using a Regular Expression (RE) is the most obvious. However, REs are often slow, and rarely easier to read. Here is the RE code for the example shown, details later:

```
import re  
text = 'hello world'  
  
print(len(re.findall(r"(o)", text)))  
print(len(re.findall(r"(o)", text[5:])))  
  
if re.match(r"^hell", text):  
    print("It's hell in there")  
  
if re.match(r"^[A-Za-z]+$", text):  
    print("string is all alpha")  
  
text = '\t\r\n'  
if re.match(r"^\s+$", text):
```

```
print("text is whitespace")
```

QA String formatting

Call the format method on a string

```
"string".format(field_values)
```

- *string* - contains text and the format of values to be substituted in.
- *field_values* - the parameters to be evaluated in string.

String format specifications are all optional, and of the form:

```
{ position : fill align sign # 0 width . precision type }
```

```
"text{0:5.3f} text{1:3.d} text{2}" .format(var1, var2, var3)
```

positions are optional if sequential (3.1)

We can also specify index numbers or key names

```
"text{0[index]} text{1[key]}" .format(list, dictionary)
```

Other languages have printf and sprintf, Python has string formatting.

Python 3 introduced this new formatting style, for the old style, see slides after the summary. This style is also available in Python 2, from version 2.6.

The fields in the format specifications have the following meaning:

fill	Any character except {
align < left, > right, ^ centred, = pad sign	
sign	+ force sign, " " only force - (minus)
#	prefix integers with 0b, 0o, or 0x
0	Pad numbers with zero
width	Minimum field width
.precision	Maximum field width (strings), or number of decimal places
type b, c, d, n, o, x, X - integer formats e, E, f, g, G - floating point formats	

These are all optional, the default type being a string. Taking the first example:

{0:5.3f}

0 is the position, the first variable passed to the format method (var1)

From Python 3.1 they are automatically incremented if omitted

5 is the minimum field width, including the decimal point
3 is the precision, meaning 3 characters after the decimal point.

Field names (also known as nested arguments) may be used for substitutions, e.g.:

```
var1 = 42
var2 = 'hello'
print("{name} {value}".format(value=var1,
name=var2))
'hello 42'
```

Q^ String formatting example

Common conversion specifiers:

- (d) Treats the argument as an integer number.
- (s) Treats the argument as a string.
- (f) Treats the argument as a float (and rounds).

```
# Distance of planets to Sun in gigametres(Gm) .  
planets = {'Mercury': 57.91,  
           'Venus': 108.2,  
           'Earth': 149.597870,  
           'Mars': 227.94  
}  
  
for i, key in enumerate(planets.keys(), 1):  
    print("{:2d} {:<10s} {:06.2f} Gm".format(i, key, planets[key]))
```

1	Earth	149.60	Gm
2	Mercury	057.91	Gm
3	Mars	227.94	Gm
4	Venus	108.20	Gm



The format specifiers shown on the slides are type specifiers - they specify how the argument should be treated.

The example format string contains the following specifiers:

- {2d} Right-justified, at least two digits, space padding
- {<10s} Left-justified, at least 10-character string, space padding
- {06.2f} Right-justified, at least 6 characters, zero padding. One of the six characters is the decimal point, and the argument is rounded to two digits after the decimal point

The second parameter to enumerate (1) is the start of the enumeration sequence (default is zero).

Here is the example using field names:

```
for i, key in enumerate(planets.keys(), 1):  
    print("{pos:2d} {planet:<10s} {distance:06.2f} Gm".  
          format(pos=i,  
                  planet=key,distance=planets[key]))
```

By the way, Gm is the symbol for gigametre, the little used metric unit of 1000 kilometres. The example shows the mean distance from the Sun for these planets.

QA Other string formatting aids

Often more efficient and easier

- `string.capitalize()`
- `string.lower() / string.upper()`
- `string.center()`
- `string.ljust()`
- `string.rjust()`
- `string.zfill()`

```
text = 'hello'  
print(text.capitalize())  
print(text.upper())  
print('<'+text.center(12)+ '>')  
print('<'+text.ljust(12)+ '>')  
print('<'+text.rjust(12)+ '>')  
print('<'+text.zfill(12)+ '>')
```

```
Hello  
HELLO  
< hello >  
<hello>  
< hello>  
<0000000hello>
```

Often the string format strings are not required, and the formatting string methods may be enough. We feel the examples are self-explanatory.

There are other string methods, for example `string.title()`, which changes the string to "title case" where the first character of each word is in upper case.

QA Literal string interpolation

Python expressions may be embedded inside a text string.

- Available from Python 3.6

Special string literals are used, known as *f-strings*.

- Embed a python expression inside braces.

```
names = ['Tom', 'Harry', 'Jane', 'Mary']
s = f"The third member is {names[3]}"
```

String formats may be embedded

- Syntax is {value:(width).(precision)}
- This is the planets example rewritten to use an f-string.

```
for i, key in enumerate(planets.keys(), 1):
    print(f"{i:2d} {key:<10s} {planets[key]:06.2f} Gm")
```

Python has had interpolation of sorts previously, in `string.Template`, but it was unwieldy and not used very often. A new system, specified in PEP 498 and introduced with Python 3.6, is very similar in appearance to that used by Ruby.

Single or double quotes may be used, even triple quotes. Basically, we just put the Python expression, commonly a variable, inside braces. A special syntax is used for format strings.

Q& Literal string interpolation (2)

Not just variable values may be represented:

```
names = ['Tom', 'Harry', 'Jane', 'Mary']
suffix = ['st', 'nd', 'rd', 'th']
n = 1
s = f"{str(n+1) + suffix[n]} of \
{len(names)} is {names[n]}"
```

2nd of 4 is Harry

Can also be combined with raw strings:

```
drive = 'C:'
dir = 'Windows'
path = fr"{drive}\{dir}"
```

f-strings supports only Unicode:

- Byte objects do not support f-strings.

4

Function calls, arithmetic, any valid python expression may be put inside the braces. However, we suggest that you Keep It Simple – complex expressions inside a string would be difficult to debug and read, so it is probably preferable to restrict yourself to simple expressions.

A raw string may be combined with an f-string, either `fr` or `rf` prefix may be used. An f-string cannot be combined with the `u'` (Unicode) prefix, since f-strings can only be Unicode. The specification rules out the possibility of f-string interpolation being back-ported to Python 2.

The `fr' '` example is there to demonstrate the syntax, the correct way to join path components is to use `os.path.join()`. Raw and f-strings are most commonly seen with regular expressions, which we will look at later in the course.

QA Slicing a string

A Python string is an immutable sequence type.

- Slicing is the same for all sequence types.

Slice by start and end+1 position.

- Counting from zero on lhs, from -1 on rhs.

```
#      01234567890123456789012345678901234
text = "Remarkable bird, the Norwegian Blue"
print(text[11:14]) ← [start: end+1]
bir
print(text[-7:-1])
an Blu
```

- Start and end positions may be defaulted.

```
print(text[:14])
Remarkable bir
print(text[-7:])
an Blue
```



Beautiful plumage

Sequential composite objects like strings, tuples, and lists may be sliced. Often only one item is required, in which case the syntax uses the familiar square brackets with the index inside. Items are indexed from zero on the left, or -1 (with a negative count) from the right.,

To slice a range of elements, we specify the start index, a colon, then end index plus one. That is, the slice is taken up to, but not including, the second index position. If the start index is not given, the default is zero (first element). Defaulting the second index slices to the end of the object. For example, `string[:-1]` will remove the last character from a string.

Python 2.3 added support for a third index, a step. For example:

```
#      0123456789012345
text = "Now that's what I call a dead parrot."
print (text[5 : 15 : 2])
Gives: htswa
```

This has printed every other character, from position 5 through to

15.

Here is another example:

```
print(name[::-1].upper())
```

what do you think that does? (Try it)

QA String methods - split and join

String to a list - split

- `string.split([separator[, max_splits]])`
- If `separator` is omitted, split on one or more white-space.
- If `max_splits` is omitted, split the whole string.
- `string.splitlines()` is useful on lines from files.

Sequence to a string - join

- `separator.join(sequence)`
- `sequence` can be a string, list or a tuple.

```
line = 'root::0:0:superuser:/root:/bin/sh'
elems = line.split(':')

elems[0] = 'avatar'
elems[4] = 'The super-user (zero)'
line = ':'.join(elems)
print(line)
```

```
avatar::0:0:The super-user (zero) :/root:/bin/sh
```

The `split` function is commonly used to extract fields from records read from a file. If you want to split around line-endings, then use `splitlines()` instead. Why might you want to do that? Because it is common to read a small file into a single string and then split it, so that each line goes into an element of a list.

There are other versions of `split` available: in the `re` module and in the `os.path` module. They have different functionality to the string `split` function.

The `join` function is a method called on the separator string, which is rather unusual and takes some getting used to. When called on an empty string, it concatenates the characters in the list into one.

We mentioned `join` earlier. If you are concatenating a string in a loop, it is actually faster to append each item to the end of the list instead, then join them all together after the loop. With a small number of iterations, you will not notice any difference, but with a large number there is a measurable improvement in performance.

QA

SUMMARY



- Python 3 strings are Unicode.
- Python variables are not embedded inside quotes.
 - However, escape characters like \r\n\t can be.
- No difference between ' and “.
 - Use three quotes for multi-line text.
- Several methods available on a string:
 - Many for conversions.
 - Formatting uses the format method.
 - Split a string with split().
 - Join items in a list with join().
- Strings can be sliced[start:end+1].
 - As can other sequences.

QA

MORE STRING FORMATTING EXAMPLES



```
flt = 22/7
print("Float: {0:11.8f}, sci: {0:e}".format(flt))
        Float: 3.14285714, sci: 3.142857e+00
```

```
first = 'Gengis'
second = 'Khan'
print("Name: {:<20s} {:<10s}".format(first, second))
```

Name: Gengis Khan

```
file = sys.argv[0]
perms = '{0(:o)}'.format((os.stat(file).st_mode) & 0o7777)
```

Octal number
0640

```
fred = '{:#x}'.format(3735928559)
```

0xdeadbeef

format is particularly useful for displaying floating point numbers, in a variety of formats. The first example shows a number displayed with a total width of 11 characters (including the decimal point), and 8 digits after the decimal point, followed by engineering notation. Notice also that only one value is supplied, this is because both format specifications use position zero. In the other examples, we are not specifying the position, so they are automatically numbered for us (introduced at 3.1).

Text columns may be left or right aligned, the second example shows two fields left aligned and padded with spaces.

Converting to hexadecimal is useful (use **o** for octal), and we also show an example of returning the formatted string instead of printing it.

Values normally stored in octal, like UNIX file permissions, are displayed or manipulated more easily using **format**.

QA

PYTHON 2 STRING FORMATTING



Older method for formatting strings

- Like sprintf in some other languages.

`"format_string" % (argument_list)`

`format_string`

- Contains text and format specifiers, prefixed %.
- Describe format of the plugged-in value.

%s	string	%o	octal
%c	character	%x	lowercase hex
%d	decimal	%X	uppercase hex
%i	integer	%%	literal %
%u	unsigned int		
		%e, %E, %f, %g, %G	- alternative floating point formats

`argument_list`

- Contains text or variables to be plugged-in.

19

This style of string formatting is still supported in Python 3 but considered deprecated. It might be removed from the language eventually, so should not be used for new applications.

Many languages have the `printf` and `scanf` family of functions (originally from C), but Python 2 overloaded the `%` operator. If you are mixing text variables and escape sequences, then this provides a lot more flexibility than `print`.

The general form of a format specifier is:

`%[flag][width].[precision]letter`

where: **[flag]** specifies padding a signed options

[width] specifies how wide the field is

[.precision] specifies decimal digits for floating point numbers

and **letter** indicates the data type. To print a literal '%', use '%%'.

Only the common format specifiers are shown, for a full list see the online help text. Floating point numbers are rounded, but the rounding algorithm does not always produce what you might expect. If a particular rounding method is important, with money amounts, for example, it is best to implement it yourself rather than relying on %.

QA

PYTHON 2 STRING FORMATTING EXAMPLES



```
flt = 22/7
print("Float format: %11.8f, scientific: %e" % (flt, flt))
Float format: 3.14285714, scientific: 3.142857e+00
```

```
first = 'Gengis'
second = 'Khan'
print("Name: %-20s %-10s" % (first, second))
```

Name: Gengis Khan

```
file = sys.argv[0]
perms = '0%lo' % (os.stat(file).st_mode & 0o7777)
```

Octal number

0640

```
fred = '%x' % 3735928559
```

deadbeef

20

% is particularly useful for displaying floating point numbers, in a variety of formats. The first example shows a number displayed with a total width of 11 characters (including the decimal point), and 8 digits after the decimal point, followed by engineering notation.

'Columns' may also be left or right aligned, the second example shows two fields left aligned and padded with spaces.

Converting to hexadecimal is useful (use %o for octal), and we also show an example of returning the formatted string instead of printing it.

Values normally stored in octal, like UNIX file permissions, are displayed or manipulated more easily using %.

QA String formatting and bytes objects

String formatting produces strings, not bytes.

- In Python 2, they are the same.

Bytes objects do not have a `.format()` method.

- So the only formatting available is using %.

The following are additional formats for bytes objects:

- %a – create a bytes object by encoding.
- %b – create a bytes object using class specified method.
- %c – a single byte.

```
print(b'%a' % 42)
print(b'%b' % b"hello")
print(b'%c' % 42)
```

```
b'42'
b'hello'
*
```

42 is the ASCII ordinal number of the * character

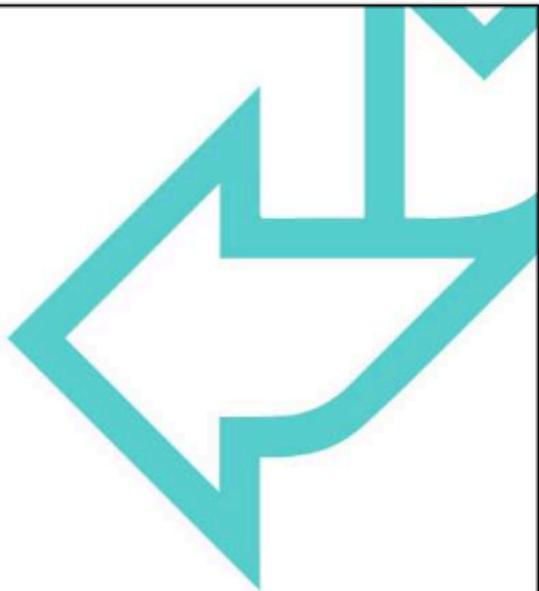
Omitting the b' on the left side of the format string will produce an str object rather than a bytes object. The exception is %b which is not supported inside a str.

See *printf-style Bytes Formatting* in the online documentation.



Python 3 Programming

Collections



QA

COLLECTIONS



Contents

- Python types – reminder
- Generic built-in functions
- Useful tuple operations
- Python lists
- Tuple and list slicing
- Adding items to a list
- Removing items from a list
- Sorting
- List methods
- Sets
- Set operators
- Python dictionaries
- Dictionary methods
- View objects

2

This chapter discusses more basic building blocks of a Python program - its container variable types - collectively known as collections.

QA Python types - reminder

Built in sequence types:

- Strings (str)
`'Norwegian Blue', "Mr. Khan's bike"`
- Lists (list)
`['Cheddar', 'Camembert', 'Brie', 'Stilton']`
- Tuples (tuple)
`(47, 'Spam', 'Major', 683, 'Ovine Aviation')`
- We also have bytearray (read/write) and bytes (read only) used for binary data.

Not all collections are sequences

- A set is an *unordered* collection of *unique* objects.
- Dictionaries are a special form of set.
`{'Totnes': 'Barber', 'BritishColumbia': 'Lumberjack'}`

Strings, Lists, and Tuples are ordered collections of objects (also known as sequences).

The types bytearray and bytes are used for raw binary data and were introduced in Python 2.6. They are similar to strings and can be accessed using the same methods as strings and lists. They may be indexed using []. They hold 8-bit signed integers in the range 0-255.

Dictionaries are collections of objects accessed by key and are similar to associative arrays in awk and PHP, and hashes in Perl and Ruby.

Sets were introduced into Python 2.4.

There is also a collections module in the Python Standard Library, this includes alternative base classes for the default containers, as well as some other types, such as deque, and in Python 3.1 the OrderedDict type was added.

In Python 3.6, dictionaries in CPython are ordered in Insertion Order, and from Python 3.7 this is now a language feature.

QA Generic built-in functions

Most *iterables* support the `len`, `min`, `max`, and `sum` built-in functions.

- `len` Number of elements
- `min` Minimum value
- `max` Maximum value
- `sum` Numeric summation (not string or byte objects)

Dictionaries implement `min`, `max`, and `sum` on keys.

The `sum` built-in will raise a `TypeError` if the item is not a number.

```
myn = [45, 66, 12, 3, 99, 3.142, 42]
print("min:", min(myn), "max:", max(myn))
print("sum:", sum(myn))

myd = {'fred':3, 'jim':8, 'dave':42}
print("min:", min(myd), "max:", max(myd))
```

`min: 4 max: 99
sum: 270.142
min: dave max jim`

The example code produces the following output:

```
min: 3 max: 99
sum: 270.142
min: dave max: jim
```

Range objects can also be included in the list of sequence objects.

sum() supports an optional second parameter, `start`, which gives the initial value of the sum (defaults to zero).

The **len()** function returns the number of characters when used with a string, and the number of bytes when used with a bytes object. For example:

```
mys = chr(0x20ac)
print(mys, len(mys))
myb = mys.encode()
print(myb, len(myb))
```

Gives:

```
€ 1
b'\xe2\x82\xac' 3
```

QA Useful tuple operations

Swap references:

```
a, b = b, a
```

Set values from a numeric range:

```
Gouda, Edam, Caithness = range(3)
```

```
0 1 2
```

Repeat values:

```
mytuple = 'a', 'b', 'c'  
another = mytuple * 4
```

```
('a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c')
```

Be careful of single values and the trailing comma:

```
thing = ('Hello')  
print(type(thing))  
thing = ('Hello',)  
print(type(thing))
```

```
<class 'str'>
```

```
<class 'tuple'>
```

Tuples *elements* can be assigned as long as they are named variables. The example shown, swaps two variables (a and b), but many more variables could be involved. This is just a special case of assigning one tuple to another. Notice we are not altering a tuple; we are altering the values of the variables within. There must be the same number of variables on the left as there are values on the right. The parentheses are optional, so the example could also be written as:

```
(a, b) = (b, a)
```

The second example of a tuple operation sets three variables and gives them values from a range. Gouda will have the value 0, Edam will be 1, and Caithness 2.

In Python 2, the **range()** function created a complete temporary list of integers in memory, so the **xrange()** function was used instead. In Python 3 **range()** returns a *lazy list* (as **xrange()** did), which only generates the next value when it is needed.

Tuples (and strings and lists) may be repeated using the *****

operator. In this case, the tuple `another` will have the values of `mytuple` repeated four times.

It is tempting to use parentheses around a single item, but this will not produce a tuple. A trailing comma is required.

QA Python lists

Python lists are similar to arrays in other languages.*

- Can contain objects of any type.
- Multi-dimensional lists are just lists containing references to other lists.

Lists can be created using [] or the built-in list() function.

Access list elements using [] or by method calls.

- Indexes on the left start at zero.
- Indexes on the right start at -1.
- Multiply operator * can also be applied to a list.

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg']
print(cheese[1])
cheese[-1] = 'Red Leicester'
print(cheese)
```

Stilton
['Cheddar', 'Stilton', 'Red Leicester']

Lists are objects containing a sequential collection of other objects, commonly called *elements*. Elements may be accessed by a position (counting from zero) specified within [] which is probably familiar from other languages.

Lists are *Mutable*, that is they may be changed, so they are similar to arrays in some languages. They are dynamic in that they may be extended or shrunk. New items may be added anywhere with the list, and also removed (discussed in later slides).

List may be concatenated (joined together) using the + operator.

*Python lists are similar to arrays in other languages.

Don't carry this analogy too far. Python lists are not like arrays in C. That's a good thing in general programming but carries a performance overhead – C arrays are very fast. So, Python has a module called **array** in the standard library which provides objects that are very similar to C arrays.

QA Tuple and list slicing

Slicing can return multiple objects from Tuple:

- Indexed by [start, end] but does not include end-index.
- Counting from zero on LHS, from -1 on RHS.

```
mytuple=('eggs', 'bacon', 'spam', 'tea', 'beans')
print(mytuple[2:4])
('spam', 'tea')  
print(mytuple[-4])
bacon  
mylist = list(mytuple)
print(mylist[1:])
['bacon', 'spam', 'tea', 'beans']
print(mylist[:2])
['eggs', 'bacon']
```

- List elements may be removed using del.

```
cheese = ['Cheddar', 'Camembert', 'Brie', 'Stilton']
del cheese[1:3]
print(cheese)
```

```
['Cheddar', 'Stilton']
```

7

Sequential composite objects like strings, tuples, and lists may be sliced. Often only one item is required, in which case the syntax uses the familiar square brackets with the index inside. Items are indexed from zero on the left, or -1 (with a negative count) from the right.

To slice a range of elements we specify the start index, a colon, then end index plus one. That is, the slice is taken up to, but not including, the second index position.

If the start index is not given, then the default is zero (first element). For example, string[:-1] will give the characters up to, but not including, the last character in a string, effectively deleting it.

Defaulting the second index slices to the end of the object. Python strings may be sliced in a similar way.

The del statement can delete a slice, a comma separated list of elements, or the whole list. Notice that del is a statement, not a built-in function, so parentheses are not required around the name

being deleted.

Q\Extended iterable unpacking

Python 3 allows unpacking to a wildcard.

- Only allowed on the left-side of an assignment.

```
mytuple = 'eggs', 'bacon', 'spam', 'tea'  
x, y, z = mytuple
```

ValueError: too many values to unpack

```
mytuple = 'eggs', 'bacon', 'spam', 'tea'  
x, y, *z = mytuple  
print(x, y, z)
```

eggs bacon ['spam', 'tea']

```
t1 = 'cat', 'dog', 'python', 'mouse', 'camel'  
t2 = 'kelp', 'crab', 'lobster', 'fish'  
  
for a, *b, c in t1, t2:  
    print(a, b, c)
```

cat ['dog', 'python', 'mouse'] camel
kelp ['crab', 'lobster'] fish

PEP 3132 introduced a welcome simplification of the syntax when assigning to tuples - unpacking. In the past, if assigning to variables in a tuple, the number of items on the left of the assignment must be exactly equal to that on the right.

In Python 3, we can designate any variable on the left as a tuple by prefixing with an asterisk *. That will grab as many values as it can, as a list, while still populating the variables to its right (so it need not be the rightmost item). This avoids many nasty slices when we don't know the length of a tuple.

Another form of unpacking, also using a *, has been available for some time, including Python 2, in function arguments, which we shall see later.

QA Adding items to a list

On the left:

```
cheese[:0] = ['Cheshire', 'Ilchester']
```

On the right:

```
cheese += ['Oke', 'Devon Blue']
cheese.extend(['Oke', 'Devon Blue'])
```

Same effect

- Append can only be used for one item.

```
cheese.append('Oke')
```

Anywhere:

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg']
cheese.insert(2, 'Cornish Brie')
cheese[2:2] = ['Cornish Brie']
print(cheese)
```

Same effect

```
['Cheddar', 'Stilton', 'Cornish Brie', 'Cornish Yarg']
```

List may be extended in any direction from any position. In the first example, 'Cheshire' and 'Ilchester' are added to the front of the cheese list.

In the second example, we show two ways of adding items to the end of a list, using the `+=` operator and using the `extend` method. In theory, `extend` is more efficient, but it is unlikely that you would notice, or even be able to measure, a difference.

The third example shows the `append` method, which can only be used to add one item - but that is often enough.

Finally, we show two ways of inserting an item at a specific position - using the `insert` method (specifying the index position) and using a slice. Note that with `insert` we can only insert one item, but using a slice we can insert as many items as we wish.

Q& Removing items by position

Use `pop(index)`

- The index number is optional, default -1 (rightmost item).
- Returns the deleted item.

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg']
saved = cheese.pop(1)
print("Saved1:", saved, ", Result:", cheese)

saved = cheese.pop()
print("Saved2:", saved, ", Result:", cheese)
```

Saved1: Stilton , Result: ['Cheddar', 'Cornish Yarg']
Saved2: Cornish Yarg , Result: ['Cheddar']

Remember that `del` may also be used:

- Does not return the deleted item.
- May delete more than one item by using a slice.

The `pop` method removes the specified item from a list, the default being the last (rightmost) item. For example, use `cheese.pop(0)` to remove the leftmost item from the list.

An advantage of `pop` over `del` is that it returns the deleted item, on the other-hand `del` may remove more than one. Deleting from the right-hand end of the list, which `pop` does by default, is generally more efficient. Deleting from anywhere else, can mean that the internal representation of the list has to be rebuilt.

Q^ Removing list items by content

Use the `remove` method:

- Removes the leftmost item matching the value.

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg',
          'Oke', 'Devon Blue']
cheese.remove('Oke')
print(cheese)
```

['Cheddar', 'Stilton', 'Cornish Yarg', 'Devon Blue']

Raises an exception if the item is not found.

- Exceptions will be handled later...

```
cheese.remove('Brie')
```

Traceback (most recent call last):
 File "...", line 57, in <module>
 cheese.remove ('Brie')
ValueError: list.remove(x): x not in list

Sometimes we might not know the position of an item, we might want a "search and destroy" method - which is exactly what **remove** does. Note that data items in a list need not be unique and **remove** will find the leftmost occurrence of the value.

QA Sorting

sorted built-in and sort method

- sorted can sort any *iterable* (often a sequence).
- sorted returns a sorted list - regardless of the original type.
- sort sorts a list in-place.

Both have the following optional named parameters

key=sort_key	Function which takes a single argument.
reverse=True	Default is False.

```
cheese = ['Cornish Yarg', 'Oke', 'Edam', 'Stilton']
cheese.sort(key=len)
print(cheese)      ['Oke', 'Edam', 'Stilton', 'Cornish Yarg']

nums = ['1001', '34', '3', '77', '42', '9', '87']
newstr = sorted(nums)
newnum = sorted(nums, key=int)

newstr: ['1001', '3', '34', '42', '77', '87', '9']
newnum: ['3', '9', '34', '42', '77', '87', '1001']
```

The sort algorithm used by Python is the Adaptive Stable Mergesort (algorithm by Tim Peters).

sorted was introduced in Python 2.4 and returns a sorted list. **sort** alters the list in-place, it returns None.

The **key** is a single argument function to be called which returns the key value. This is often a **lambda** function- an anonymous inline function, discussed later.

The **key** function is called once for each element to be sorted and determines the actual value to be compared. Note that in the second set of examples, with the numbers, the first result is from a textual comparison and the second is numeric. If the values had not been enclosed with quotes, then the default comparison would have worked correctly.

Old versions of Python **sort** and **sorted** also had a **cmp** argument (and there was a **cmp** built-in).

The `sorted()` built-in can sort anything which is iterable, which includes sequence types like tuples and strings, but always returns a list (remember that tuples and strings are immutable).

A `sort()` method should be described for all *mutable* sequence types. In the base types (those not requiring an external module) that only includes **lists** and **bytearrays**.

QA Miscellaneous list methods

Count `list.count ('value')` Return the number of occurrences of 'value'

Index `list.index ('value')` Return index position of leftmost 'value'

Reverse `list.reverse()` Reverse a list in place

```
cheese = ['Cheddar', 'Cheshire', 'Stilton', 'Cheshire']
print(cheese.count('Cheshire'))
print(cheese.index('Cheshire'))
cheese.reverse()
print(cheese)
```

2
1
['Cheshire', 'Stilton', 'Cheshire', 'Cheddar']

A full table of list methods is shown on the next slide.

The `index` method returns the index position of the leftmost item found (counting from zero). Throws a `ValueError` exception if the item is not found.

QA

LIST METHODS



<code>list.append(item)</code>	Append <i>item</i> to the end of <i>list</i>
<code>list.clear()</code>	Remove all items from <i>list</i> (3.3)
<code>list.count(item)</code>	Return number of occurrences of <i>item</i>
<code>list.extend(items)</code>	Append <i>items</i> to the end of <i>list</i> (as <code>+=</code>)
<code>list.index(item, start, end)</code>	Return the position of <i>item</i> in the <i>list</i>
<code>list.insert(position, item)</code>	Insert <i>item</i> at <i>position</i> in <i>list</i>
<code>list.pop()</code>	Remove and return last item in <i>list</i>
<code>list.pop(position)</code>	Remove and return item at <i>position</i> in <i>list</i>
<code>list.remove(item)</code>	Remove the first <i>item</i> from the <i>list</i>
<code>list.reverse()</code>	Reverse the <i>list</i> in-place
<code>list.sort(...)</code>	Sort the <i>list</i> in-place - arguments are the same as <code>sorted()</code>

This slide is for reference. The `clear()` method was added at Python 3.3 and is not in Python 2.

QA Sets

A set is an *unordered container of object references*:

- A set is *mutable*, a frozenset is *immutable*.
- Set items are unique.

Creating a set:

- Any iterable type may be used.

```
s1 = {5, 6, 7, 8, 5}
print(s1)

s2 = set([9, 10, 11, 12, 9])
print(s2)

s3 = frozenset([9, 10, 11, 12, 9])
print(s3)
```

{8, 5, 6, 7}
{9, 10, 11, 12}
frozenset((9, 10, 11, 12))

Sets can be considered to be like lists, only being unordered. Somewhat like one half (the keys) of a dictionary. Indeed, we shall see later that the `dict.keys()` method can be treated as a set. They are useful for lookup tables, where an associated value is not needed and only membership need be tested (using `in`), and for operations between sets, like intersection.

Notice that any original order is lost.

The method `add` is used to add a single element. To add multiple elements, use `update`.

Note that the output shown is for Python 3, on Python 2.4 the set was shown inside [] with the prefix 'set'.

QA Set methods

Add using the `add` method, remove using `remove`.

```
s4 = {23, 42, 66, 123}
s5 = {56, 27, 42}
print("{:20} {:20}".format(s4, s5))

s4.remove(123)
s5.add(123)
print("{:20} {:20}".format(s4, s5))

(66, 123, 42, 23)      (56, 42, 27)
(66, 42, 23)           (56, 123, 42, 27)
```

Other set methods:

- `len` Return the number of elements in the set.
- `discard` Remove element *if present*.
- `pop` Remove and return the next element from the set.
- `clear` Remove all elements.

The method `add` is used to add a single element.

To add multiple elements, use `update`. There are three "updates", and alternative operators:

<code>update</code>	<code> =</code>	Update the set
from another		
<code>intersection_update</code> &=		Update the set, with common elements
<code>difference_update</code> -=		Remove elements found in the other

Frozensets are immutable so do not have the `add`, `update`, or `remove` methods, but they are hashable so can be used as entries in other sets.

QA Exploiting sets

How do I remove duplicates from a list?

- But we lose the original order.

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg',
          'Oke', 'Stilton', 'Cheshire']
cheese = list(set(cheese))
['Cornish Yarg', 'Cheshire', 'Cheddar', 'Stilton', 'Oke']

list() is required, otherwise 'cheese' would now refer to a set
```

How do I remove several items from a list?

- Subtract sets and convert back into a list, of course.

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg',
          'Oke', 'Stilton', 'Cheshire']
cheese = list(set(cheese) - {'Stilton', 'Oke', 'Brie'})
['Cornish Yarg', 'Cheshire', 'Cheddar']
```

Sets can be used for a number of functions, including membership using `in`. Here, we have used them to exploit their features.

Items in a set are unique, so putting a list into a set will automatically remove any duplicates - the only problem being that any original order is lost. One reason is to remove duplicates before a sort. Without sets, we would use a dictionary for this, but with some 'throw-away' value.

We can apply various set operations, so taking a list and turning it into a set allow us to apply those operations to (what was) the list.

Notice that when we remove one set from another, the items in set on the right does not have to be in the left. In the example, 'Brie' does not exist in the set built from cheese, and it is not an error to try to remove it.

QA

SET OPERATORS AND METHODS



Operator	Method	Returns a new set containing
&	s6.intersection(s7)	Each item that is in both sets
	s6.union(s7)	All items in both sets
-	s6.difference(s7)	Items in s6 not in s7
^	s6.symmetric_difference(s7)	Items that occur in one set only

```
s6 = {23, 42, 66, 123}
s7 = {123, 56, 27, 42}

print(s6 & s7)
print(s6 | s7)
print(s6 - s7)
print(s6 ^ s7)
```

{42, 123}
{66, 27, 42, 23, 56, 123}
{66, 23}
{66, 23, 56, 27}

The set method versions can take any iterable as its parameter, so there is no need to convert to a set first.

QA Python dictionaries

A dictionary is an *ordered (Python 3.6)* container of objects:

- Like sets but objects are accessed by keys.
- Constructed using {} or the built-in dict() function:

```
varname = {key1:object1, key2:object2, key3:object3,...}
```

```
varname = dict(key1=object1, key2=object2, key3=object3,...)
```

- The key is usually a text string, or anything that yields a text string.

```
varname[key] = object
```

```
mydict = {'Australia':'Canberra', 'Eire':'Dublin',
          'France':'Paris', 'Finland':'Helsinki',
          'UK':'London', 'US':'Washington'
        }
print(mydict['UK'])

country = 'Iceland'
mydict[country] = 'Reykjavik'
```

Dictionaries are like sets but have keys referencing the objects. These are similar to associative arrays, hashes, and hash tables in other languages and in general computing the keys are unordered. However, from Python 3.6 the keys are in insertion order in CPython, and from Python 3.7, it is now a language feature across all Python platforms.

They are constructed from lists of key:object pairs, inside braces (curly brackets), although you may also assign them from the dict function, for example:

```
mydict = dict(Sweden = 'Stockholm', Norway = 'Oslo')
```

This form can only be used if the keys are text strings, not if they are numbers.

The value is an object of any valid class, including a list, tuple, or dictionary. No special syntax is required to access them.

Dictionary key:value pairs are not ordered, as you would expect. A list of the keys may be extracted using the keys() method, and

values with the values() method.

You can also create dictionaries with just keys, from a list.

```
mydict = {}.fromkeys(mylist)
```

These can be used as look-up tables, or the values added later.

Dictionary keys can be any immutable type: strings, numbers, or tuples, but not mutable types, such as lists. To get a list of keys from an existing dictionary, then use the dictionary as a list:

```
keys = list(mydict)
```

Q^ Dictionary values

Objects stored can be of any type:

- Lists, tuples, other dictionaries, etc...
- Can be accessed using multiple indexes or keys in [].
- Add a new value just by assigning to it.

```
mydict = {'UK':['London', 'Wigan', 'Macclesfield', 'Bolton'],
          'US':['Miami', 'Springfield', 'New York', 'Boston']}
print(mydict['UK'][2])

homer = 1
print(mydict['US'][homer])

mydict['FR'] = ['Paris', 'Lyon', 'Bordeaux', 'Toulouse']
for country in mydict.keys():
    print(country, ': ', mydict[country])

```

FR : ['Paris', 'Lyon', 'Bordeaux', 'Toulouse']
US : ['Miami', 'Springfield', 'New York', 'Boston']
UK : ['London', 'Wigan', 'Macclesfield', 'Bolton']

20

Shown is a simple dictionary containing just two keys ('UK' and 'US'), and each has a list as its value. The dictionary may be extended dynamically merely by assigning a value to a new key, 'FR' in the example. Notice how any original order of the keys is lost, since Python dictionaries are not ordered.

To access a "multi-dimensional" object, just add the key or index inside square brackets, as you would in most other languages. They can be literals (don't forget the quotes around keys) or variables.

We show a way of iterating through a dictionary, we will be discussing this in more detail later.

QA Removing items from a dictionary

To remove a single key/value pair:

```
del dict[key]
```

- It will Raise a `KeyError` exception if the key does not exist.
- Returns `default` if the key does not exist.

```
>>> fred={}
>>> del fred['dob']
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    del fred['dob']
KeyError: 'dob'
>>> fred.pop('dob', False)
False
```

Also:

- `dict.popitem()` removes the next key/value pair used in iteration.
- `dict.clear()` removes all key/value pairs from the dictionary.

21

Deleting a key will delete the associated value as well. The value is not returned by the `del` statement but is returned by the `pop()` method. Both can raise a `KeyError` exception if the key does not exist, but `pop()` can take an optional default value which is returned instead.

The `popitem()` method removes an arbitrary key/value pair, in that the order of keys within a dictionary is not defined. It would be of use if we were iterating through a dictionary removing each key/value in turn. `popitem()` returns the key/value pair deleted as a tuple or raises a `KeyError` exception if the dictionary is empty.

QA

DICTIONARY METHODS



<code>dict.clear()</code>	Remove all items from <code>dict</code>
<code>dict.copy()</code>	Return a copy of <code>dict</code>
<code>dict.fromkeys(seq[, value])</code>	Create a new dictionary from <code>seq</code>
<code>dict.get(key[, default])</code>	Return the value for <code>key</code> , or <code>default</code> if it does not exist
<code>dict.items()</code>	Return a view of the key-value pairs
<code>dict.keys()</code>	Return a view of the keys
<code>dict.pop(key[, default])</code>	Remove and return <code>key</code> 's value, else return <code>default</code>
<code>dict.popitem()</code>	Remove the next item from the dictionary
<code>dict.setdefault(key[, default])</code>	Add <code>key</code> if it does not already exist
<code>dict.update(dictionary)</code>	Merge another dictionary into <code>dict</code> .
<code>dict.values()</code>	Return a view of the values

22

Return values from `keys()`, `values()`, and `items()` are *view objects*. These are discussed on the next slide.

Note that `copy()` returns a *shallow copy* of the dictionary, not a deep copy. See the *Advanced Collections* chapter for more on shallow and deep copies.

As we have said, Python dictionaries are **unordered**. However, this *might* be changing. Associated with other internal changes, in the C implementation of Python 3.6 the keys stay the order in which they were defined. To quote the documentation: '*The order-preserving aspect of this new implementation is considered an implementation detail and should not be relied upon*'.

It is possible that this could become a language feature in the future, but until then, if you really need an ordered dictionary then it is safer to use `OrderedDict` from the **collections** module in the standard library.

QA View objects - examples

- May be used in iteration:

```
nebula = {'M42':'Orion',
          'C33':'Veil',
          'M8' : 'Lagoon',
          'M17':'Swan'
        }
for kv in nebula.items():
    print(kv)
```

```
('M42', 'Orion')
('M17', 'Swan')
('M8', 'Lagoon')
('C33', 'Veil')
```

- To store as a list:

```
lkeys = list(nebula.keys())
print(lkeys)
```

```
['M42', 'M17', 'M8', 'C33']
```

- In set operations:

```
jelly = nebula.keys() | {'M37', 'M5'}
print(jelly)
```

```
{'M5', 'M37', 'M17', 'M42', 'M8', 'C33'}
```

xx

View objects, as returned by **items()**, **keys()**, and **values()** methods, can be used in iteration and as objects to construct a list. The Set operations, &, |, ^ and -, may be used with a set on **dict_keys** and **dict_items** view objects, but not **dict_values**.

Dictionary view objects are new to Python 3 but have also been introduced into 2.7.

QA

SUMMARY



Collections can store multiple objects.

Lists and tuples are like arrays in other languages:

- Lists are mutable.
- Tuples are immutable.
- Both can be indexed and sliced [start:end+1].
- Can contain variables.

Dictionaries are like associative arrays:

- Keys are unique.
- They are mutable.
- Ordered (in insertion order) from Python 3.6.
- Accessed by a key.

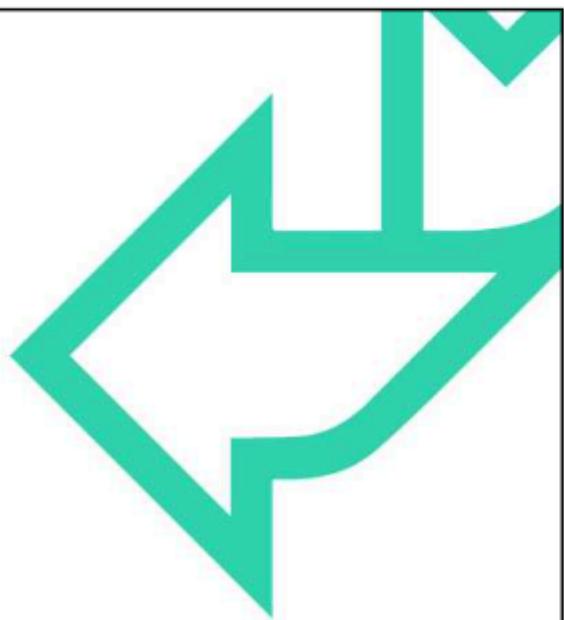
Sets are like a special version of dictionaries:

- Unordered.
- They do not have keys.
- Objects are unique.
- Can be combined with other sets.



Python 3 Programming

Regular expressions



QA

REGULAR EXPRESSIONS



Contents

- Regular Expressions
- Python Regular Expressions
- Elementary extended RE meta-characters
- Regular expression objects
- Regular expression substitution
- Matching alternatives
- Anchors
- Class shortcuts
- Repeat quantifiers
- Parentheses groups
- Back-references
- Flags

QA Regular expressions

What are they?

- A string of text with special meta-characters to help **match**, **replace**, or **validate** text.
- Often called a Regex pattern or just pattern.

Where are they used?

- Search engines, databases, SQL, word processors, and text editors.
- Web form validation.
- Command line tools such as grep, sed, awk, and vim.
- Programming languages such as C, C++, Java, JavaScript, Perl, and Python.

Types of Regex

- Two dialects defined in POSIX standard:
 - B.R.E – Basic Regex
 - E.R.E - Extended Regex
- Python supports both and many extensions.

How to Regex



Editor, Keyboard, Cat & Mouse

Firstly, it's worth pointing out that regular expressions are not a programming language, they are simply a set of text and symbols that find patterns in strings. Sounds rather simple really.

But knowing a little bit of history of regular expressions can be useful to aid our understanding and appreciate how widely they are used - and they didn't start in the computing world. Instead, it originated in the world of neuroscience in 1943 by McCulloch and Pitts describing how the human nervous system worked. In 1951, Stephen Kleene invented regular sets/expressions using algebraic notation to describe these neural networks. And the term 'regular expressions' was born.

In 1968, Ken Thompson, a mathematician and the creator of UNIX, updated a simple text editor called QED (quick editor) to support the notation for string searching. He initially ported the editor into Multics before writing a new line editor called ed for the UNIX operating system. These were the first text editors to support regular expression - but not the only tools.

Later in the 70s, other UNIX command line tools were developed that required string matching. For example, Bill Joy of BSD and Sun Microsystems fame, wrote an extended version of ed called ex before eventually writing a full screen and much 'loved' visual interface editor called vi. Ken Thompson then renamed his command line search tools to 'grep' (named after its own syntax: g/re/p – globally search for regex pattern and print line). Several tools were now supporting basic regular expressions (B.R.E).

Alfred Aho developed an extended version of grep called egrep with extended regular expressions (E.R.E). And with the collaboration of Weinberger and Kernighan, created the programmable editor called awk. Then came Perl and many other programming languages all supporting either B.R.E or E.R.E.

Now we have two dialects of regular expressions, so it was standardised in 1986 as the POSIX (Portable OS Interface uniX) standard. Most programming languages support either as a core feature or through a library module, and many have additional extensions.

They have a reputation for being difficult to understand (unless you have a well-trained cat), but they are well worth the effort to learn.

Q^ Python regular expressions

Extended regular expressions, with extensions

- Requires the **re** module (in the standard library).
- Can pre-compile an expression for performance.
- Multi-line pattern matches are supported.
- Powerful substitution.
- Replace a pattern using a variable-expression.
- Create self-referencing patterns.
- Match part of a pattern with result of previous sub-pattern.

BUT - Python string methods are powerful and fast

- Don't use REs when functions or methods will do.

Python supports Extended Regular Expressions (E.R.E) through the **re** module in the Python Standard Library and has many extensions. Although Python supports E.R.E, anyone familiar with tools like grep, vi, sed, and awk should find them relatively easy to learn. The most noticeable difference between E.R.E and the basic equivalent, is that B.R.E needs to backslash (escape) braces (quantifiers) and parentheses (groupings). In E.R.E and Python this is not required.

An overview of the differences between grep, sed, awk, and Python regular expressions:

- Python REs can be applied across multiple lines, so you can match a group of lines that match specific large patterns, without having to keep track of state yourself.
- Python substitution expressions allow you to use variables in the search text, including dictionary and list lookups using parts of the pattern just being matched.
- Python can replace a pattern or part of a pattern by the result of a function call.
- Python patterns can be self-referencing, for example, we can

```
def __init__(self, pattern, flags=0):
    re.compile(pattern, flags)

def sub(self, repl, count=0):
    return self._compile().sub(repl, self.string, count)

def subn(self, repl, count=0):
    return self._compile().subn(repl, self.string, count)

def search(self, string):
    return self._compile().search(string)

def fullmatch(self, string):
    return self._compile().fullmatch(string)

def match(self, string):
    return self._compile().match(string)

def split(self, string, maxsplit=0):
    return self._compile().split(string, maxsplit)

def findall(self, string):
    return self._compile().findall(string)

def finditer(self, string):
    return self._compile().finditer(string)
```

match a line that starts and ends with the exact same expression with anything in-between.

Just remember that there is a performance cost (CPU resource) to compiling and searching for a regular expressions – and sometimes it is cheaper, faster and easier to use Python string testing operators (==/!=/in) and methods (startswith()/endswith() etc.).

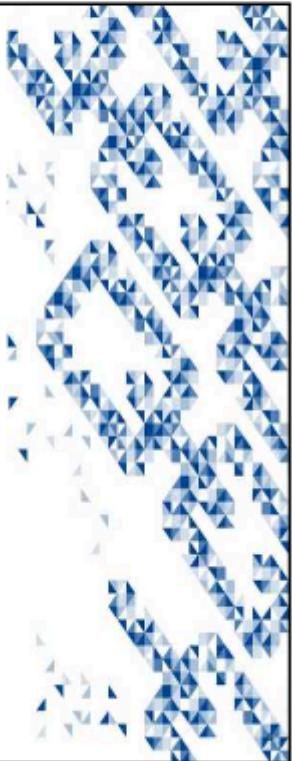
QA Core ERE meta-characters

Line Anchors	Single Char Class	Quantifiers
Start " <code>^The</code> "	Any 1 char " <code>.ing\$</code> "	0 or more " <code>:[0-9]*:</code> "
End " <code>ing\$</code> "	Any 4 chars " <code>....\$</code> "	0 or once " <code>:[0-9]?:</code> "
Escape Char Class	Char set " <code>[rdz]ing\$</code> "	1 or more " <code>:[0-9]+:</code> "
Escape " <code>\.</code> "	Char range " <code>[A-Z]ing\$</code> "	Exact " <code>:[0-9]{10}:</code> "
Escape " <code>[.]</code> "	Char ranges " <code>[A-Za-z]ing\$</code> "	Min, max " <code>:[0-9]{10,20}:</code> "
	NOT Char set " <code>[^dz]ing\$</code> "	At least " <code>:[0-9]{10,}:</code> "
	Multiple sets " <code>[aeiou][aeiou][aeiou]</code> "	

What would these match?

"`^A-Z|1,2|0-9|1,2|A-Z?|0-9|A-Z|2$`"

"`^A-CEGH]-PR-TW-Z|A-CEGH]-NPR-TW-Z|?0-9|2|?0-9|2|?0-9|2|?A-DFMP]$`"



The examples listed above cover some of the most common regular expression meta-characters. If you are new to regular expressions, this is a good place to start.

The line anchors can be used to match strings starting and ending with a pattern of characters although this capability is also available with string methods.

The expression Character Class or Character Set is introduced here. The single character class is the dot and can be used to match any single character. Historically this was to match one ASCII character, but most modern tools and languages can match a Unicode character. The square [] brackets can be used to limit the characters matched to a set of characters rather than all characters. Typing a caret after the opening bracket negates the character set. It does not matter the order that characters are defined in the set, but when using ranges (with a hyphen) then the characters must be in increasing lexical order ([a-z] rather than [z-a]).

Any meta-character can be escaped by preceding it with a backslash or placing it within square brackets – except of course the caret and hyphen which have a meaning if used correctly inside the brackets.

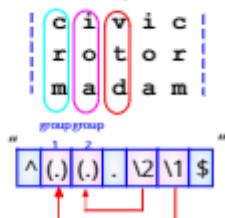
The quantifiers allow you to repeat character classes by using the ?, * and + operators. The ? repeats the entire character class 0 or once (i.e. optional), the * repeats 0 or more (be careful of the zero occurrence) and + repeats 1 or more. An important note is that Regular Expressions are GREEDY and will match the largest pattern.

So, we can limit the number of repetitions using the curly brackets {}. For example, [0-9]{10} will match exactly ten digits, {10,20} will match between 10 and 20 repetitions, and [0-9]{20,} will match at least 20 repeating digits.

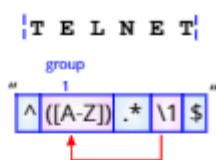
QA Core ERE meta-characters cont..

Grouping/Backreferences

1.Match 5 char palindromes



2.Match strings start/ending with same capital



Alternation(or) Class

"eric|graham|michael|terry|john"



Grouping plus Alternation

"a (bottle|glass|keg) of (lager|wine|beer)"

Will match the following strings:

"a bottle of lager"
"a glass of wine"
"a glass of beer"

What would these match?

"^(.)(.)\\$1\\$" and "^(0-9)(1,3)\.(0-9)(1,3)\.(0-9)(1,3)\.(0-9)(1,3)\\$"



This slide continues with some more useful and powerful extended regular expressions.

The alternation (pipe | char) can be used to match a single pattern out of several alternatives. For example, match 'eric' or 'graham' or 'michael' or 'terry' or 'john' or anyone else who can do a silly walk.

Parentheses can be used to group together characters into a capturing group – which can then be repeated using the quantifiers from the previous slide or combined with alternation to restrict alternation to part of the pattern – see the example matching "a glass of wine" and "a glass of beer" for reference.

Backreferences allows us to match the same text as previously matched in a capturing group.

For example, we could use '^([A-Z]).*\\$1\\$' to match a string starting and ending with the exact same capital.

- a. The caret ^ matching the start of text, followed by...

- b. A capture group 1 () with a single character set to match a capital [A-Z], followed by...
- c. A filler .* that matches 0 or more of any characters, followed by...
- d. A back reference to match same character as group 1, followed by...
- e. The \$ matching the end of text.

This could also be used to match palindromes which have repeating identical characters in the text. See slide for solution.

An alternative solution for the Grouping plus Alternation example on the slide would be to use “a (?:bottle|glass|keg) of (?:lager|wine|beer)”. The (?:) is called a non-capturing group as it is a way to group a set of characters without capturing the matched text – if you don’t need to access the groups later and is more efficient.

Q&A Regular expression objects

Must import re module:

- Can pre-compile the RE for efficiency and returns an RE object.

We can search or match:

- `search()` searches for a pattern anywhere in a string.
- `match()` matches from the start of the string.
- `fullmatch()` matches from the start to the end of the string (3.4).
- On failure raises a `re.error` exception

Returns a MatchObject or None (False):

```
testy = 'The quick brown fox jumps over the lazy dog'

m = re.search(r"(quick|slow).*?(fox|camel)", testy)
if m:
    print('Matched', m.groups())
    print('Starting at', m.start())
    print('Ending at', m.end())
```

Matched ('quick', 'fox')
Starting at 4
Ending at 19

Importing the `re` module allows methods on the `re` class to be called, with `search()` and `match()` being the most common. The `fullmatch()` method was introduced at Python 3.4. They all return an object of class `MatchObject` or `None` (if not matched) and in this slide it is simply named `m` (for match object). Bond would approve.

The match object `m` can then be tested as `None` will evaluate as `False` in Boolean context and a `re.error` exception (enhanced in Python 3.5) is raised. A `MatchObject` will evaluate as `True`.

Remember, the group in parentheses is also known as a *capturing parentheses group* which means text inside the group may be referred to later in the pattern as a back-reference.

The `MatchObject` has a useful method called `groups()` which returns a tuple containing all the matched text for all the capture groups and may be used like back-references. We could use Tuple indexing to extract a particular group, for example, `m.groups()[0]`

will return the matched text from group 1.

A more Pythonic (and readable) way to extract text from group 1 is to use the `group(1)` method with the correct group number as a parameter, for example, `m.group(1)`. Without parameters, the `group()` method will return only the matched string.

Other methods include `start()` and `end()`, which return the starting and end positions of the pattern in the text. There are several `MatchObject` attributes, including `re`, which gives the original regular expression, and `string` which gives the original input.

Python RE syntax is like that used by lower-level language libraries, such as the GNU C regex package. We can also compile our REs for efficiency, for example:

```
reobj = re.compile (r"([Ii]).*(\1)")  
for line in file:  
    m = reobj.match(line)  
    if m:  
        print(m.string[m.start():m.end()])
```

Notice the use of raw strings (`r"..."`), these mean we do not have to escape (\) special characters like brackets and braces – particularly useful with regular expressions.

Q^Regular expression substitution

The `sub` function returns the modified string:

- `re.sub(pattern, replacement, string[, count, flags])`

The `subn` functions returns a tuple of (*modified string, number of changes*):

- `re.subn(pattern, replacement, string[, count, flags])`

The optional `count` argument determines the occurrence to modify:

- Default is to replace *all* occurrences.

```
line = 'Perl for Perl Programmers'  
cs, num = re.subn('Perl', 'Python', line)  
if num:  
    print(cs)          Python for Python Programmers  
  
cs, num = re.subn('Perl', 'Python', line, 1)  
if num:  
    print(cs)          Python for Perl Programmers
```

Substitution is reminiscent of the awk scripting language, in that we have a couple of method calls – `sub()` just returns the modified string and `subn()` returns a tuple of the modified string and the number of substitutions. Both perform global substitutions from the left of the string.

There are other useful `re` functions, for example `split()`, which is shown on the next slide.

We have also shown a compiled Regular Expression object. Compiling the Regex pattern makes for more efficient code when the same pattern is used many times, for example in a loop. Methods like `match()`, `fullmatch()`, `findall()`, `search()`, `split()`, `sub()`, and `subn()` may be called on these objects.

Q& Regular expression split

Similar in functionality to the string split:

- Uses a Regular Expression for the separator instead of a string.
- `re.split(pattern, string [, max_splits=0, flags=0])`
- Note the default value for `max_splits` is zero which means no limit.

Only use the RE version if you need alternative separators.

- The string version is more efficient.

```
import re
line = 'root;;0.0:superuser,/root;/bin/sh'
elems = re.split('[:;,]', line)
print(elems)
['root', '', '0', '0', 'superuser', '/root', '/bin/sh']
```

String objects have the method `split()` that splits a string using a given field delimiter and returns a list of objects. This can be useful when reading in lines of text from a file and the text is to be modified. String objects are immutable, but list objects are mutable, can be modified and joined back into a string using the `string join()` method.

The `re` module's version of `split()` enables a regular expression to be used to describe the field delimiter (this is much like `split` in Perl and PHP). In the example on the slide, the `re.split()` method has a set `[;,]` of alternative characters than can be used. It supports an optional max number of fields to return.

The optional parameter `flags` was added at 3.1 (see later) and can be used, for example, to ignore case.

Q& Matching alternatives

The | character separates alternative words or patterns.

```
drink = 'A glass of Coors'  
if re.search(r'Bud|Miller|Coors', drink):  
    print("It's a beer!")
```

Use parentheses to group alternatives.

- Required with text before or following alternatives.

```
pattern = r'A (glass|bottle|barrel) of (Bud|Miller|Coors)'  
if re.search(pattern, drink):  
    print("This drink is suitable for Americans")
```

The first example in the slide can match three alternative patterns in the text and in this instance will match 'Coors' and evaluate to True for the if statement.

The second example above can match nine different possibilities.

A side effect of the parenthesis-notation is that they capture the matched text in the groups and can be extracted for further use - which we will see later.

QA Anchors

The ^ and \$ characters indicate start or end of text.

- Only when used at start or end of pattern.

```
name, old, new = sys.argv[1:]  
  
new_name = re.sub(fr"\.{old}$", f".{new}", name)  
print(f"Renaming {name} to {new_name}")  
os.rename(name, new_name)
```

Shortcuts \b matches a word-boundary and \B not a word-boundary:

```
txt = 'Stranger in a strange land'  
m = re.search(r'range\b', txt)  
print(m.start())  
  
txt = 'Stranger in a strange land'  
m = re.search(r'range\B', txt)  
print(m.start())
```

16

2



The \$ anchor is special: if you use '00\$', it will match either two zeroes at the end of the search text, or two zeroes followed by a newline at the end of a search text – that is, it automatically ignores a new-line at the end of a line.

When the ^ character is used anywhere except at the start of a pattern, it indicates a normal ^ character.

When you have a search text that contains multiple lines, the ^ and \$ anchors apply to the whole of the text. If you use the m flag (see later), they will be applied to each individual line within the search text.

For single-line matches, this is of no importance, but for multi-line matches genuine start of text can be marked with \A, and end of text with \Z.

In addition to the start and end of line anchors, we have the word anchor \b. It indicates a word boundary (either the beginning or the end of a word), but, like ^ and \$, does not take up any space.

Exactly what constitutes a word boundary is, however, not always intuitive when it comes to apostrophes. The non-word boundary anchor, \B is used when we explicitly want the text imbedded in another word.

Beware! When used in square brackets (a character class) \b means a single back-space character!

Q\Shorthand Character Classes

A Character Class describes a set of characters:

- For example: [a-z] [^A-Z] [aeiou]

A Class shortcut matches a pre-defined character class:

Digit	\d	[0-9]	\D	[^0-9]
Word char	\w	[A-Za-z0-9_]	\W	[^A-Za-z0-9_]
Whitespace	\s	[\t\r\n\f\n]	\S	[^ \t\r\n\f\n]
Word boundary	\b	[\t\r\n\f\n+-]+()	\B	[^ \t\r\n\f\n+-]+()
Line anchors	\A	^	\Z	\$

```
m = re.search(r'^ttyp\d$', port)
if m:
    print(port)
```

ttyp0
ttyp9

Exact meaning can be changed with flags...

Python supports the same single character classes and sets that B.R.E and tools like sed and awk support. But Python also supports some shorthand character classes using escape chars such as \d (digit), \s (whitespace) and \w (word character). These are outlined in the table in the slide and can also be used in characters sets, for example, [\da-fA-F] to match a hexadecimal number or within groups.

Currently (as of Python 3.12), the Python re module does not support POSIX character classes such as [:digit:], [:space:], and [:alnum:]. This may change in the future.

Python also supports an additional \g shortcut for named groups and is discussed later.

Q4 Repeat quantifiers

Quantifier characters repeat the preceding pattern:

0 or once	?	[+]??	Optional + or -
0 or many	*	[0-9]*	0 or more digits
1 or many	+	[A-Z]+	1 or more uppercase chars

```
m = re.search(r'[::,]?\s*\w+', line)
```

optional ::, followed by zero or more whitespace,
followed by at least one alphanumeric.

```
m = re.search(r'boink+', sound)
```

boink, boinkk, boinkkk

```
m = re.search(r'(boink)+', sound)
```

boink, boinkboink, boinkboinkboink

The repeat-quantifiers described here are greedy: they eat as much as possible, while not breaking the rest of the pattern. If we have the text 'The dog eats dog-food', then the pattern 'The.*dog' will match the text 'The dog eats dog'.

Python also supports *minimal* repeat-quantifiers, that eat the least amount possible while still making the match work. This is achieved by appending a question mark ? after the repeat quantifier.

Q\ Quantifiers

Repeat an indicated number of times..

exact	{3}	[0-9]{3}	3 digits
min,max	{3,5}	[0-9]{3,5}	between 3 and 5 digits
at least	{5,}	[A-Z]{5,}	at least 5 digits
up to	{0,5}	[A-Z]{0,5}	up to 5 digits

Match a U.S. telephone number:

- Start of text or a non-digit character.
- Followed by three digits followed by a hyphen.
- Followed by between 2 and 4 digits.
- Followed by an optional whitespace.
- Followed by between 4 and eight digits.
- Followed by a non-digit character or end-of-text.

```
m = re.search(r'(^|\D)\d{3}-\d{2,4}\s?\d{4,8}(\D|$)', phone)
```

4

In their simplest form, quantifiers specify the number of characters, for example, to match a line of at least 80 characters:

.{80,}

QA Back-references

Python also allows substitution strings to reference groups within the pattern

- To create self-referencing regular expressions.

Referenced by `\n` or `\g<n>`, representing the 'nth' group

- Don't forget to use a raw string.

Can also be used in the replacement string in `sub()` and `subn()` functions

```
import re
from datetime import date
year = str(date.today())[:4] ← Get current year
strn = 'copyright 2005-2006'
print(re.sub(r'((19|20)[0-9]{2})-((19|20)[0-9]{2})', r'\1-' + year, strn)
      copyright 2005-2016
```

Python supports the use of parentheses to group a number of characters or regular expressions into a single unit and then apply a regular expression quantifier to the entire group. This can be useful when the pattern consists of recurring blocks of text or words.

The group in parentheses is also known as a capturing parentheses group.

Text inside parentheses may be referred to later in the RE as a back-reference. This is done by the use of `r'\1'`, `r'\2'` etc. to refer to the contents of the first and second sets of parentheses.

If you nest parentheses, the order of the opening parenthesis is the order in which the back-references are allocated. Make sure you use 'raw' strings for the back-references, since `'\1'` is itself a special character.

The alternative back-reference syntax using `r'\g<1>'`, `r'\g<2>'` etc. is a Python extension and is associated with named captures (see the

Advanced Regular Expressions appendix). It has the additional feature of supporting group 0 (zero) in `r'\g<0>'`, which represents the whole of the matched string (`r'\0'` is not supported).

The use of many back-references in a RE will cause Python to work harder and increase the time taken to find a match. Use sparingly!

Q& Global matches

re.findall

- Returns a list of matches or groups.

```
home_stuff = '/dev/sda3 135398 69683 52176 57% /home/stuff'
nums = re.findall(r'\b\d+\b', home_stuff)
print(nums)
```

['135398', '69683', '52176', '57']

re.finditer

- Returns an iterator to a match object.

```
home_stuff = '/dev/sda3 135398 69683 52176 57% /home/stuff'

for m in re.finditer(r'\b(\d+)\b', home_stuff):
    print(m.groups())
```

('135398',)
('69683',)
('52176',)
('57',)

By default, most re methods search for the first (leftmost) occurrence of a pattern but the `findall()` method iterates over the text finding and returning all the matches in a list.

The `finditer()` works the same except it returns an iterator object which can be combined with an iterator for loop to extract all matches.

These are particularly useful for repeated patterns over multiple lines.

The `findall()` method was added in Python 2.2.

In Python 3.7, non-empty matches for both can now start just after a previous empty match.

QA Flags

Change the behaviour of the match

Long Name	Short	RE	
re.IGNORECASE	re.I	(?i)	CaSe insensitive match
re.MULTILINE	re.M	(?m)	^ and \$ match start and end of line
re.DOTALL	re.S	(?s)	DOT also matches a newline
re.VERBOSE	re.X	(?x)	Whitespace is ignored to allow comments

- Can be embedded in the RE.
- Can be applied to parts of the string (3.6 – *modifier spans*).
- Can be specified as an optional argument to search, match, split, sub, etc.
- Can be combined with '|' separator.

```
m = re.search(r'(?im)^john', name)
m = re.search(r'^john', name, re.IGNORECASE|re.MULTILINE)
m = re.search(r'^(?i:j)ohn', name)
```

The optional *flags* parameter was added to the `re` methods at Python 3.1 and can be used to change the behaviour of the match. For example, the `re.IGNORECASE` or its shorthand equivalent `re.I` (that is an upper-case aye!) do the obvious.

It would be tempting to state that the short names are the initial letter of the long name, and that the RE syntax is just the short name in lowercase. You can see that this is not the case, the S and X flags are there for compatibility with other RE engines.

The first two examples combine the IGNORECASE and MULTILINE flags. They look for 'john' in any case at the start of the text or immediately after a new-line character. The third example is a *modifier span* and only applies the i (ignore case) to the letter j – so John or john but not JOHN.

When embedded in the RE the single characters can be in any order. When using the `re` module attribute flags, they are

Long name	Short	RE	
re.ASCII	re.A	(?a)	Class shortcuts do not include Unicode
re.LOCAL	re.L	(?L)	Class shortcuts are locale sensitive

even more confusing.

There are two additional flags `re.ASCII` and `re.LOCAL` that are not shown on the slide as they are either deprecated (Python 3.5) or only supported (in Python 3.6) for byte objects for backward compatibility and are not recommended.

QA

SUMMARY



Regular expressions are used by many programs.
Regular expressions create a MatchObject on match.

Several functions available:

- | | |
|--------------|--|
| re.search | - Find a pattern somewhere in the string. |
| re.match | - Match from the start of the string. |
| re.fullmatch | - Match from start to end of string (3.4). |
| re.sub | - Substitute the pattern, returning the new string. |
| re.subn | - Substitute the pattern, returning the new string and a count of substitutions. |

Support many char classes including:

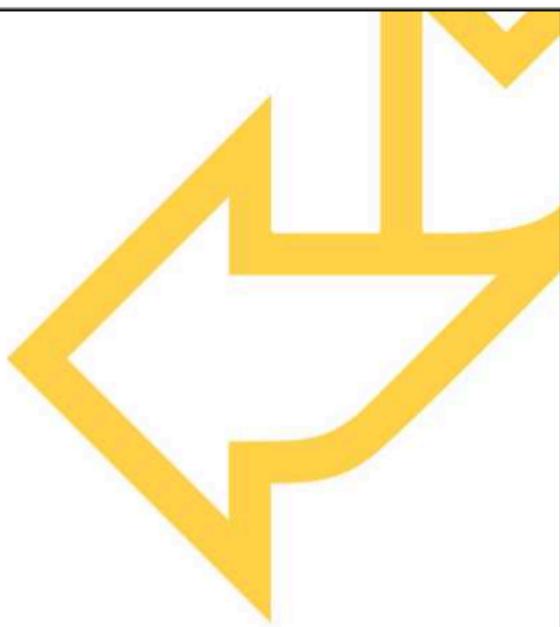
Class shortcuts	. \w \d \s \W \D \S
Alternatives	
characters	[]
strings	
Repeat qualifiers	? * + {m,n}

»



Python 3 Programming

Data storage and file handling



QA

DATA STORAGE AND FILE HANDLING



Contents

- File objects
- Reading files
- Writing files
- Standard streams
- More tricks
- Random access
- Pickles
- Shelves
- Compression

Summary

- Database interface overview
- Binary files: pack/unpack

2

Python has input/output features which are very easy to use, with syntax reminiscent of C and PHP.

QA Interacting with files

Useful programs often read and write to files.

- How do we open, close, and identify files?
- How do we want to interact – read, write, or append?
- Sequential or random reading?



One of the most common tasks for a program is to interact with file system objects such as files. A file system can accumulate a large number of files with different formats and different data within. There are many Python modules to deal with specific types of files such as html, xml, json, and csv.

Python has several built-in functions and objects to open, close, read, and write strings and bytes to files. But there are several important considerations we to make. How many files can we open at any given time and how do we identify them? How do we open files for reading, writing, and appending or some combination of these. Are we restricted to reading and writing lines sequentially or can we seek faster through a file to a specific position? And do we have to close files explicitly or can they be closed automatically? Can we just write data, or can we preserve Python structures like lists and dictionaries to files?

These are some of the considerations that we will examine in this chapter.

QA New file objects

File objects are created with the open function:

```
FileObject = open(filename, mode='r', buffering=-1, encoding=None, errors=None,  
newline=None, closefd=True, opener=None)
```

Valid open modes :

- 'r' open existing file for read (default)
- 'w' open file for write, create or overwrite existing file
- 'a' open file for append, create if does not exist
- 'x' create file and open for write, fails if file exists (3.3)
- 'r+' open existing file for read/write
- 'w+' create & truncate file for read/write
- 'a+' create & append file for read/write

File will be closed on exit or better still closed explicitly:

```
FileObject.close()
```

4

Files are accessed through file objects, which are created using the open function. A 'b' must be appended to the mode if the file is non-text, and optionally 't' may be appended for text files (which is the default).

All files are automatically closed (and flushed) when the program exits, or the file object is destroyed (drops out of scope). It is usually considered to be good programming practice to explicitly close the file as soon as possible, and this may be done by calling the close method.

The mode 'x' was introduced at Python 3.3 and is used to exclusively create and open the file. If the file already exists, then this will fail with FileExistsError: [Errno 17] File exists.

Python 3 also supports the 'U' (universal newline) mode, but that is provided for compatibility reasons only - it is no longer required and is deprecated.

The buffering argument can be set to zero on binary files, which

means buffering is switched off. A value of one (1) means line-buffering, and a greater value gives the actual size of the buffer required. The default, -1, will choose a suitable buffer size for the system in use.

There are other parameters available to open, but they are not often needed: encoding=None, errors=None, newline=None, closefd=True). See the on-line documentation for details.

A further argument, opener=None, was added at 3.3. This allows us to provide an alternative method for opening the file.

QA Reading files into Python

Create a file object with `open()`:

```
infile = open('filename', 'rt')
```

Read *n* characters (in text mode):

- This may return fewer characters near end-of-file.
- If *n* is not specified, the entire file is read.

```
buffer = infile.read(42)
```

Read a line

- The line terminator "\n" is included.
- Returns an empty string (False) at end-of-file.

```
line = infile.readline()
```

Once a file object has been created, we can read from the file using a method, and there are several to choose from.

With `read()` we can specify the number of characters to be read, starting at the current file position. In binary mode (see later), the number given is the number of bytes, not Python characters (which are multi-byte). Related to this, we also have `seek()`, which moves the current file position to a specified offset, and `tell()`, which returns the current file position.

Reading a text line, a record terminated by a new-line character, may be done using `readline()`, and this is a very common way of accessing text files from Python. Notice that the record terminator is included in the buffer and can be removed using a slice `[:-1]` or the `str.rstrip("\n")` method.

QA Reading tricks

Reading the whole file into a variable

- Be careful of the file size.

```
lines = open('brian.txt').read()
llines = open('brian.txt').read().splitlines()
linelist = open('brian.txt').readlines()
```

Reading a file sequentially in a loop

- Inefficient:

```
for line in open('lines.txt').readlines():
    print(line, end="")
```



- Use the file object iterator:

```
for line in open('lines.txt'):
    print(line, end="")
```



With the first three examples, the first (lines) reads the whole file into one string. The second (llines) reads the whole file into a string, but then produces a list from the splitlines. The third example (linelist) is similar in that it produces a list, but the new-line characters remain.

When reading each line in a loop (a common requirement), it is generally better to invoke the file iterator. Calling readlines in a for loop will read the entire file into a temporary list in-memory then iterate through that, which is rather inefficient.

Note the end parameter in the print statements. This prevents an additional new-line from being printed - we already have one at the end of each record.

Incidentally, do not use seek (random access) in the for loop - it upsets the file iterator, and you can end-up in an infinite loop.

Q\ Managing file handles using try/except/finally

Under very rare circumstances, a file could be left open

- An error causing an unhandled exception.
- Leading to lost data or memory leaks.

Use Exception Handler – try/except/finally

- Ensures file handle is closed.
- Can catch named exceptions during opening.
- Finally block always executes.

```
fh_out = open('spam.txt', 'wt')

try:
    fh_out.write('Spam, spam, spam!')
except FileNotFoundError as err:
    print('Blooming Vikings! ')
finally:
    # Always close file handle after use
    fh_out.close()
```

A common problem with interacting with external resources like files is remembering to release them once finished reading and writing. If not released, your program may retain these resources indefinitely causing memory leaks or losing unwritten data forever.

Most programmers will devise a setup and teardown phase in their programs, with the teardown phase performing clean up actions such as closing files, releasing locks, closing network or database connections etc. In the case of writing to files, it is likely that data will be buffered in memory and in error conditions, this data may be lost if the file resource is not properly closed. So, it is important to use the file handle `close()` method to flush buffers and release the resource.

However, to mitigate against exceptions occurring before the resource is closed, we can use two strategies – a `try/finally` construct or a context resource manager (the `with` statement).

One approach is to wrap the `open()` function in a `try/except/finally` block. Any errors raised in the `try` block will be caught and handled

in the except block, and the finally block (which always executes) will close the file handle.

The try/except/finally construct is discussed in more detail in a later chapter.

Q^A Managing file handles using Context Resource Manager

Under very rare circumstances, a file could be left open.

- An error causing an unhandled exception.

Some python classes are context managers.

- `io` is the most common - file objects are context objects.
- Uses the `with` keyword.

Ensures files are closed on error.

- This usually happens anyway with a for loop.
- This is rarely needed - but safer!

```
with open('spam.txt', 'rt') as fh_in:  
    for line in fh_in:  
        print(line, end='')
```

`fh_in` is a file and context object
And only exists for the block.

PEP 343 introduced an alternative approach to managing resources, like file handles, using a Context Resource Manager – the `with` statement. This can reduce the need for using a `try` block with arguably more elegant, efficient and safer code. The `with` statement creates a runtime context that is controlled by a context manager, that is, it has an `__enter__()` and `__exit__()` methods that can setup and teardown resources.

One such object that implements the context management protocol is the file handle object. The general syntax is:
`with expression [as variable]:`

BLOCK

The expression must return a context object, with its `__enter__()` method called, and if the `as` keyword is used, then the variable will be bound to the `__exit__()` method and called when the BLOCK completes. In very simple terms, it emulates block variable scope in other languages.

Many Pythonistas prefer this method of iterating through a file.

Using a context manager gets over reliability issues with destructors (see later).

So, you probably don't need to use this, but do you want to take the risk? We suggest that you don't change existing code to use this mechanism but consider using it for new code.

Q Writing to files from Python

Open a file handle with `open`:

- Specifying write or append.

```
output = open('myfile', 'w')
append = open('logfile', 'a')
```

- Write a string.
- Append "\n" to make it a line.
- Returns the number of chars (text) or bytes (binary) written.

```
num = output.write("Hello\n")
```

- Write strings from a list.
- Append "\n" to each element to make lines.

```
output.writelines(list)
```

Just as there are several ways of reading from a file, there are a couple of ways of writing. The `write` method can be used to write characters to a file at the current position but remember to include the new-line terminator when writing to text files.

In Python 2, `write` did not return anything, but in Python 3 it returns the number of characters or bytes written, depending on how the file has been opened.

Writing records from a list uses the `writelines()` method. Its name is a little misleading, it does not add line terminators (new-lines) to the end of each line, they must already be in the data.

QA Binary mode

By default, open modes are text:

- Reading and writing uses native Python strings.
- Remember that Python 3 strings are multi-byte (Unicode).

Open a file as binary using 'b' with the mode:

- Reading and writing uses bytes objects, not Python strings.
- Convert to a Python string using `bytes.decode()`

```
for line in open('lines.txt', 'rb'):  
    print(line.decode(), end="")
```

- Can also write a bytes object.
- Convert from a Python string using `string.encode()`

```
fh_out = open('out.dat','wb')  
num_bytes = fo.write(b'Single bytes string')  
my_string = "Native string as a line\r\n"  
num_bytes = fh_out.write(my_string.encode())
```

Many programming languages, such as C and Perl, support a binary open mode on Windows. This is because the Windows operating system supports text and binary files differently. On UNIX, with those other languages, we just open the file for read or write, there is no difference between text and binary.

Not so with this language. With old fashioned languages text is single-byte character based, with multi-byte characters being the exception. In Python 3, (Java, and .Net) strings are multi-byte, and that is what is used if a file is opened as text, which is the default. Writing a Python string still looks like ordinary text in the file if the characters are within a single-byte character set, like ISO Latin 1.

Setting binary mode, even on UNIX, forces single-byte characters, a byte object. Fortunately, it is easy to convert between bytes and strings, and the methods which can be applied to a byte object are like those for strings.

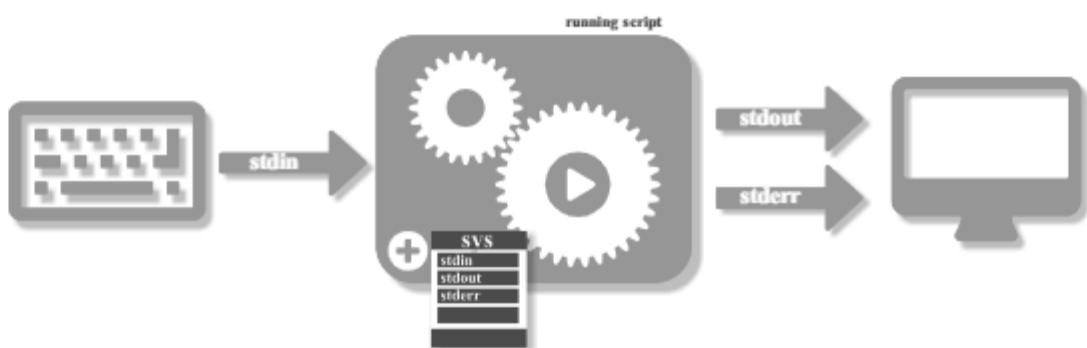
Text mode (the default) also handles line-endings relevant for the platform you are running on. With binary mode, there is no line-

ending handling, so if you want end-of-line you have to explicitly say so.

QA Standard streams

The **sys** module exposes **stdin**, **stdout**, **stderr** as open file objects.

- No need to open/close these standard IO streams.
- Import **sys** for visibility of names **stdin**, **stdout** and **stderr**.



Windows, like Unix/Linux, uses three standard “files” for keyboard input, displaying output and printing error messages to the screen. These IO channels can be used or ignored by processes but give them the ability to interact with the user and display messages to the screen. The three channels are often numbered 0, 1, and 2, but are also given file handle names of **stdin**, **stdout** and **stderr**.

The Python programming language supports these channels allowing built-in and module functions to request input and generate output and errors without having to explicitly open or close them. However, to use them by name requires the importing of the **sys** module.

The built-in **input()** function reads from the **stdin** file handle, and the **print()** function writes to **stdout** by default.

Q Standard streams

The `sys` module exposes `stdin`, `stdout`, `stderr` as open file objects.

- This example works on Python 2 and 3.

```
import sys
sys.stdout.write("Please enter a value: ")
sys.stdout.flush()
reply = sys.stdin.readline()
print("<", reply, "> was input")
```

Please enter a value: one
< one
> was input

Simple keyboard (stdin) input.

- This example works on Python 3 only.
- The "\n" is stripped from rhs.
- Remove additional whitespace entered using `rstrip()` method.

```
reply = input("Please enter a value: ").rstrip()
print("<", reply, "> was input")
```

Please enter a value: two
< two > was input

The three standard IO streams, `stdin`, `stdout`, and `stderr`, can be accessed directly through the `sys` module. Each one is a file object by that name and may be used as any other file object.

In addition, the `input` statement reads from `stdin`, which is usually the keyboard but may have been redirected. If the `readline` module is imported first, then `input()` will use it to provide line editing and history features. The `input()` function was called `raw_input()` in Python 2.

On Windows cmd.exe, line endings are "\r\n", and in Python 3.2.0 a bug stripped out the new-line but not the "\r". Fortunately, this bug was fixed in 3.2.1., but the work-around, using the `str.rstrip()` method, is sometimes a good idea anyway in case the user adds trailing spaces.

Normally standard streams like `stdin` and `stdout` are used in text mode, but if you wish to use them in binary mode then use the `-u` command-line option, for example:

```
python -u myprog.py
```

Other tricks with standard streams will be revealed later in the course.

QA More tricks

print normally writes to stdout, but can write to other file handles:

```
fh_out = open('myfile', 'wt')
print("Hello", file=fh_out)
print("Oops, we had an error", file=sys.stderr)
```

File writing is normally buffered:

- To flush the buffer:

```
fh_out.flush()
```

Emulating the Linux tail -f command in Python:

```
import time
while True:
    line = fh_out.readline()

    if not line:
        time.sleep(1)
        fh_out.seek(fh_out.tell()) ←
    else:
        print(line, end="")
```

Assumes the file is
open for read

Set the file position
to EOF

In Python 3, the `print()` function may be used for writing to a file using the `file=` parameter. In earlier versions of Python "redirection" notation was used, and the `print` shown would have looked like this:

```
print >> output, "Hello" # Python 2
```

Buffering can also be turned off using the `-u` command-line option to `python`, or by setting the `PYTHONUNBUFFERED` environment variable to a non-empty string.

The final example requires some explanation: `tail -f` is a UNIX command which displays lines as they are appended. It works by reading to the end-of-file then waiting for one second. If a record has been added, then it will be displayed, otherwise we wait for one second again. There are several ways of improving this example!

Possibly the most peculiar aspect of this example is setting (`seek`) the current file position. When we hit end-of file the current file position becomes invalid, the line

```
fh_out.seek(fh_out.tell())  
restores the file position to (physically) the same position as  
before.
```

QA Random access

Access directly at a position, rather than sequentially:

- Of limited use with text files - all lines must be the same length.
- Get the current position with the `tell()` method.
- Set the position with `seek(offset[, whence])`
- Binary access may be required for the correct offsets.
- Offsets are in bytes, not characters!

```
fh_in = open('country.txt', 'rb') ← Binary access

index={}
while True:
    line = fh_in.readline()
    if not line: break
    fields = line.decode().split(',')
    index[fields[0]] = fh_in.tell() - len(line) ← Construct an
                                                index keyed on
                                                the first field

key = input('Enter a country:')
fh_in.seek(index[key])
print(fh_in.readline().decode(), end="")
```

Most file IO, particularly with text files, is sequential. To get to a specific record in this way can be slow, particularly if we are doing this many times with the same data. Instead, we can access the file randomly, or by position.

Unless we can calculate the position in some way, we need to know where each record is, so an index can be constructed - the obvious way to do that is to use a dictionary (which can be saved in a pickle).

Notice that we open the file in binary mode. This is important on Windows since the '`r`' is hidden in text mode and would mean our offsets would be one byte out. means that byte objects are read, rather than strings.

You may be wondering why we have used a convoluted while loop to read the file and create the index. This is because `tell()` is disabled inside a for loop based on `readline` or `open`. The iterator (a method called `next()`) may read-ahead into an internal buffer for efficiency, so the current file position might not actually be where

you think - for example, a small file might be read into memory in one go. Therefore, the method shown might be slower than using a for loop and accessing the file sequentially!

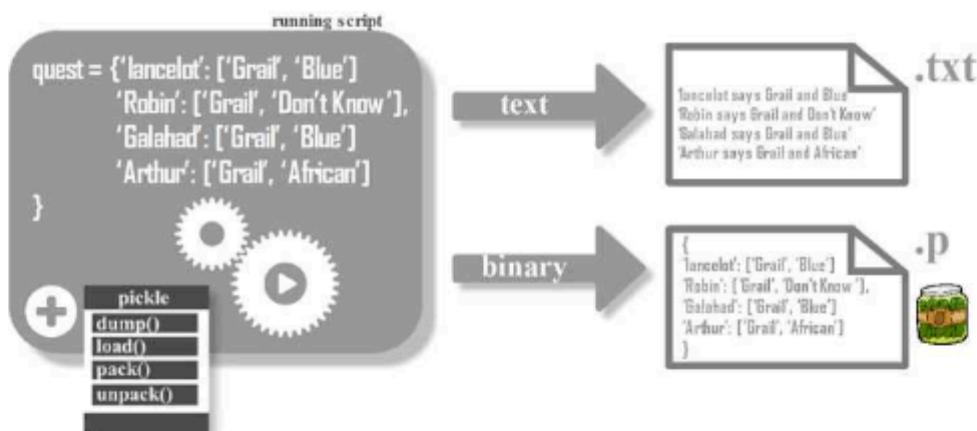
When we have a position, we can seek to that point, then read or write. By default, the position (offset) is relative to beginning of file, but we can specify a different whence value:

0	offset is relative to beginning of file
1	offset is relative to the current file position can supply a negative offset to overwrite a record just read
2	offset is relative to end of file it is not an error to seek beyond end-of-file on most file systems

QA Python pickle persistence

Pickling converts Python objects into a stream of bytes:

- Which can be written to a file or across a network.
- It means we can save data structures as well as data.



We have already seen how to read, write and append text and binary DATA to a file. But the original python object structure in memory is lost and could prove very difficult to recreate after loading the DATA from the file. Think about how tricky it is to recreate an empty tin can after standing on it.

Pickling is a process whereby a Python object structure (to whatever level) is converted into a byte stream (serialising) and preserving the DATA and STRUCTURE to a file. And unpickling is the inverse operation. Its operation is similar to how JSON files work on Java, but the pickle module can serialise the data into text and binary format.

Nowadays the pickle protocol is commonly used to transmit data and commands from one process to another process on a local or remote machine, rather preserving to disk. If your application is dealing with very large data such as Numpy/Pandas dataframes, then you might want to consider 3rd part modules such as Dask, PyArrow, and IPyParallel which have more efficient serialisation.

Q Python pickle persistence

Pickling converts Python objects into a stream of bytes:

- And the code to preserve your Python object is simple.
- The file must be opened in *binary* mode.

```
import pickle  
  
caps = {'Australia':'Canberra', 'Eire':'Dublin',  
        'UK':'London', 'US':'Washington'}  
  
outp = open('capitals.p', 'wb') ←  
pickle.dump(caps, outp)  
outp.close()
```

Using 'b' to indicate binary

```
import pickle  
  
inp = open('capitals.p', 'rb') ←  
caps = pickle.load(inp)  
inp.close()
```

- Without pickle, how would you preserve complex data structures?

Pickling is a Python way of storing its own object types in a file.
Make sure the file is opened as binary.

What can be pickled?

None, True, and False

integers, floating point numbers, complex numbers

strings, bytes, bytearrays

tuples, lists, sets, and dictionaries containing only pickleable objects

classes that are defined at the top level of a module

The python documentation also says that functions can be pickled, but this is misleading because only the function names survive. Code itself cannot be pickled easily, that is what a module is for. In fact, the .pyc compiled module file is very similar to a pickle. Python internal serialization (pyc files) can be exposed by the marshal module. That is more primitive than pickle and offers fewer features.

QA Pickle protocols

How would you like your data pickled?

- Currently 5 different protocols to choose from, pickle.protocol=n
 - 0 ASCII, backwards compatible with earlier versions of Python
 - 1 Binary format, also backwards compatible
 - 2 Added in Python 2.3. Efficient pickling of classes
 - 3 Added in Python 3.0. Not backward compatible
 - 4 Added in Python 3.4. Very large objects and more kinds of objects (default)
 - 5 Added in Python 3.8 Support for out-of-band data

The pickle module has protocol attributes.

- HIGHEST_PROTOCOL and DEFAULT_PROTOCOL

```
import pickle

outp = open('capitals.p', 'wb')
pickle.dump(caps, outp, pickle.HIGHEST_PROTOCOL)
outp.close()
```

None of these protocols are secure.



Pickle files can be written in several formats. If in doubt, use the default.

If ASCII is used, you do not need to open the file as binary, but the same mode should be used for reading and writing. ASCII is useful for debugging, and binary format is useful if you require earlier Python programs to read your files. Otherwise, the default (which is dependent on version of Python) is your best bet.

Pickle had problems in Python 3.0 when creating pickles for Python 2 (protocols 0-2). This was fixed in Python 3.1, but the fix meant that protocols 0-2 cannot be passed between Python 3.0 and 3.1. So, if passing between Python 3 versions use protocol 3, if passing between Python 3 and Python 2 then make sure you upgrade to 3.1.

Pickle protocol 4 was added at Python 3.4 and adds support for additional and very large objects such as data frames. It also has improved performance and is not backward compatible.

Pickle protocol 5 was added at Python 3.8 to support out-of-band data buffers. This is a logically independent transmission channel between a pair of connected stream sockets and allows applications to send extra meta data. It also has improved performance for in-band data.

QA Build some shelves

We often wish to dump keyed structures.

A **shelve** is a keyed pickle dumped to a database:

- Looks just like an ordinary dictionary.
- Uses a simple bundled database system, usually `dbm`.
- You only need methods: `open()`, `sync()`, `close()`

```
import shelve
db = shelve.open('capitals')
db['UK'] = 'London'
...
db.close()  close() does a sync() - like a commit
```

```
db = shelve.open('capitals')
print(db['UK'])
db.close()
```



The `shelve` module is worth considering if you are going to use pickling on a large scale. It uses a database (usually `dbm`, depending on the environment) to store pickled objects by a string key. You should not consider the database to be portable, nor should you assume that it can be used by other (non-shelve) tools.

Once we open the database, (different pickle protocols can be specified) then it is exposed as if it was an ordinary dictionary, for both read and write.

Note that you should not attempt to have more than one program (or thread) reading or writing the file at the same time. This will be prevented by file locking on most operating systems.

QA Compression

The standard library includes several modules to compress files.

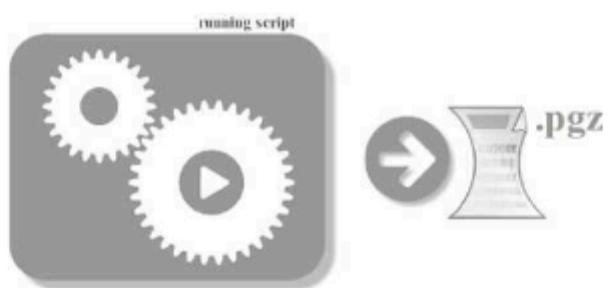
- One popular module is gzip which provides its own open() function.
- Then call the usual methods on the file handle.
- Often used with pickles.

```
import pickle
import gzip

f_outp = gzip.open('capitals.pgz', 'wb')
pickle.dump(caps, f_outp)
f_outp.close()
```

```
import pickle
import gzip

f_inp = gzip.open('capitals.pgz', 'rb')
caps = pickle.load(f_inp)
f_inp.close()
```



For small amounts of data, the file might not be any smaller (it might even be slightly larger). Notice that the zip files are open as binary, that means we get byte streams from zip files.

If you see a built-in function called zip(), it has nothing to do with this. This function is for advanced combining of collections.

QA Other compression modules

Can compress/decompress in memory, or to/from a file

- zlib GNU zlib compression API
- bz2 bzip2 compression
- zipfile Zip archive access (pkzip compatible)
- tarfile TapeARchive (tar) access
- shutil High level archiving operations

All these
modules are in
the Python
standard library

```
import tarfile
import time

filename = 'qapyth3_linux.tgz'
if tarfile.is_tarfile(filename):
    tfo = tarfile.open(filename, 'r')

    for mdata in tfo.getmembers():
        tm = time.localtime(mdata.mtime)
        print("%-12s %s" % (mdata.name,
                             time.strftime("%d/%m/%Y %X", tm)))
    tfo.close()
```

The zlib module includes checksum functions (adler32 and crc32) are intended to check data integrity and are not cryptographically secure.

bz2 supports a file-like class called bz2.BZ2File, which is used in a similar way to the open() built-in.

The tarfile module supports the usual expected mechanisms, in addition getmembers() returns all the metadata for files in an archive, not just the file name. The example shown was run on Windows, using a tar file with -z compression which came from Linux.

A higher level (simpler) generic interface is provided by the shutil module. This is a general module which includes archiving operations.

QA

SUMMARY



A file object is created by calling open().

Read from a file:

- Call read, readline(), or readlines() methods.
- Or invoke the file iterator in a for loop.

Writing to a file:

- Call write or writelines() methods.
- The print() function can also be used.
- Good practice to close the file as soon as possible.

Many other methods available.

Objects can be preserved by pickling them.

Multiple objects can be preserved using shelves.

Compressing files saves disk space.

Q^ Database interface overview

Available for most popular databases:

- Database drivers are expected to conform to a standard.
- Import the required database driver, then call standard methods.
- Most drivers include extensions.

Two main objects:

- Connection object.
- Connect to the database.
- Create the cursor object.
- Transaction management.
- Cursor object.
- Execute queries on this.

Python is shipped with SQLite.

Python database device drivers are available for most popular relational databases and are likely to be bundled in with your release. They are written to a common standard - Python Database API Specification v2.0 (DB-API 2.0).

To use a database, we first have to connect with it, and that requires a connection object. That is then used to create a cursor, and for transaction management (commit and rollback).

The cursor object is the main workhorse, and a summary of the methods is shown below. Check the online documentation for details.

Cursor methods:

arraysize	Number of rows fetched by fetchmany
close	Close the cursor
description	Table meta-data
execute placeholders)	Execute an SQL statement (supports
executemany	Execute a sequence

fetchall	Get rows remaining
fetchmany	Get a number of rows
fetchone	Get next row
rowcount	Number of rows in the last operation

Python (like many languages) is shipped with SQLite (formally sqlite) which is a simple single file database.

Q Example - SQLite from Python

Use the `sqlite3` module.

- Bundled with the Python release.

```
import sqlite3

db = sqlite3.connect('whisky')

cur = db.cursor()

cur.execute('SELECT BRANDS.BNAME, REGION.RNAME      \
            FROM BRANDS,REGION      \
            WHERE REGION.REGION_ID = BRANDS.REGION_ID      \
            ORDER BY BRANDS.BNAME;')

for row in cur.fetchall():
    print("{0[0]}:{<30s} {0[1]}:{<30s}".format(row))

db.close()
```

Python uses cursors, which is traditionally the method used with embedded SQL in languages like COBOL and C. Neither PHP nor Perl support cursors at this time, but their use is being considered. The overall strategy for the SQL statement is not too dissimilar however.

Q A Binary files - struct.pack/unpack

Binary file formats don't map to Python variable types.

- Unless they were written using Python (like pickle).
- Typically, they might be written using C/C++, or similar.

Convert to/from primitive types using pack and unpack.

```
import struct
fin = open("bindata", "rb")
data = fin.read(1024)
fin.close()
clean = struct.unpack("iid80s", data)
print(clean)
txt = clean[3].decode().rstrip('\x00')
print(txt)
```

```
typedef struct {
    int a;
    int b;
    double c;
    char name[80];
} data;
```

```
(37, 42, 3.142, b'Hollow World!\x00\x00\x00\x00\x00...')
Hollow World!
```

The first problem when reading binary data is to determine the format. Unless the data was written by Python (or, for example, using the Python API from C), then the data will not be directly compatible with Python types. All high-level dynamic programming languages have this problem.

The example shown is of a simple C struct which has been written to a file. Even if you have never seen C before, it is obvious that the data consists of two integers (int), followed by a double precision number (double), followed by an array of 80 characters (that's a C string, and has a binary zero marking the end of the text). To convert this to Python, we can use unpack from the struct module (part of the Python standard library). This function takes as its first parameter, a string which describes the block of data. The example shows "iid80s" - 2 integers, followed by a double, followed by an 80-character string.

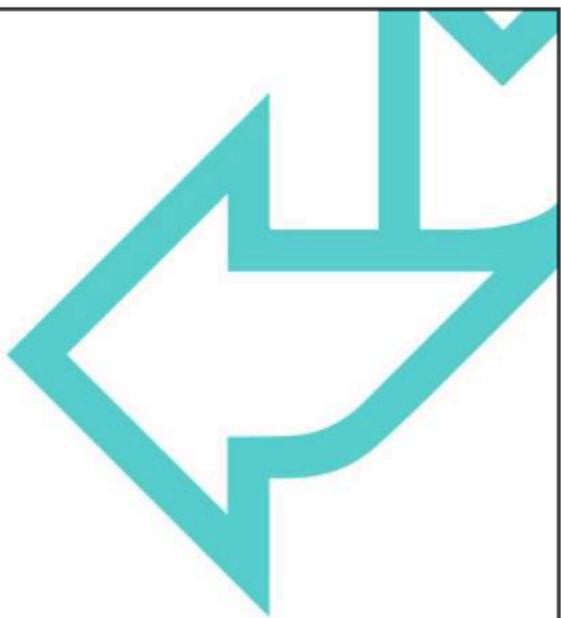
The C string is single byte, so it is converted to multi-byte using decode().

Note: When looking at C struct's be careful of padding bytes. By default, C compilers will align on item (often word) boundaries and might insert pad bytes. This will vary between 64-bit and 32-bit compilers and can even be adjusted (`#pragma pack`) by the program itself. You should also be aware of 'endianness' issues well described in the documentation for the Python struct module.



Python 3 Programming

Functions



QA

FUNCTIONS



Contents

- Python functions
- Function parameters
- Variadic functions
- Assigning default values
- Named parameters
- Annotations
- Returning objects
- Variables in functions
- Nested functions
- Function documentation
- Lambda functions

Summary

- Function attributes

QA Python functions

Functions are objects.

Defined with the def statement, followed by the argument list.

- Just like conditionals, membership is by *indentation*.

```
def make_list(val, times):  
    res = str(val) * times  
    return res
```

- Arguments are named and can have defaults.
- `return` statement is optional but recommended.
- Can optionally return any object type or `None`.
- Variables are local and safe if assigned, unless marked as global.

'Function names should be lowercase...' – PEP008

Python functions follow the same general syntax as conditional statements. The function is named, with an argument list terminated by a colon. Statements are part of the function by indentation. The argument list defines the variables passed with default values set. This is a little like awk, PHP, and some mainframe scripting languages, and with similarities to aspects of C++. The arguments supplied by the caller must match in number those required, otherwise the program will not compile. Arguments may be of any type, however.

QA Function parameters

Values required by the function:

- Specified within the parentheses of the function declaration.

```
def print_list(val, times):  
    print(str(val) * times)
```

- Literal parameters are passed by assignment (copy).

```
print_list(5, 3)  
print_list(0, 4)
```

- Variable parameters are passed by references, so changes alter the callers' variables.

```
def change_list(inlist, val, times):  
    inlist += str(val) * times  
  
mylist=[]  
change_list(mylist, 'h', 8)  
print(mylist)
```

['h', 'h', 'h', 'h', 'h', 'h', 'h', 'h']

*

Passing parameters by assignment means that the function gets a local reference to the object passed. If that local reference is changed by the function, it will have no effect outside the function.

However, passing variables means we are assigning a reference, so changing a parameter within a function will alter the caller's object when a variable (reference) is passed. To prevent this for lists and dictionaries, pass a slice of the entire structure:

```
print_thing(mylist[:])
```

Yes, it's a hack! Unfortunately, we get no indication that the function attempted to change the list and failed. Alternatively, we could use a tuple, which would produce a runtime error:

```
print_thing(tuple(mylist))
```

Generally, we prefer to return values from functions rather than alter parameters, since it is not always clear if a function is going to clobber your variable.

Q1 Assigning default values to parameters

Assign the default value when defining the function

- Parameters are now optional.
- Following parameters must have defaults as well.

```
def print_vat(gross, vatpc=17.5, message='Summary:'):
    net = gross/(1 + (vatpc/100))
    vat = gross - net
    print(message, 'Net: {0:5.2f} Vat: {1:5.2f}'.format(net, vat))

print_vat(9.55)      Summary: Net: 8.13 Vat: 1.42
```

- When calling a function, you can use named parameters instead.

```
print_vat(9.55, message='Final sum:')
                           Final sum: Net: 8.13 Vat: 1.42
```

In Python, a function can have parameters with default values.

For example, suppose a function calculates the VAT on an item. Most items in the UK were subject to a rate of 17.5%, which seemed reasonable as a default. However, a different rate might sometimes be used, so we do not want to hard-code it.

Parameters may only be assigned defaults on the right, if one is defaulted then all that follow must also be defaulted. So:

```
def print_vat2 (vatpc=17.5, gross):
```

...

Gives:

File "functions.py", line 31

```
def print_vat2 (vatpc=17.5, gross):
```

SyntaxError: non-default argument follows default argument

Q& A Passing parameters - review

```
def my_func(file, dir, user='root'):
    print('file: {}, dir: {}, to: {}'.
          format(file, dir, user))
```

- By position:

```
my_func('one', 'two', 'three')
```

The diagram shows the function call `my_func('one', 'two', 'three')` on the left, connected by a horizontal arrow to the function body on the right. The function body contains the line `print('file: {}, dir: {}, to: {}'.`. To the right of the arrow, the arguments are mapped to their corresponding parameter names: `file: one, dir: two, to: three`.

- By default:

```
my_func('one', 'two')
```

The diagram shows the function call `my_func('one', 'two')` on the left, connected by a horizontal arrow to the function body on the right. The function body contains the line `print('file: {}, dir: {}, to: {}'.`. To the right of the arrow, the arguments are mapped to their corresponding parameter names: `file: one, dir: two, to: root`.

- Or by name:

```
my_func(file='one', user='three', dir='two')
```

The diagram shows the function call `my_func(file='one', user='three', dir='two')` on the left, connected by a horizontal arrow to the function body on the right. The function body contains the line `print('file: {}, dir: {}, to: {}'.`. To the right of the arrow, the arguments are mapped to their corresponding parameter names: `file: one, dir: two, to: three`.

Before we look further at parameter passing, we should review the mechanisms we have seen so far.

Q& Enforcing named parameters

Use a bare * to force a user to supply named arguments.

- No need for a dictionary.

```
def print_vat(*, gross=0, vatpc=17.5, message='Summary'):  
    net = gross/(1 + (vatpc/100))  
    vat = gross - net  
    print(message, 'Net: {:.2f} Vat: {:.2f}'.format(net, vat))  
  
print_vat(vatpc=15, gross=9.55)  
print_vat()
```

Summary: Net: 8.30 Vat: 1.25
Summary: Net: 0.00 Vat: 0.00

Attempting to pass positional parameters will fail.

```
print_vat(15, 9.55)
```

TypeError: print_vat() takes exactly 0 positional arguments (2 given)

7

The example shows a technique to force the user to specify parameters by name rather than position. However, calling such a function with *no* parameters is perfectly legal, since (in this case) they all have defaults.

All parameters defined after (to the right) of the * must be named parameters, but those to the left may be positional. This syntax is Python 3 specific.

Q4 Unpacking and variadic functions

Functions usually have a fixed number of parameters.

- *Unpacking* passes a sequence's elements as single arguments.

```
def my_func(a, b, c):
    print(a, b, c)

mytup = 23, 45, 67
my_func(*mytup)
```

23 45 67

Variadic functions have a variable number of parameters.

- They can be collected into a tuple with a * prefix.

```
def my_func(dir, *files):
    print('dir:', dir, 'files:', files)

my_func('c:/stuff', 'f1.txt', 'f2.txt', 'f3.txt')

dir: 'c:/stuff', files: ('f1.txt', 'f2.txt', 'f3.txt')
```

Aside from defaults, Python functions have a fixed number of parameters. But what if we have the parameters in a tuple or list? We can pass them into a function by *unpacking*, by specifying a leading asterisk at the call. Note though that the container must contain the right number of elements.

We can pass any sequence, which is usually a tuple or a list, but it could be a string (unlikely) in which case each character would be seen as a different parameter.

Collecting parameters into a tuple within the function is known in the C/C++ world as a *variadic* function, that is it takes a variable number of parameters. The named parameter (in this case *files*) is the name of a tuple. Remember that tuples are immutable, but the individual elements may be references to objects that can be changed.

Q4 Keyword parameters

Named parameters just look like the key-value pairs of a dictionary.

- Because that is what they are.

Prefix a parameter with ****** to indicate a dictionary.

- Since a dictionary is unordered, then so are the parameters.
- May only come at the end of a parameter list.

```
def print_vat(**kwargs):
    print(kwargs)
print_vat(vatpc=15, gross=9.55, message='Summary')
{'gross': 9.55, 'message': 'Summary', 'vatpc': 15}
```

Use ****** to unpack caller's parameters from a dictionary.

```
argsdict = dict(vatpc=15, gross=9.55, message='Summary')
print_vat(**argsdict)
```

9

Keyword parameters (by convention called **kwargs**) enable the parameters to be passed as a dictionary.

Unpacking named parameters requires two asterisks as a prefix. The parameters are placed into a dictionary which, of course, is unordered. This means that the user does not need to get the order correct. This is particularly useful with a long parameter list.

A parameter prefixed ****** may only be used at the end of a parameter list, any other position will give a syntax error.

QA Returning objects from a function

Use a `return` statement, followed by the object to be returned.

- Any Python object may be returned.

Returning an object:

- Stops the execution of the function.
- Passes the object back to the caller.
- If `return` is not used, a reference to `None` is returned.

```
def calc_vat(gross, vatpc=17.5):
    net = gross/(1 + (vatpc/100))
    vat = gross - net
    return [f'{net:05.2f}', f'{vat:05.2f}']

result = calc_vat(42.30)
print(calc_vat(9.55))
```

['08.13', '01.42']

A function may return a reference to any type of object, including a list, a tuple, or a dictionary.

If a function does not return a value, and the calling code tries to use one, then `None` is passed. `None` is a "catch-all" object which means the lack of a value. It does not explicitly mean zero but can have the same effect under some circumstances. For example, `None` evaluates to `False` in a Boolean context.

QA Function annotations

Introduced in PEP 3107 for Python 3.0.

- Optional feature to add arbitrary comments to parameters and return types.
- Annotating parameters:

```
def my_func(dir:'str', files:'list'=[]):
```

- Annotating tuple and dictionary parameters:

```
def print_vat(**kwargs:'VAT, gross and message'):
```

- Annotating return types:

```
def calc_vat(gross:'float', vatpc:'float'=17.5)->'list':
```

- Accessible in special attribute __annotations__:

```
print(calc_vat.__annotations__)
```



Function annotations were introduced in Python 3 to add arbitrary metadata to arguments and return values. They are completely optional but can be useful for explaining the use of a parameter and return type by using an expression, e.g., 'str' or 'VAR, gross and message'. These expressions are evaluated at compile time and have no life or use in the runtime environment of the program. So, although similar in looks to type checking in other languages, they are strictly just explanations and cannot enforce type checking; remember Python is a dynamic language.

Python 2 did not support annotations, but several tools and libraries filled this gap using decorators (see PEP 318) or parsing the functions docstring for annotations.

Even in Python 3, annotations are effectively ignored and only useful with 3rd party libraries. For instance, one library might use annotations to perform type checking, or another might use them for improved help messages.

The annotations can be accessed using the special attribute:

function_name.__annotations__

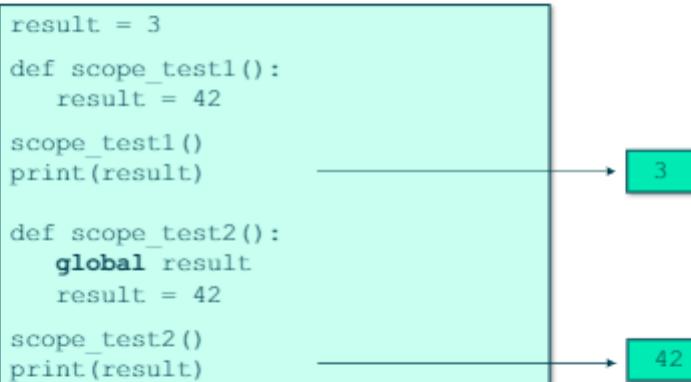
QA Variables in functions

Variables in a function can be local or global:

- Variables are defined as local if a value is assigned, otherwise global.
- Global variables can be referenced using the `global` keyword.
- Are local to the current module, or *namespace*.

```
result = 3
def scope_test1():
    result = 42
scope_test1()
print(result)

def scope_test2():
    global result
    result = 42
scope_test2()
print(result)
```



The rules of scope in Python are that an undefined variable used in a function must be a global, but if a value is assigned in the function before it is used then it is a local (unless it is prefixed with `global`).

Variables first assigned in a function are local to the function body, they disappear on exit from the function. In the example, we have two variables called `result` (bad coding, but it can happen). The variable inside the function bears no relationship with the one outside it. Assigning a different object to the local version does not affect the "outer" version in any way.

Occasionally, we might want to alter a global variable, in which case we must declare it as being global. The `global` keyword should be used before the variable is used. You can get away with declaring it as global later in the function, but you will get syntax warnings.

Global variables are generally frowned upon and are particularly problematic when an application is multi-threaded. Try to avoid

them.

QA Nested functions

A function can be defined within another function.

- This has the same scope as any other object.

```
def outer():
    num = 42

    def inner():
        print(num, "in inner")

    inner()
    print(num, "in outer")

outer()
inner()
```

42 in inner
42 in outer
NameError: name 'inner' is not defined

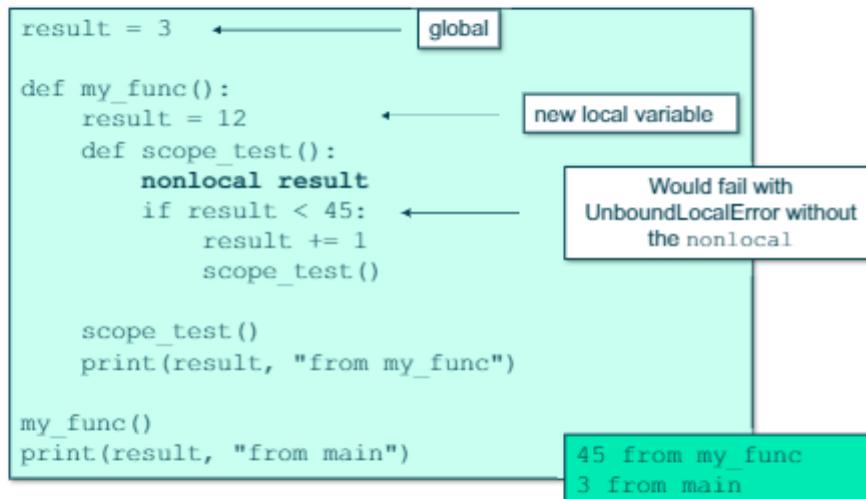
Nested functions can be used to encapsulate helper code.

- Can be returned as *closures*.

Placing a function inside another is not something which all languages support. It can be useful for simple functions, for so-called "closures", and for function factories. Notice that, in the example, the function has not declared num as global, yet the inner() function has access to it because it is still in scope. If an assignment was made to num in inner(), then a new variable would be created which would only be visible to inner().

Variables in nested functions

Since scopes may be nested, we need to indicate that:



Python's clean syntax and dynamic typing means that there is no mechanism for being able to define a new variable. This means that, on occasion, there can be confusion concerning exactly when a new variable should be created, and when an existing one should be used. We have already seen **global** as a statement which identifies a variable as being in global scope.

What about other enclosing scopes? Python is a little unusual in being able to define local, or nested, functions. If we wish to define a variable as being within an enclosing scope, then **nonlocal** is used.

In the example on the slide, if **nonlocal** was omitted then the `if` statement would be testing a variable before it was declared, since we have not defined `result` as global (in fact we don't want to, because that is not one we wish to use).

Defining as **nonlocal** means the variable `result` declared in the outer function will be used, and the recursion works successfully.

Of course, in this case, we could have passed the variable as a

parameter instead.

Note that if the inner function (`scope_test`) is returned then it may be called from the outside, still with the nonlocal variable in scope:

```
def my_func():
    ...
    return scope_test
rfunc = my_func()
rfunc()
```

This is known as a *closure*. In Python 2, it could only be achieved using globals.

QA Function documentation

Comments have limited use:

- Useful for maintainers, but not designed for users.

Python supports *docstrings*:

- Used by `help()` and for automated testing.
- Define a bare string at the start of the function.
- A triple quoted string, not an inline # comment.
- Or explicitly assign to the attribute `__doc__`

```
def my_func():
    """ my_func has no parameters
        and prints 'Hello'.
    """
    print("Hello")
```

>>> help(my_func)
Help on function my_func in module __main__:

my_func()
 my_func has no parameters and prints 'Hello'.

Use triple quotes over several lines

One reason why the online help in Python is so comprehensive is because it is built-in to the language - not added as an afterthought like some we could mention.

If we place a string at the start of a function (or a module), it is taken as documentation. Most people use a three-quoted multi-line string, but it can be any form of string.

By "start of the function" we mean that only white-space or comments are allowed between the function `def` statement and the docstring. Alternatively, we can assign the string to a special variable called `__doc__`.

QA Lambda functions

Anonymous short-hand functions:

- Cannot contain branches or loops.
- Can contain *conditional expressions*.
- Cannot have a `return` statement or assignments.
- Last result of the function is the returned value.

```
compare=lambda a, b: -1 if a < b else (+1 if a > b else 0)
x = 42
y = 3
print("a>b", compare(x, y))
```

Parameters

a>b 1

Often used with the `map()` and `filter()` built-ins:

- Applies an operation to each item in a list.

```
new_list = list(map(lambda a: a+1, source_list))
```

The term **lambda** comes from Lisp and is a little off-putting. Being Greek and sounding mathematical, it has the aura of complexity, but Python **lambda** functions are quite simple, for **lambda** read **inline**.

Lambda functions cannot really do anything that "normal" **def**'ed functions cannot do, they are just shorter and quicker to write. They are great for relatively small in-line functions that do not need a name - **lambda** returns a reference to the function, which may be stored in a variable. That variable is then used as if it was the name of the function.

Many Python built-ins take a "callable-object" as a parameter. This can be the name of a function, or the name of a reference to a function, or a **lambda** statement used without any names being involved. The example with **map()** adds one to each item in the list.

Q^ Lambda as a sort key

Customised Sort using a lambda function:

- Takes the element to be compared.
- Returns the key in the correct format.
- Sort each country by the second field, population.

```
countries = []
for line in open('country.txt'):
    countries.append(line.split(','))

countries.sort(key=lambda c: int(c[1]))

for line in countries:
    print(', '.join(line), end='')
```

Antarctica,0,--,Antarctica,1961,--,--
Arctica,0,--,Arctic Region,--,--,--
Pitcairn Islands,46,Adamstown,?,Oceania,--,...
Christmas Island,396,The Settlement,?,Oceania,....
Johnston Atoll,396,--,Oceania,--,US Dollar,--,...

"

Lambda functions are often used as parameters to built-ins. We could have defined a function to do the comparison, but instead we have defined a lambda function that takes one argument (c) and returns the key field in the correct format for comparison.

Q\Lambda Lambda in re.sub

The `re.sub` method can take a function as a replacement:

- Passes a match object to the function.
- The return value is the value substituted.

```
import re

numbers = ['zero', 'wun', 'two', 'tree', 'fower',
           'fife', 'six', 'seven', 'ait', 'niner']

alphas = ['alpha', 'bravo', 'charlie', 'delta', 'echo',
          'foxtrot', 'golf', 'hotel', 'india', 'juliet',
          'kilo', 'lima', 'mike', 'november', 'oscar', 'papa',
          'quebec', 'romeo', 'sierra', 'tango', 'uniform',
          'victor', 'whisky', 'xray', 'yankee', 'zulu']

codes = {str(i):name for i, name in enumerate(numbers)} ←
codes.update({name[0].upper():name for name in alphas})
reg = 'WG07 OKD'

result = re.sub(r'(\w)', lambda m: codes[m.groups()[0]]+' ', reg)
```

This is a dict comprehension

This code uses *dictionary comprehensions*, discussed in the next chapter.

The regular expression substitute operations (`re.sub` and `re.subn`) allow the second parameter to be a function (this is where the replacement string normally goes). The function can be a pre-defined named function, or a lambda.

In the example, we are constructing a dictionary. The keys come from a list of character ranges, and values are from the NATO phonetic alphabet. Care must be taken to ensure these lists are in the correct order.

The substitution matches any alphanumeric, and copies this to a match object by the use of a parentheses group `(\w)`. The single character is then used as a key to the dictionary.

The output in this case will be:

whisky golf zero seven oscar kilo delta

QA

SUMMARY



A function is a defined object.

- Variables have local scope, unless `global` is used.
- Other functions can be nested within.

Parameters are declared local variables.

- May be assigned defaults (from the right).
- `*arg` means unpack to a tuple.
- `**arg` means unpack to a dictionary.
- `*` forces the caller to use named parameters.

Can return any object:

- Including lists and dictionaries.

Can include a `docstring` - a bare string at the start.

Short, inline, anonymous functions can be defined using `lambda`.

QA More on keyword parameters

```
import sys

def my_func(**user_args):
    args = {'country':'England', 'town':'London',
            'currency':'Pound', 'language':'English'}
    diff = set(user_args.keys()) - set(args.keys())
    if diff:
        print('Invalid args:', tuple(diff), file=sys.stderr)
        return
    args.update(user_args)
    print(args)

my_dict = dict(town='Glasgow', country='Scotland')
my_func(**my_dict)
my_func(twn='Glasgow', county='Scotland')

{'town': 'Glasgow', 'currency': 'Pound',
     'language': 'English', 'country': 'Scotland'}
Invalid args: ('county', 'twn')
```

20

The first line of the output has been wrapped around to fit.

QA Function attributes

py3

<code>__annotations__</code>	Parameter comments
<code>__closure__</code>	A tuple containing bindings for the function's free variables.
<code>__code__</code>	Code object representing the compiled function body.
<code>__defaults__</code>	A tuple containing default argument values for those arguments that have default values.
<code>__dict__</code>	The namespace supporting arbitrary function attributes.
<code>__doc__</code>	The <i>docstring</i> defined in the function's source code.
<code>__globals__</code>	Reference to the global namespace of the module in which the function was defined.
<code>__kwdefaults__</code>	Dictionary containing defaults for keyword-only parameters
<code>__module__</code>	The name of the module the function came from
<code>__name__</code>	The function's name
<code>__qualname__</code>	The function's qualified name (where it was defined) 3.3

21

In Python 2, most of these were prefixed `func_`, but `__annotations__`, `__kwdefaults__`, and `__qualname__` are Python 3 specific.

Use them like this:

```
def myfunc():
    print (myfunc.__name__)
```

You might think there is little point in getting the name if you already need it, but it is used more with references to functions. To give a simple example:

```
fref = myfunc
...
print (fref.__name__)
```

The attribute `__qualname__` (introduced at 3.3) will give the original name and where it was defined.

QA Function annotation

Similar to inline comments

- Not supported in lambda functions.

```
def print_vat (*,
    gross:"Gross amount (including VAT)"=0,
    vatpc:"VAT in percentage terms"=17.5,
    message:"Free text"='Summary:') \
-> "No usable return value":
```

Function attribute `__annotations__` gives details.

```
for kv in print_vat.__annotations__.items():
    print(kv)
    ('gross', 'Gross amount (including VAT)')
    ('message', 'Free text')
    ('vatpc', 'VAT in percentage terms')
    ('return', 'No usable return value')
```

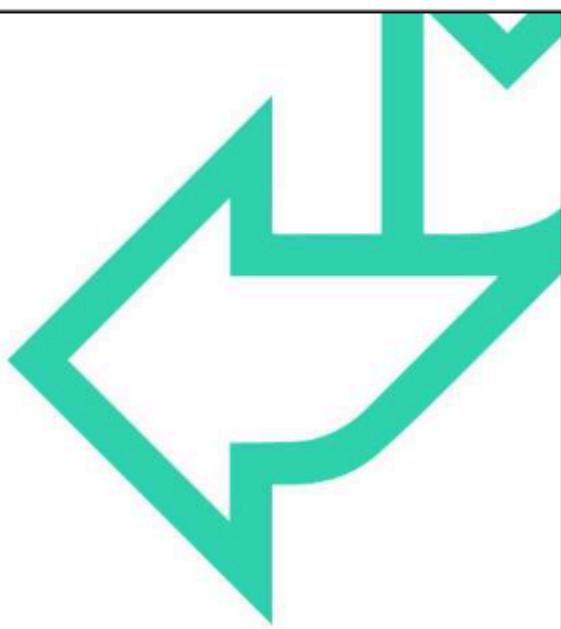
xx

Annotations allow meta-data to be attached to parameters and return values. Python itself ignores them, as it would comments, but they are available for use by third-party libraries. Aside from the basic syntax, there are no standard data types or formats - these are left flexible for third-party libraries to exploit.



Python 3 Programming

Advanced collections





ADVANCED COLLECTIONS



Contents

- Advanced functions - filter
- List comprehensions
- Set and dictionary comprehensions
- Lazy lists
- Generators
- Copying collections

Summary

- Generator objects
- Co-routines and send()
- Generator delegation

QA Advanced functions - filter

Syntax: `filter(function, iterable-object)`

- Returns an iterator for every item where function returns true.
- The function could be named, or a lambda.
- The iterator can be used in a loop:

```
import glob
import os

pattern = 'C:/QA/Python/*'
for fname in (filter(os.path.isdir, glob.iglob(pattern))):
    print(fname)
```

Print a list of directories

- Or we can construct a list:

```
dirs = list(filter(os.path.isdir, glob.iglob(pattern)))
```

Get a list of directories

The *function* in **filter** can be the name of a function, or a **lambda** function. Here we have used `os.path.isdir` which returns true for each item that is a directory. The first example prints each filename that is a directory. `glob.iglob` does an expansion of the glob (wildcard) pattern and returns an iterator. The second example constructs a list of directory names.

The *list* in **filter** can be any iterable object, that is a list, tuple, string, or even a file.

As a special case, if the function is **None** then the items returned are those that resolve to true (rather like `grep`).

In Python 2 **filter** returned a list, in **Python 3** it returns an iterator.

QA List comprehensions

A list comprehension returns a list.

It consists of:

- A loop - typically a `for` loop.
- An `expression` which identifies a list item.
- An optional `condition` to filter items.

Example, get a list of file size:

```
pattern = 'C:/QA/Python/*'  
sizes = [os.path.getsize(fname) for fname in glob.iglob(pattern)]
```

Example, get a list of directories:

```
dirs = [fname for fname in glob.iglob(pattern) if os.path.isdir(fname)]
```

Pythonic replacement of the `filter()` built-in.

A list comprehension is often a more natural way of expressing list operations. The original idea comes from set theory and became popular in the Haskell language.

There are three parts to a list comprehension: the expression which will be executed for each item, the loop (usually a `for` loop), and an optional condition. The item will only be returned when the condition is true.

List comprehensions fit naturally into a world of functions which, increasingly, return iterators rather than lists.

The `map` and `filter` built-ins are their functional equivalents, but list comprehensions are more *Pythonic*.

PY3: In Python 3, the syntax was tightened to require a single (iterable) expression after the `in`. Hence, this worked in Python 2:

```
a = [i for i in 1,2,3]
```

but that gives a syntax error in Python 3. The correct notation in Python 3 (which also works in 2) is:

```
a = [i for i in (1,2,3)]
```

In Python 3, the loop variable is within its own scope, it will not affect a variable of the same name outside the comprehension (which is not the case in Python 2).

Q4 Set and dictionary comprehensions

Python 3 allows comprehensions on sets...

```
myset = {'booboo', 'yogi', 'care', 'fozzie'}  
results = {do_ftp(m) for m in myset}
```

... and dictionaries.

```
pattern = 'C:/*.py'  
tsizes = [(fname, os.path.getsize(fname))  
          for fname in glob.iglob(pattern)]  
  
[(('C:/first.py', 90), ('C:/think.py', 0), ('C:/try.py', 21))]
```

ftp to 'fozzie'
ftp to 'yogi'
ftp to 'booboo'
ftp to 'care'

```
dsizes = {fname:size for fname, size in tsizes  
          if size > 0}  
{'C:/first.py': 90, 'C:/try.py': 21}
```

← List of Tuples

← Dictionary

In addition to list comprehensions, Python 3 also supports set and dictionary comprehensions.

A set comprehension takes a set and iterates through it to produce another set. In the example, a user written function called `do_ftp()` (which probably uses the standard module `ftplib`) carries out its task on the four machines, in an unknown order. The `results` set will hold whatever the function returns on each occasion.

Dictionary comprehensions are twice as complex, in that they need a key, value tuple to feed. In the example, a list of two item tuples is created first, using a list comprehension. This is then used to construct our dictionary, which contains a further modifier (`if size > 0`).

This example *could* be done in one statement:

```
sizes={fn:os.path.getsize(fn) for fn in glob.iglob(pn)  
      if os.path.getsize(fn) > 0}
```

The variable `fname` has been shortened to `fn`, and pattern shortened to `pn` (to fit).

QA Lazy lists

- Generating lists in memory can be an overhead.
 - How big is a list?
 - What about sequences that have no end?
- Lazy lists only return a value when it is needed.
 - One item at a time, as and when required.
- Particularly suitable when iterators are used.
 - An iterator function returns items one at a time.
- Many Python 3 functions return iterators, rather than lists.
 - map(), filter(), range(), reversed(), zip(), and so on.

Historically, many functions used or generated in-memory lists for iteration. This was fine for relatively small data sets, but when the list becomes large then the overhead is unacceptable. Enter the lazy list.

A lazy list is when the next item is supplied when needed, and not before. If we stop processing before reaching the end of the list, then those unused items will never be generated or use up resources. Lazy lists are implemented by supplying an iterator, i.e., position, to the list, rather than the whole list itself.

Q& Generators

Generator functions are a special kind of function that return a lazy iterator.

- These are objects that you can loop over like a list.
- Unlike lists, lazy iterators do not store their contents in memory.

Example 1 – Reading large files

- Used to read through large files a row at a time where the file would be too big for memory.

```
def csv_reader(file_name):
    for row in(file_name, "r"):
        yield row
Row count is 64186394
```

Q& Generators

Example 2 – Generating an infinite sequence

- Used to read through large files a row at a time where the file would be too big for memory.

```
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1
```

- If this is run, the program will never end and will need to be cancelled by a keyboard interrupt .

```
for i in infinite_sequence():
    print(i, end=" ")
```

- Creating a generator means you can create resources as needed (excellent for data analysis).

```
gen = infinite_sequence()
print(next(gen))
```

QA Generators

A lazy list item is returned at the **yield** statement.

```
def get_dir(path):
    pattern = os.path.join(path, '*')
    for file in glob.iglob(pattern):
        if os.path.isdir(file):
            yield file
```

- Generators can often replace list comprehensions.
- Can be used anywhere an iterator is expected.

```
for dir in get_dir('C:/QA/Python'):
    print(dir) ← Print a list of directories
```

```
dirs = list(get_dir('C:/QA/Python')) ← Get a list of directories
```

A generator can be used to perform a lazy evaluation, often replacing list comprehensions.

In each example on the slide, the function is entered just once. The **yield** statement temporarily suspends operation of the loop within the function and returns an intermediate value which supplies the next item in the list.

Executing a **return** from within a loop containing a **yield** expression will give a `SyntaxError`, you should **break** out of the loop instead.

Contrast this with the code on the slides for filters and list comprehensions. In those structures, a temporary result list was created in memory. With generators and lazy lists, only the result immediately being processed is held.

Note: The following imports have been omitted for clarity:

```
import os.path
import glob
```

QA List comprehensions as generators

A list comprehension may be used instead of `yield`.

- Sometimes - this does not support sending values.
- Enclose the comprehension in `()` instead of `[]`

```
def get_dir(path):
    pattern = os.path.join(path, '*')
    for file in glob.iglob(pattern):
        if os.path.isdir(file):
            yield file
```

- Rewritten as a list comprehension:
 - Function returns a generator object, as before.

```
def get_dir(path):
    pattern = os.path.join(path, '*')
    return [file
            for file in glob.iglob(pattern)
            if os.path.isdir(file)]
```



In Python 3*, we have an alternative syntax to `yield`, and that is to use a list comprehension instead. The syntactical difference is the use of rounded brackets (parentheses) instead of squared brackets.

The slide shows an earlier example of `yield`, with the list comprehension equivalent. The two code fragments are functionally the same, both return a generator object. An obvious downside is that there is no way for the caller to return a value to the generator function, as there is with `yield`.

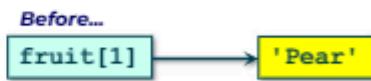
* It was actually introduced into 2.6.

QA Copying collections - problem

Any problems with assignments?

- Remember that Python objects are references:

```
fruit = ['Apple', 'Pear', 'Orange']
```

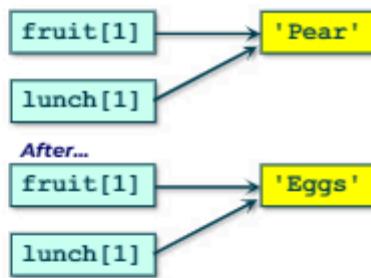


- Copy by reference:

```
lunch = fruit
```

Oops...

```
lunch[1] = 'Eggs'  
print(fruit)  
['Apple', 'Eggs', 'Orange']
```



Since Python uses reference objects, copying one container (collection) to another can give unexpected results. An assignment copies the references, not the data to which they are referring. All is well with *immutable* objects, such as string literals, integers or tuples, but *mutable* objects (by definition) can be altered, and that can have unexpected effects, as shown.

During an assignment of one collection to another, only a shallow copy is done (we are using lists here, but this applies to any collection object).

Don't panic, there is a simple solution...

Q& Copying collections - slice solution?

For a sequence, take a slice:

```
fruit = ['Apple', 'Pear', 'Orange']
lunch = fruit[:]
lunch[1] = 'Eggs'
print('fruit:', fruit, '\nlunch:', lunch)
```

```
fruit: ['Apple', 'Pear', 'Orange']
lunch: ['Apple', 'Eggs', 'Orange']
```

We need a better solution for more complex structures.

- A slice is still a shallow copy.

```
fruit = ['knife', 'plate', ['Apple', 'Pear', 'Orange']]
lunch = fruit[:]
lunch[2][1] = 'Eggs'
print('fruit:', fruit, '\nlunch:', lunch)
```

```
fruit: ['knife', 'plate', ['Apple', 'Eggs', 'Orange']]
lunch: ['knife', 'plate', ['Apple', 'Eggs', 'Orange']]
```



For a simple sequence, as we saw in the previous example, the solution is easy to implement - use a slice. The rather strange syntax of an empty slice means to take all the elements. A slice is always an independent copy of the original.

Unfortunately, this is not good enough when we have a more complex data structure - even though we don't have to get too complex for the slice solution to fail.

In the second example, the assignment of `fruit[:]` to `lunch` will make an independent copy of `fruit`, but only the reference to the nested list will be copied, not the nested list itself.

Q^Copying collections - deepcopy solution

A better solution for more complex structures:

- The `copy` module, distributed with Python.
- Can do a shallow copy or a deep-copy.

```
import copy

fruit = ['knife', 'plate', ['Apple', 'Pear', 'Orange']]
lunch = copy.deepcopy(fruit)
lunch[2][1] = 'Eggs'
print('fruit:', fruit, '\nlunch:', lunch)
```

```
fruit: ['knife', 'plate', ['Apple', 'Pear', 'Orange']]
lunch: ['knife', 'plate', ['Apple', 'Eggs', 'Orange']]
```

Beware! "copy" usually means a shallow copy.

To solve the problem of nested collections we need to do a deep copy, and a standard module, called `copy`, is provided for that. There are two methods exposed, `copy` (which does a shallow copy) and `deepcopy`.

Not all objects can be copied using the `copy` module, those include other modules, class objects, functions and methods, files, and so on.

When we use the term "copy" we usually mean a shallow-copy, so be careful!

QA

SUMMARY



filter() returns items that are true.

- Maybe with the help of a lambda.

List comprehensions replace filter() and map().

- Possibly with the help of a lambda.

Generators yield values as they are needed.

Generators can replace list comprehensions.

Copying collections might not be a simple assignment.

- A deep copy might be required.

QA Generator objects and next

A generator function returns a generator object.

- Can be used when a 'for' loop is not appropriate:

```
gen = get_dir('C:/QA/Python')
```

*Using the generator function
get_dir from earlier*

- The next built-in gets the next item from a generator:

```
while True:  
    name = next(gen, False)  
    if name: print(name)  
    else: break
```

C:/QA/Python\Appendicies
C:/QA/Python\bak

- A loop does not have to be used:

```
gen = get_dir('C:/QA/Python')  
dir1 = next(gen, False)  
dir2 = next(gen, False)  
dir3 = next(gen, False)
```

Generator objects can be created from a generator function call.
Each time a generator function is called the iteration is restarted.

If used outside a loop, the **next** built-in will get the next yield item.
Alternatively, call the `__next__` method on the generator object,
for example `gen.__next__()` (this is done by the `for` loop).

The optional second parameter to `next()` gives the value returned
when the generator sequence ends.

QA Co-routines and send() method

Data can be returned to the generator using send.

```
import glob
import os
import os.path

def get_dir(path):
    while True:
        pattern = os.path.join(path, '*')
        path = None
        for file in glob.iglob(pattern):
            if os.path.isdir(file):
                path = yield file
                if path: break

        if not path: break
    return

gen = get_dir('C:/QA/Python')

print(next(gen))
print(next(gen))
print(gen.send('C:/MinGW'))
print(next(gen))
```

Both next() and
gen.send() get the
next yielded value

```
C:/QA/Python\AdvancedPython
C:/QA/Python\Appendicies
C:/MinGW\bin
C:/MinGW\dist
```

The send() generator method can be used instead of next(), the difference is that send() also sends a value back to the generator function, which can be picked-up as the return value from yield().

When the send() method is not used, yield returns None.

These expressions were introduced at Python 2.5 and are called coroutines. They are described in PEP 342 - *Coroutines via Enhanced Generators*.

In the example on the slide, the generator function (get_dir()) searches for sub-directories in the given directory. After reporting the first two, it is sent, as the return value from yield(), a different directory to search.

Q& Generator delegation (3.3)

Generator delegation allows a large and complex generator to be decomposed into sub-generators.

- In the same way as a large and complex function might be split into smaller components.

Simplistically:

- `yield from iterable`:

```
for file in glob.iglob(pattern):  
    yield file
```

- Can be written as:

```
yield from glob.iglob(pattern)
```

Generators, like any other code, can get unwieldy. The `yield from` syntax introduced at 3.3 allows us to delegate to a sub-generator, which means we can split-up (decompose) complex code.

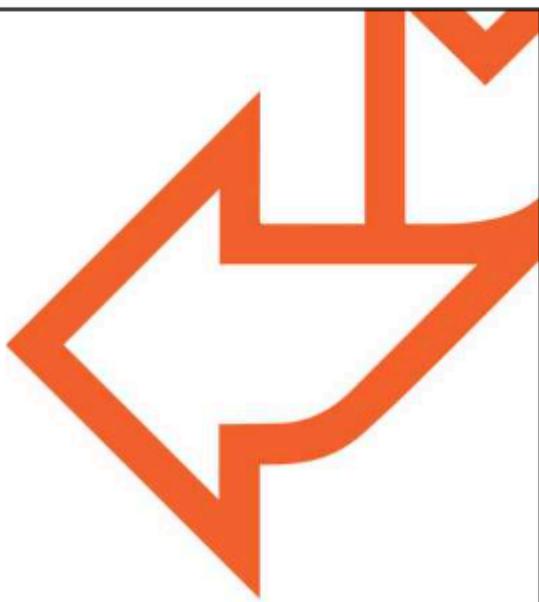
Most useful in closures and sub-generators (generators called from other generators).

See PEP 380 for further details and examples.



Python 3 Programming

Modules and packages



QA

MODULES AND PACKAGES



Contents

- What are modules and packages?
- How does Python find a module?
- Multiple source files
- Importing a module
- Importing names
- Directories as packages
- Writing a module

Summary

- Distributing a module - distutils

2

We have already met modules that are bundled with Python, now we shall discuss writing our own.

QA What are modules?

A module is a file containing code:

- Usually, but not exclusively, written in Python.
- Usually with a .py filename suffix (some modules are built-in).

A module might be byte-code:

- Python will create a .pyc file if none exists.
- Held in subdirectory `__pycache__` from Python 3.2.
- Python will overwrite this if the .py file is younger.

A module might be a DLL or shared object:

- With a .pyd filename suffix.
- Often written in C as a Python extension.

'Modules should have short, all-lowercase names.'

PEP008

The modules we have seen so far have been bundled with Python, and we have used them as just another part of the language. The built-ins are not physically separate modules, although we have a logical view of them as being just like any other. Those aside, all modules are represented by separate files, which are logically independent from the main program.

Rather than compile a module each time it is loaded, Python compiles once and dumps the byte-code into another file. From Python 3.2, these files are held in a sub-directory called `__pycache__`.

Byte-code files (.pyc) are not necessarily portable, either across platforms or between releases. The files themselves contain a 'magic number' which indicates the Python release they were built for, and incompatible byte-code files will be recreated. It is therefore important that a bundled package includes the source code (.py files) as well as the byte-code files if required. This has led to issues where multiple versions of Python are regularly used, in that Python would be continually recompiling. From Python 3.2, the version of Python is also included in the filename in `__pycache__`, for example: `abc.cpython-32.pyc`.

Modules written as C extensions are generally created as a DLL (on Windows) or shared object (Linux/UNIX .so files). These have the file extension .pyd, but are otherwise identical to a native binary.

Prior to Python 3.5 we also had .pyo files. The .pyo files contained optimised byte-code and were created using the -O option to the python command-line. See PEP 488.

Q& A What are packages?

A package is a logical group of modules.

A directory containing a set of modules is a package.

The difference is a file called `__init__.py`

- Often empty.
- Can contain initialisation code.
- Can even contain functions.
- Can contain a list of the public interfaces as attribute `__all__`.
- These are the names imported with `from Module import *`.

```
# Public interface
__all__ = ['getprocs', 'getprocsall', 'filter']
```

- See *Namespace packages* later...

In Python, a module is the file itself, and a package is a group of modules in a directory (or folder, if you prefer). The directory itself is the package - provided it has a file called `__init__.py` in it.*

This file is often empty, or maybe just has a comment in it. It can also have a huge amount of code in it, depending on the whim of the author. One of the more useful attributes which can optional be set is `__all__`, which gives a list of the public elements of the package.

* At least, that is the situation in Python 2, and up to Python 3.2. From Python 3.3, we have *Namespace packages* that don't have this file. Namespace packages are discussed later...

Q&A Multiple source files - why bother?

Increase maintainability:

- Independent modules can be understood easily.

Functional decomposition:

- Simplify the implementation.

Encapsulation & information hiding:

- Easier re-use of modules in a different program.
- Easier to change module without affecting the entire program.

Support concurrent development:

- Multiple people working simultaneously.
- Debug separately in discrete units.

Promote reuse:

- Logical variable and function names can safely be reused.
- Use or adapt available standard modules.

A Python module is somewhat like a separate source file or DLL in C or C++.

However, it is more than that:

Variables can be local to the module

Packages have an independent namespace

For OO-programming, each module can implement a single class

The term *package* is used to indicate a collection of modules, on a local disk or stored on the network. Specifically in Python, it is the directory that the modules reside in.

The reasons for splitting-up an application into modules are all based on good structured programming techniques; however, the overriding reason is code reuse - why reinvent when someone else has already written the code.

Q&How does Python find a module?

The initial path is from `sys.path`

- May be modified using `sys.path.append(dirname)`.
- Starts with the directory from which the main program was loaded.

```
import sys
sys.path.append('./demomodules')
import mymodule
print(sys.path)
['c:\\QA\\Python\\MyDemos', 'c:\\Python30\\Lib', ...
 ./demomodules]
```

Or change environment variable `PYTHONPATH`:

- Contains a list of directories to be searched.
- Separator is the same as your system's PATH - : for *NIX and ; for Windows.

The directories searched for Python modules will vary depending on the platform and installation, but always includes the current directory. Windows also has C:\Python3n\Lib\site-packages. To find the path used just print `sys.path`.

To add a directory to the path, either use `sys.path.append`, or the environment variable `PYTHONPATH`.

Note that either directory separator (/ or \) may be used on Windows.

QA Importing a module

Surprisingly, use the `import` command:

- At the top of your program, by convention:

```
import mymodule  
print (mymodule.attribute)
```

Case sensitive, even on Windows

- Can specify a comma-separated list of module names:

```
import mymodule_a, mymodule_b, mymodule_c
```

- Can specify an alias for a module name:

```
import mymodule_win32 as mymodule  
print (mymodule.attribute)
```

- Trouble is, you have to specify the module name for each call.

We have seen the basics of importing modules already - after all, you can't do much in a Python program without using `import`. Here are a few more details.

Notice that the case of the module name must match that of the file name. By default, this also applies to Microsoft Windows, unless the environment variable `PYTHONCASEOK` is set. We can specify an alias if required, and that is a commonly used feature.

Modules already loaded can be reloaded using (surprise) `imp.reload`. This may be useful if you are creating the modules programmatically.

QA Importing names

Alternatively, import the names into your namespace.

- Beware! Risk of name collisions!

```
from mymodule import *
```

Specify specific object name(s):

```
from mymodule import my_func1  
...  
my_func1()
```

How do we know which module my_func1 came from?

Or use an alias:

```
from mymodule import \  
    (my_func1 as mf1, my_func2 as mf2)  
  
mf1()  
mf2()
```

Better or worse?

Specifying the name of the module for each function call can be tedious, so we can import all the external names on the module into our own namespace. The problem is that this can lead to ‘Namespace pollution’, so instead, we can specify exactly which names to import.

If those names clash with existing names within the program (*name collisions*), then we can assign aliases to individual names. However, it can then get very difficult to track back which names belong to which module, so choose your aliases carefully!

Note that we can only import public names, that is those not prefixed with an underscore, or those specified in __all__.

Q& Directories as packages

Keep related modules together in the same directory.

- The name should not be the same as a Python system directory.

An `__init__.py` file is required

- Might be empty.

```
import workingmodules.mymodule_a ← Directory name/package name  
workingmodules.mymodule_a.myfunc1()
```

May be nested

- Each nested sub-directory should have a `__init__.py` file.
- Each is just another name in the hierarchy.
- Import relative to the current package using `.module`
- Import relative to the parent using `..module`

Important to know that we used to need an `__init__.py` but that it is no longer mandatory.

Q^A Namespace packages (3.3)

From Python 3.3 `__init__.py` is no longer mandatory

- A directory without `__init__.py` is a *Namespace package*.
- A directory with `__init__.py` is a *Regular package*.

Advantages:

- We no longer need to supply an empty `__init__.py`
- Namespaces can now span directories

```
sys.path.append('./date_packages')
sys.path.append('./person_packages')
from mynames.date import Date
from mynames.person import Person
```

Where both directories
have a sub-directory
named `mynames`

Disadvantages:

- No initialisation code.
- No `__name__` attribute for the namespace.

If all you want a package for is to logically group modules together, then the traditional package mechanism (*Regular packages*) is rather inflexible. The initialisation performed by `__init__.py` can be very powerful, but it is not always appropriate.

Enter *Namespace packages* at Python 3.3. These allow a subdirectory to be the namespace, but that same subdirectory name can occur in any number of other parent directories. It is the subdirectory name that is used for the Namespace. See PEP0420 for further details.

Q& Writing a module

No special header or footer required in the file.

- Just write your code without a 'main'.
- Default documentation is generated and available through help().

Conventions with underscores – reminder.

- Names beginning with one underscore are private to a module.
- Includes function names.
- Names beginning and ending with two underscores have a special meaning.

Name of the module is available in `__name__`

```
def my_func1():
    print("Hello from", __name__)
```

A module in Python required no special header or delimiter in the file - any Python code file can be a module. The module can be without a "main", or a #! line and execute access (on UNIX) but see later when we discuss testing.

A single underscore prefix means that the name is not exported from a module, unless `__all__` is specified in the package `__init__.py` file, in which case only those names will be exported. Names with two leading underscores are mangled, and so localised.

QA The 'main' trick

Code outside of a function is executed at import time.

- That is undesirable if our module could be run as a program.

Fortunately, we can test the name of the module.

- Will be `__main__` if run as a program.

```
def main():
    """
    Stand-alone program code,
    usually function calls or tests
    """

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

Now our code can be
run as a module or a
stand-alone program

- Using a function called `main()` is not mandatory, but common practice.

It is not uncommon to develop a Python script and then realise that the majority of it would be useful as a module. Of course, that requires that we have written it by breaking down the functionality into callable functions, which good programmers will do naturally anyway. In a full program, there is always the need for a 'main', which might do nothing more than call functions in the correct order, but that would get run at import time if we tried to run our program as a module.

We could just remove the 'main' code, but that would make life more difficult if we wanted to have the choice of running it as a module or a program. So, we can use trickery by testing the module name. We don't know what you will choose as your module name, but it won't/can't be `__main__`. That is the name used for the main module when running a program. So, we can test the module attribute `__name__` and choose which code to execute.

It is common practice to use a function called `main`, since that gives us the opportunity to have scoped variables (remember that in Python a conditional statement is not a unit of scope).

Some advocate *always* writing Python code in this way, including stand-alone programs, for maximum flexibility.

QA The 'main' trick

When we use the attribute `__name__` in the file that's being run:

- It will return the name `__main__`

```
print(__name__)
      main
```

Place the `print(__name__)` into a function and import into another file:

- Attribute `__name__` will return the name of the file it was imported from.
- NOT the name `__main__`

```
from mainTest import importThis
importThis()
      mainTest
```

To checks to see if the python file is being run as a script and not a module:

- Use If `__name__ == "__main__"`

A script is what we call a python file that is run from a command line. The three main methods of running python are from the command line, as a file in an IDE or as the functionality being imported and run in another file.

QA Module documentation

Docstring for the module must be at the (very) start.

- Or explicitly assigned to `__doc__`
- Used by the `pydoc` utility to generate documentation files.
- A default help format is provided.

```
>>> help(mymodule_a)
Help on module mymodule_a:

NAME
    mymodule_a

FILE
    c:\qa\python\mydemos\demomodules\mymodule_a.py

DESCRIPTION
    This is a test module containing one
    function, my_func1

FUNCTIONS
    my_func1()
        my_func1 has no parameters and prints 'Hello'

DATA
    var1 = 42
```

The diagram shows two arrows originating from the text "Module docstring" and "Function docstring". The arrow labeled "Module docstring" points to the "DESCRIPTION" section of the help output. The arrow labeled "Function docstring" points to the "FUNCTIONS" section, specifically to the description of the `my_func1()` function.

Like functions, modules can contain docstrings, and these will be used as the documentation when `help()` is called. The format of the docstring is the same as for functions, and must occur at the very start of the module, even before any imports. If not at the start of the module, it can be assigned to the variable `__doc__`, for example:

```
__doc__ = """
    This is a sample module which
    does various date operations.
    """
```

Documentation in forms other than `help()` can be generated, using the `lib/pydoc.py` program (bundled with python). Documentation in HTML and UNIX man page format can be created, as well as searched using a small browser.

This is very useful on its own, but docstrings have other hidden magic...

QA Python profiler

The **cProfile** module

- Profile a specific function from a script.

```
import mymodule
import cProfile
cProfile.run('mymodule.start()', 'start.prof')
```

Save statistics to this file (optional)
Default: display statistics to stdout

- Or the whole script from the command-line.

```
C:\QA>python -m cProfile thing.py
```

- Or analyse the output file using **pstats** shell.

```
C:\QA>python -m pstats start.prof
Welcome to the profile statistics browser.
% help
```

A profiler is useful to understand the way your code behaves. With simple scripts, it will not tell you much, it is really useful with large and complex applications and modules.

The module **cProfile** is bundled with the standard Python release from version 2.5.

There is also a pure Python profiler called **profile**, but it is slower than **cProfile** and has the same functionality. In its simplest form, the output generated will tell you how many calls each of your functions and methods received, and how long was spent executing each function. The functions are identified by their line number and file name.

The module, and the analysis of the data, can be used in several ways. By using **cProfile** from a program it is possible to profile specific parts of the code - particularly useful when unit testing. The default output goes to **stdout** but can be saved in a binary format for later analysis by **pstats**.

The **pstats** module can also be run in a number of ways. It can be run as a shell, as in the example, or programmatically. The program interface is best suited for analysing multiple statistics files in a complex environment and is documented in the Python documentation.

See also the **trace** module in the Python standard library.

QA

SUMMARY



- **Writing a module in Python is simple.**
 - Just a bunch of code in a file.
- **Python loads modules based on sys.path.**
- **Import a module using import.**
 - Can also specify importing names into our namespace.
- **Directories can be packages.**
 - Require the `__init__.py` file.
- **There are several features and base modules to assist testing.**

Q4 Distributing libraries - **distutils**

Enables programs, modules, and packages to be bundled and unbundled in a standard way.

- Part of the standard library.

Based on **setup.py** written by the distributor (see over).

Creating a distribution:

- Compressed file is placed into sub-directory ./dist

```
C:\product> python setup.py sdist
```

Installing a distribution:

```
C:\product\dist> unzip product-1.0.zip
C:\product\dist> cd product-1.0
C:\product\dist> python setup.py install
```

The **distutils** module is designed to provide a uniform interface for users to create, distribute, and install modules and associated files. From the user's viewpoint, it is based around a script normally called **setup.py**, which is described on the next page. The distribution is usually in the form of a zip file or a gzip tarball, depending on the platform, created in a sub-directory called **dist**.

For pure python modules, the **sdist** argument for **setup.py** should be sufficient, **sdist** means source distribution. If the distribution includes binary files, such as executables or other platform specific files, then it should be **bdist**. A binary distribution will include the .pyc files for the modules.

The argument **bdist_winst** will produce an .exe file which the user has to just double-click on to invoke the Windows installer. This also registers the module, so it can be uninstalled using the control panel. Beware that the generated .exe file can be architecture specific, so that a 64-bit .exe file will not run on a 32-bit Windows installation.

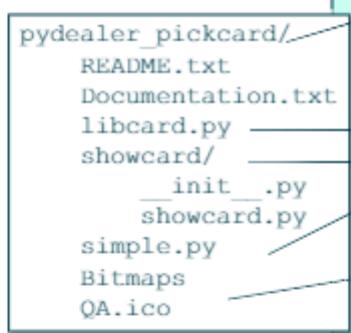
QA Distributing libraries - distutils

There is a standard way of organising your files:

- Described in setup.py

```
from distutils.core import setup
from glob import glob

setup(
    name = "pydealer_pickcard",
    version = "1.0",
    author = "QA",
    author_email = "QA.com",
    py_modules = ['libcard'],
    packages = ['showcard'],
    scripts = ['simple.py'],
    data_files = [
        ('Bitmaps',glob('Bitmaps/*')),
        ('.', ['qa.ico']),
    ]
)
```



When generating a distribution, the first thing to do is to organise your files in the standard way. If you don't like the standard layout, then you can specify a different one in **setup.py**, but that is not worth the effort unless you have a very good reason. The top-level directory does not have to be the name of the distribution, but it would be confusing if it was not.

The next step is to write **setup.py**. The example shown does not include all possible combinations but covers many that are optional. The absolute minimum **setup.py** for a single module is:

```
from distutils.core import setup

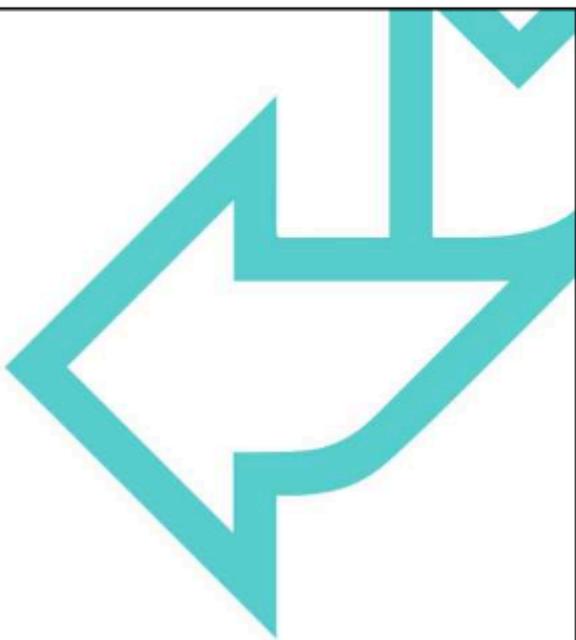
setup(
    name = "modulename",
    py_modules = ['modulename'],
)
```

The default version number is 0.0.0, so it is probably best to set a version number as well.



Python 3 Programming

Classes and object-oriented
programming



QA

CLASSES AND OOP



Contents

- Using objects
- Duck-typing
- A little Python OO
- A simple class
- Defining classes
- Defining methods
- Constructing an object
- Special methods
- Operator overloading
- Properties and decorators
- Inheritance

Summary

2

Object-orientation offers one of the best approaches to getting the most out of the Python language and libraries. This chapter presents and works through some of the key structuring concepts found in object modelling.

QA Using objects

Calling a class creates a new *instance object*.

- Invokes the constructor.

```
from account import Account  
  
some_account = Account(1000.00)  
some_account.deposit(550.23)  
some_account.deposit(100)  
some_account.withdraw(50)  
print(some_account.getbalance())  
  
another = Account(0)  
  
print(Account.numCreated)  
print("object another is class",  
      another.__class__.__name__)
```

```
1600.23  
2  
object another is class Account
```

Just calling a class will call specific code to construct or *instantiate* an object. Unsurprisingly, the function to construct an object is called a *constructor*, and it can be passed parameters in the usual way. In the Account example above, we are passing 1000.00 as a parameter to the constructor, presumably an opening balance.

The constructor will return a reference to the object, which we can then call functions (methods) on. As we shall see, these functions are passed the object reference as their first parameter.

Object references are not normally printable, but we can provide our own *stringification* function, and others. This is part of special function overloading, which is related to operator overloading - a feature often seen as important in OO programming (although sometimes overdone).

The example shows a way of getting the name of the class that an object belongs to. This uses the attribute `__class__` which all objects have. In Python 3, you could also use the `type` built-in, but that gives output in a less useful format:

```
>>> print(type(another))
<class 'Account.Account'>
```

However, testing to see which class an object belongs to breaks the principles of duck-typing - you should instead check to see if it quacks, i.e. use `hasattr()` to see if it does what you need.

QA A class is not a type!

Don't ask what type an object is, only ask what the object can do.

```
hasattr(object, name)
```

This is known as **duck-typing**.

"If it walks like a duck, swims like a duck, and quacks like a duck ..."

- In the example, we don't care what class `x` belongs to, only that it supports representation as a string.

```
if hasattr(x, '__str__'):
    val = str(x)
```

Based on the original concepts of object orientation:

- Send/receive messages to/from an object.
- Signature-based polymorphism.



A foundation concept of object orientation (OO) is that messages are sent to objects requesting actions. We should not need to check the class of the object, only that it can carry out the action requested. That principle has been lost with many OO languages, particularly static ones. Even those that fully support polymorphism, the practice is often to test the class rather than the behaviour.

Python, and most dynamic languages, allows the programmer to check if behaviour exists. This is called *Duck Typing*: we don't care what kind of thing it is, if it quacks that's good enough. It could be some kind of duck mimic, who cares?

Q&A little Python OO

A class is declared using `class`:

- Membership is by *indentation*.
- "*class names use the CapWords convention*" – PEP008.

Methods are declared as functions within that class:

- First argument passed is the object.
- The constructor is called `__init__`
- The destructor is called `__del__` but rarely required and unreliable.

Classes are usually declared in a module:

- File usually has same name as the class, with `.py` appended.
- Simple example over...

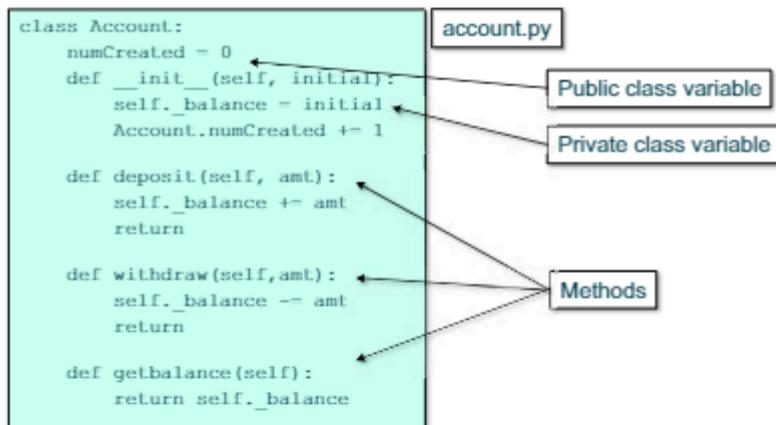
Creating a class is very simple in Python, and an example is shown on the next slide. We shall be discussing the constructor in more detail later, but you can see that special function names begin and end with two underscores. There are many more than just the constructor and destructor.

The destructor, called when an object is destroyed, is rarely needed in Python because Python does its own memory management through reference counting. Remember that Python variables are actually references to objects, not objects themselves. So, when a reference to an object drops out of scope, or gets reassigned, it does not necessarily destroy the object. The destructor does not get called until the reference count drops to zero, and even then, Python does not guarantee to call it, even when the program shuts down! Do not rely on the destructor for committing transactions, closing network connections or flushing buffers - use exception handling and a `finally` block (discussed in another chapter) instead.

Q& Defining classes

The class statement

- Defines a class object.
- Public attributes are referenced by *Class.attribute*.
- Usually in a module with the same name as the class.



Here is a simple example, which would be imported into a client program using:

```
from account import Account
```

Note the class variable `numCreated`, which is exported by default, since its name does not begin with an underscore character.

QA Defining methods

Methods are functions defined within a class.

Conventions with underscores – reminder

- Names beginning with one underscore are private to a *module/class*.
- Names beginning with two underscores are private and mangled.
- Names surrounded by two underscores have a special meaning.
- Note: these do not guarantee privacy!

Object methods

- First argument passed to a method is the object.
 - Usually called 'self', but can be anything.

Class methods and attributes

- Defined within the class.
- Can be called on a class or object.

The conventions for exporting names from modules apply with classes as well.

There are several *introspection* techniques which allow protected names to be seen, but the intention for privacy is clear.

Class methods are not often required - it is usually better to implement a module function instead. It is not advisable to call a class method on an object - mainly because it looks confusing.

A function name prefixed by a single underscore indicates the name is meant for internal use only in a class. It's not really private like other languages but merely a convention for programmers to not access the names.

A name prefixed with a double leading underscore (dunders) has its name mangled by the Python interpreter in order to avoid naming conflicts in subclasses. Any identifier, such as `_spam`, will have its name changed to `_classname__spam`. This is helpful for letting subclasses override methods without breaking the parent

classes' methods.

QA Constructing an object

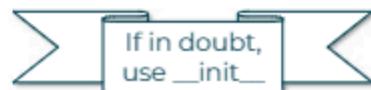
Python has two alternative methods to construct an object:

`__new__`

- Called when an object is created.
- First parameter is the class name.
- Return the constructed object.

`__init__`

- Called when an object is initialised.
- First parameter is the object.
- An implicit return of the current object.
- `__new__` is called in preference.



Which to use?

- Use `__new__` only if constructing an object of a different class.
- In most cases, use `__init__`

The constructor is a little confusing, having two possible functions.

We rarely need a `__new__` function, usually we implement

`__init__` which will be called automatically. As can be seen from the slide, `__new__` is actually a class method and is called before the object is created. We need `__new__` if we are going to create an immutable object (like a string or a tuple), which is not very often, or if we are going to use metaclasses. If `__new__` is provided, `__init__` is not called.

Python 2 note: In Python 2 `__new__` is only called for derived (new-style) classes. Base classes in Python 2 should inherit from `object` (more on inheritance later). All Python 3 classes are new-style classes.

QA Special methods

A mechanism for operator and special function overloading.

- Assists with duck-typing.

Function names start and end with two underscores:

<code>__bool__(self)</code>	Return True or False
<code>__del__(self)</code>	Called when an object is destroyed
<code>__format__(self, spec)</code>	<code>str.format</code> support
<code>__hash__(self)</code>	Return a suitable key for dictionary or set
<code>__init__(self, args)</code>	Initialise an object
<code>__len__(self)</code>	Implement the <code>len()</code> function
<code>__new__(class, args)</code>	Create an object
<code>__repr__(self)</code>	Return a python readable representation
<code>__str__(self)</code>	Return a human readable representation

Special methods are allied to overloaded operators only, here we are overloading Python built-in functionality. For example, `__str__` is called if an `str()` operation is done on our object, even implicitly such as in a `print` statement.

The Python 3 specific here is `__bool__`, which is called `__nonzero__` in Python 2.

QA Operator overload special methods

All operators may be overloaded

- See the online documentation for a complete list.

Return types vary

- Can return a `NotImplemented` object.
- Examples:

<code>__add__</code>	<code>+</code>
<code>__sub__</code>	<code>-</code>
<code>__eq__</code>	<code>==</code>
<code>__ge__</code>	<code>>=</code>
<code>__lt__</code>	<code><</code>
<code>__invert__</code>	<code>~</code> (logical NOT)
<code>__getitem__(self, key)</code>	container element evaluation
<code>__setitem__(self, key, value)</code>	container element assignment

This is not a complete list of operator overload special methods, that is far too large to fit here - see the online documentation (search for `__add__(object method)`).

Compound (augmented) assignments, such as `+=` have their own set of special methods, usually their names have a `i` prefix. For example, the special method for `+=` is `__iadd__`. Fortunately, if that is not supplied then Python looks for a `__add__` method and uses that instead, so we do not have to write versions of the same code (unlike C++).

However, methods overloading binary operators `+, -, *, /, %, **, <<, >>, &, ^, |` are not automatically symmetrical, and require a method name prefixed '`r`' to make them so. For example:

<code>object + 1</code>	calls <code>__add__</code>
<code>1 + object</code>	calls <code>__radd__</code>

Many of these methods are similar and will often call an underlying method. For example, `__eq__`, `__ne__`, `__lt__`, `__gt__`, `__le__`, `__ge__` can all be implemented as a single-line call to a generic comparison method. The comparison method will often be

implemented as returning -1, 0, or +1 (where 0 means equality).

QA Special methods - example

```
class Date:  
    def __init__(self, day=0, month=0, year=0):  
        self._day = day  
        self._month = month  
        self._year = year  
  
    def __str__(self):  
        return f"{self._day:02d}/{self._month:02d}/{self._year:d}"  
  
    def __add__(self, value):  
        retn = Date(self._day, self._month, self._year)  
        retn._day = retn._day + value  
        retn._validate_date()  
        return retn  
  
today = Date(9, 10, 2015)  
print(today)  
tomorrow = today + 1  
print(tomorrow)
```

Note private variable and method names starting with underscores

09/10/2015
10/10/2015

We are using 'this' in one method and 'self' in others just to demonstrate that it can be done - don't do this at home! Choose one convention and stick with it consistently - if in doubt use '**self**', most Python programmers do and so does the Python documentation.

In the user code, the `print` statement will execute the `__str__` method, and the `+=` operator will execute `__add__`.

Notice in the `__add__` we are not altering the current object, we are altering (and returning) a copy. There are several ways we could copy the current object we call the constructor (`Date()`) to create a new one from scratch.

However, there might be additional attributes added by other methods. In this case `copy.deepcopy()` could be used instead. The `copy` module will do its best to copy the object, but sometimes that is not enough, in which case, the class can implement a `__deepcopy__` method, which will be called when `copy.deepcopy` is called on an object of that class.

The downside of not calling the constructor is that there might be a missing side-effect. For example, if the constructor kept a global (class variable) count of all the objects - `deepcopy()` would miss that.

In general, the method shown on the slide is preferable, i.e., calling the constructor. It then follows that adding attributes to an object in later methods should be considered bad practice. If it can't be avoided, then a more complex constructor is required that optionally takes an object as an argument.

There are several date/time handling modules in the Python Standard Library.

QA Properties

Built-in property() creates an attribute

- `property()` has getter, setter, deleter, and docstring.
- The appropriate method is called depending on the way the attribute is used.

```
class Date:  
    ...  
    def mget(self):  
        return self._day  
  
    def mset(self, day):  
        self._day = day  
  
    mday = property(mget, mset)
```

Call the (default) getter method
day = today.mday

Call the setter method
today.mday = 6

Omitting the setter method means that the attribute is *read-only*

The `property()` built-in function defines get, set, delete, and docstring methods for a specific attribute. All parameters can be defaulted, following the usual rules. In the example, we have only used setter and getter methods.

When the attribute is used for read, for example on the right-hand side of an assignment, then the getter method (first parameter) is used. When used on the left-side of an assignment the setter is used.

In the example, if we used `del today.mday` then it would attempt to call a deleter method, but we have not provided one so that would fail with:

```
AttributeError: can't delete attribute
```

The docstring parameter is also missing in the example, but this time an attempt to use it will use the getter's docstring by default.

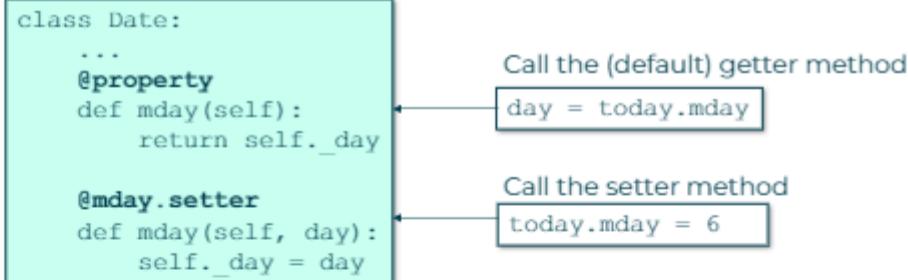
Properties and decorators

A decorator is a function name prefixed @:

- The function will normally return another function.
- The decorator is followed by the function to be returned.

Decorators are syntactic sugar, but commonly used:

- Built-in property() is usually called using a decorator.



The `property()` built-in function defines get, set, delete, and docstring methods for a specific attribute. It is rarely called in the conventional manner, but by using a *decorator*.

Decorators only work with new-style objects (they might appear to work sometimes with old-style, but some aspects do not).

The @ sign marks a *decorator expression*, and it can be applied to any function which takes function references as parameters, in this case the `property()` built-in. It is equivalent to calling the function (`property()`) and passing the supplied functions as parameters.

The example on the slide is equivalent to:

```
property(first_mday, setter=second_mday)
```

By setting `@property`, we define a default "getter" method which returns an attribute of the class. Notice that the attribute (`_day` in the example) is not defined elsewhere. We can also define a setter and deleter method by using the general syntax `@ + getter_method_name + .setter or .deleter` (docstring is not supported using `property()` with a decorator).

Note that the `names` `setter`, `getter`, `deleter`, are not the keywords to `property()`, they are methods in the `property` class which can be used as decorators.

Decorator expressions are used elsewhere, although you may find the concept rather advanced. This is the most common, and there are others, including `@classmethod` (next slide), and `@contextmanager`. Modules such as Twisted, Turbogears and Django also use decorators.

You can create your own decorators, but this is outside the scope of this course.

QA Decorators - what's the point?

Decorators are part of Python function syntax.

- Not specifically OO, but often used in OO contexts.
- Part of *metaprogramming*.

One aim is to make code easier to read.

- The decorator 'decorates' functionality.

```
def mget(self):
    return self._day
...
mday = property(mget, mset)
```

Trailing property() call might be missed, or forgotten. When does it get executed?

```
@property
def mday(self):
    return self._day
...
```

Method is bound to the attribute name, and bound to @property.

The requirement for decorators comes from *metaprogramming*, which is programming capable of modifying the code itself at runtime.

Remember that decorators are not only used with properties, but they can also be used with any function that takes a function as its first (and subsequent) arguments.

They are often used with built-ins, and they were originally added specifically to help with the syntax of creating class methods (see over).

Their justification is described in PEP 318 as an aid to readability. The problem with using built-ins like `property()` is they are not necessarily bound in code to the functions they are describing. We must create new functions that have to be named differently to the attribute that they will be tied to. It might not be obvious that there is a logical link between these functions and the attributes. The code we have shown so far is simple, imagine trying to track properties with a large and complex class.

QA Class methods

There are several ways to achieve this:

- Using a dummy class wrapper.
- Using the `classmethod` built-in as a decorator (preferred).
- The class method itself.

```
_count = 0  
...  
@classmethod  
def get_count(cls):  
    return Date._count
```

The class name is passed implicitly

- The user of the class.

```
from date import Date  
...  
counter = Date.get_count()
```

Class methods are not often required - it is often better to implement a module function instead. It is not advisable to call a class method on an object - mainly because it looks confusing.

If you try to define a function within a class without an argument, it is **unbound**, neither an object or a class method. In Python 2, this produced an error, but is now allowed.

A class method is considered **bound** to the class and gets the class name (called **cls** by convention) as a parameter. The `@classmethod` decorator is required to indicate this.

An alternative is `@staticmethod`, which is more like a class method in Java or C++ but does not pass the class name.

QA Inheritance

Use attributes and methods from a parent class.

- Important OO concept.
- Python supports multiple inheritance - not often needed.
- Attributes and methods not supplied in the derived class will be inherited from the base class.
- Common to derive our own classes from Python's own:
 - Multithreading
 - Exceptions
 - etc.

```
class DerivedClassName(base_classes):  
    def __init__(self, arguments):  
        base_class.__init__(self, arguments)
```

Other methods...

In object-oriented technology, inheritance provides an "is a" relationship between two classes. The derived class inherits all of the attributes and all of the operations of the base class. It can also add its own data members and member functions to provide more specialised characteristics.

Furthermore, the derived class can override a function in the base class, if it doesn't quite meet the specific requirements of the derived class. This is known as polymorphism.

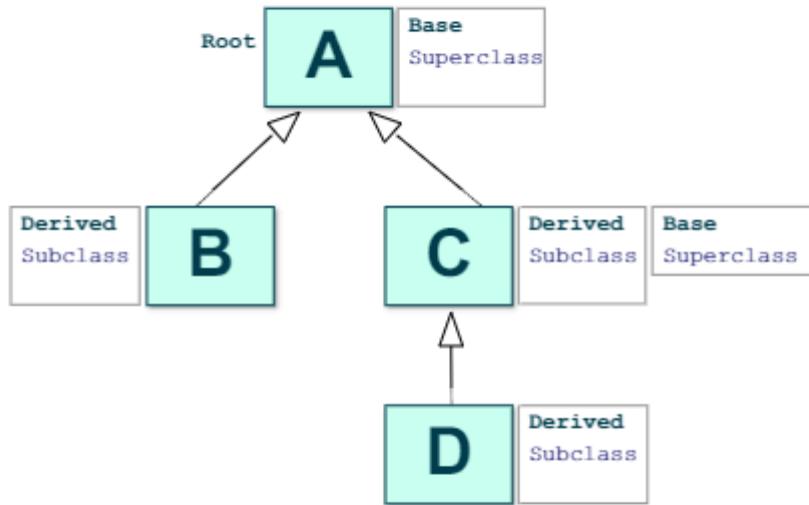
Inheritance is a natural concept, and we use it in our day-to-day lives. We learn at any early age to classify similar objects in terms of their fundamental characteristics. We speak of cars in general terms, without worrying too much about a particular type of car, such as four_wheel_drive_car or sports_car. As soon as we mention the word 'car', the basic properties are understood because all cars have certain features in common; all cars have wheels, for example, and they all have brakes.

The same is true with inheritance in the world of object-oriented

technology. Inheritance allows the class designer and implementer to group related classes together under a single umbrella, so that they can be considered collectively in general terms or individually as specific classes.

By reusing tried and tested services of an existing class, applications can be developed more quickly and with a greater degree of confidence in the end product. Inheritance can result in code implosion, since the same base-class functions can be used in many derived classes. In Python, it is very common to derive our own classes from Python classes.

Q& Inheritance terminology



There are several different methodologies available for expressing object-oriented designs, and many have an accompanying notation for representing object-oriented designs graphically. We shall adopt the simple notation shown to represent a hierarchy of derived classes, using arrows to indicate which is the base class.

For example, class C is derived from class A and class D is derived from class C. So, D inherits all the members of C, which in turn inherits all the members of A.

QA Inheritance scope

Attributes are either “public” or “private”.

- No equivalent of “protected”.
- This includes methods as well as data items.
- Enforced by the leading two-underscores rule.
- Base and derived classes can share the same module.
 - Can share attributes privately that are prefixed with a single underscore.

Public attributes of the base class can be called on an object of the derived class.

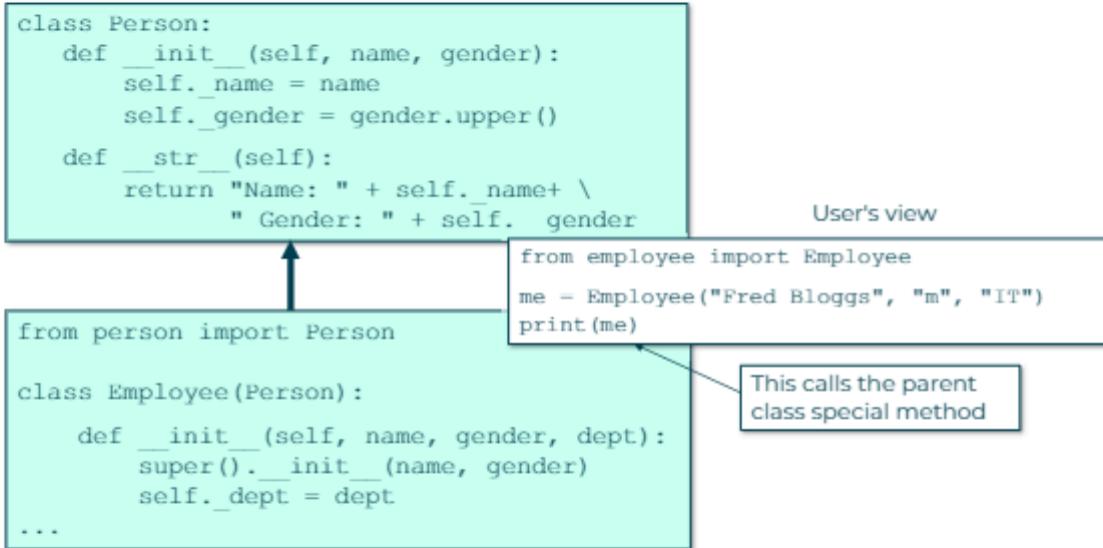
- Also applies to `__special__` methods.

When a derived-class object is used to call a method, and that method does not exist in the derived class, then a base-class method of that name is called (if one exists).

An interesting situation exists, if a base class and its derived class have the same method names. In this case, if a base-class method calls one of these duplicated method names and ‘self’ is a derived-class object, then the derived-class method is called, not the base-class one. This applies even if the duplicated method names have two leading underscores, i.e., are private.

The same problem exists with public data attributes, but not private ones.

QA Inheritance example



In this example, we have a base class called `Person`, and a derived class called `Employee`. The only thing which the `Employee` class does is provide its own constructor, although other methods would normally have been provided as well.

The `Employee` constructor (`__init__`) calls the base class constructor using `super()`, which returns a base class object. This is controversial, and many people specifically call the base class instead. The issue is with multiple inheritance: when we derive from several base classes, which one will `super()` return? So, instead of the call using `super()` we could have done:

`Person.__init__(self, name, gender)`
(notice that we have to explicitly pass `self`)

The syntax for `super()` changed at Python 3.

QA Some helper built-in functions

isinstance(*object, classinfo*)

- Returns True if *object* is of class *classinfo*

issubclass(*class, classinfo*)

- Returns True if *class* is a derived class of *classinfo*

```
from employee import Employee
from person import Person

me = Employee("Fred Bloggs", 'm', 'IT')

if isinstance(me, Employee):
    print(me, "isa Employee!")

if isinstance(me, Person):
    print(me, "isa Person!")

if issubclass(Employee, Person):
    print("Employee is a subclass of Person")
```

All these conditions
return True
(based on the
inheritance example)

In both cases, *classinfo* can be a tuple of classes. These functions could be considered to discourage duck-typing.

QA

SUMMARY



Classes vs. objects

- A class is a user defined data type.
- An object is an instance of a class.
- Objects have identity.
- To achieve behavior, we call an operation on an object.
- The operations on an object are defined by its class.

Encapsulation

- Separates interface from implementation.
- Publicly accessible operations.
- Privately maintained state.

Classes represent a convergence of the traditionally separate concepts of function and data. A class defines a new data type, providing it with a name, a set of operations expressed as member functions, and a representation expressed as member data. An object is a runtime instance of a class. There can be many objects of a single class, and many classes within a system.

The key benefit of objects is that they are self contained runtime components defined in terms of their external behaviour, collaborations and responsibilities, rather than in terms of their private representation. The combination of function and data within the same unit, and the protection of that data via an effective fire wall, is known as encapsulation.

QA Metaclasses and ABC

A metaclass is a class for creating other classes.

- The syntax for metaclasses changed at Python 3.

Abstract Base Classes

- Classes that cannot be directly instantiated.
- Created metaclass ABCMeta and decorator abstractmethod.

```
from abc import *
class Vehicle(metaclass = ABCMeta):
    @abstractmethod
    def getReg(self):
        pass
class Car(Vehicle):
    def getReg(self):
        print("Car isa Vehicle")
```

```
beepbeep = Car()
beepbeep.getReg()
Car isa Vehicle
```



```
NoGo = Vehicle()
Can't instantiate
abstract class Vehicle...
```

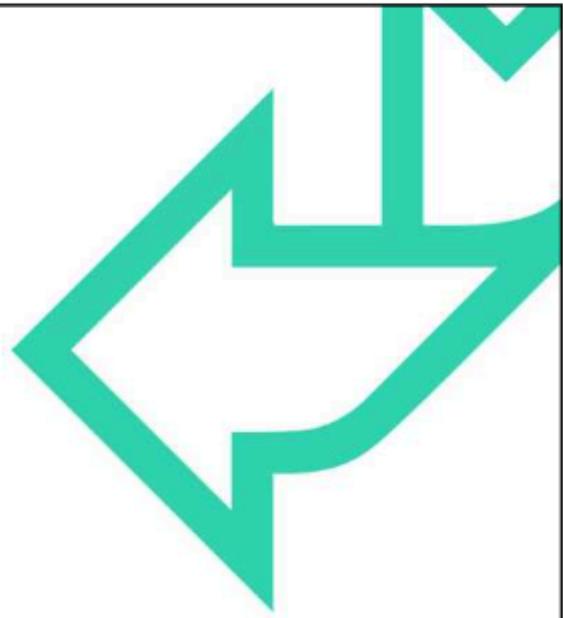
ABCMeta is exported by the abc module. With meta-classes we might want to provide a `__new__` function, since the actual class used for the object creation could be different to the current class.

See PEP 3119 for further details.



Python 3 Programming

Error handling and exceptions



QA

ERROR HANDLING AND EXCEPTIONS



Contents

- Writing to stderr
- Exception handling
- Exception syntax
- Exception arguments
- The finally block
- Order of execution
- The Python 3 exception hierarchy
- assert
- The raise statement
- Raising our own exceptions
- Getting tracebacks

Summary

QA Writing to stderr

Don't forget that error messages should go to stderr.

- Script errors are often redirected by the user.
- Ordinarily print goes to stdout, but it can be changed.
- Syntax for using stderr with print changed at Python 3.
- Using sys.stderr.write outputs to the terminal in red.

```
$ myscript.py > out 2> err
```

```
import sys  
  
if something_nasty:  
    sys.stderr.write("Invalid types compared")  
    exit(1)
```

Version neutral

- In Python 3, we can use the file parameter:

```
print("Invalid types compared", file=sys.stderr)
```

- See errno module, and os.strerror() for error number translation.

Most people know what stderr is for - writing error messages - yet it is very common to see scripts where error messages are written to stdout instead. Python routes its own error messages to stderr, but it cannot know which of your messages are errors, so it is up to you!

Using stderr is important because many tools use that stream to route error messages elsewhere, or to indicate if an error has occurred. For example, a common system administrator's trick is to redirect stderr (file descriptor 2) to a file, then test the size of that file after the script has run. If the file size is zero, then the script did not produce any errors and so it worked! You can argue this is flawed logic, the return value should be tested instead, nevertheless it is a common technique.

For more sophisticated error message routing, see the **logging** module (part of the standard library). For debugging and error reporting in web applications, see the **cgitb** module (should be bundled with your installation).

The syntax for routing **print** output to stderr changed at Python 3. Prior to that release, the old syntax was:

```
print >> sys.stderr, message
```

However, the syntax using:

```
sys.stderr.write (message)
```

did not change.

Standard error numbers are defined as literals in the `errno` module and can be converted to strings using `os.strerror()`. For example, error number 2 is `errno.ENOENT` which is "No such file or directory".

QA Controlling warnings

Warnings can be generated by Python and by user code

- Is a warning to be issued?
- Where should the warning be sent?
- Default: sys.stderr.

The warnings standard module gives us control

- Generate user warnings with `warnings.warn()`
- Sending and formatting uses functions which can be overridden.
- Warnings can be filtered by type, text, or category.

Can be controlled through the -Wd command-line option

- This makes warnings visible that are usually ignored.
- DeprecationWarnings are not displayed unless turned on using -Wd or a warnings filter.

4

By default, Python warnings are written to stderr, but they are controllable from a program in a number of ways. However, you don't see warnings very often, because the most common ones are, by default, ignored. We can also generate warnings (class `UserWarning`) from within a program.

`DeprecationWarning`, `PendingDeprecationWarning`, and `ImportWarning` are ignored unless the `-Wd` option (or a filter) makes them visible. The filter to turn the `-Wd` option on within a program is:

```
warnings.simplefilter('default')
```

Warnings are technically part of exception handling, but they are controlled in a very different way. We can describe them in a filter using a regular expression, and some simple examples are shown on the next slide.

We shall see later where Warnings fit into the exception hierarchy.

Further warnings control can be achieved using a context manager, which is outside the scope of this course.

QA Warnings - examples

Raise a non-fatal UserWarning

```
import warnings  
  
warnings.warn('Oops')  
print('Ending...')
```

warn.py:4: UserWarning: Oops
warnings.warn('Oops')
Ending...

Turn a warning into a fatal exception

```
import warnings  
  
warnings.simplefilter('default')  
warnings.filterwarnings('error', '.*')  
  
import time  
time.accept2dyear = True  
time.asctime((11, 1, 1, 12, 34, 56, 4, 1, 0))  
  
print('Ending...')
```

Equivalent to -Wd option
RegExp filter (all warnings)

This raises a DeprecationWarning

Will not be executed

The first example is very simple, sending a warning message from the user. Note that the program will complete, despite the warning message.

The second example sets DeprecationWarnings (and others) on, the equivalent to the `-Wd` command-line option. The filter then turns all warnings (the `'.*'` regular expression) into errors. The `time` methods are just a handy way of generating a `DeprecationWarning`.

Possible actions include:

error	turn matching warnings into exceptions
ignore	don't print matching warnings
always	print matching warnings
default	print the first occurrence of warnings for each location where it is issued
module	print the first occurrence of warnings for each module where it is issued
once	print only the first occurrence of warnings, regardless of location

QA Exception handling

Traditional error handling techniques include:

- Returning a value from a function to indicate success or failure.
- Ignore the error.
- Log the error, but otherwise ignore it.
- Put an object into some kind of invalid state that can be tested.
- Aborting the program.

In Python, an exception can be thrown:

- An exception is represented by an object.
 - Usually of a class derived from the **exception** superclass.
 - Includes diagnostic attributes which may be printed.
- Throwing an exception transfers control.
- The function call stack is unwound until a handler capable of handling the exception object is found.

For the majority of programmers, their least favorite chore is testing and handling error conditions. Often, there are many ways for something to fail, resulting in exception code that can dwarf the core program logic.

One of the traditional approaches to signaling errors is to return some kind of status value from a function. There are a number of problems with this:

Can lead to cascading, if else, if code that can be hard to read.
Overloaded operators do not have a 'spare' return value for exceptions.

Return values are too easy to ignore.

One technique is to have an object know whether or not it is in a good or bad state, which can then be queried. The problem with this is that, although it can handle constructor problems, it can lead to clumsy code. It also requires extra members for manipulating and representing the error, making the class less cohesive.

Aborting the program is a last resort. It is up to the caller to determine what the failure policy should be, and for the called component to detect any failure.

The Python solution is to represent exceptions as objects. As objects, they can be self describing, whereas something like a simple integer value – as used in other languages – is not very informative, as it says nothing about the details of an exception. The next feature of Python's exception handling is that there is a separate path of control for exceptions than for normal function returns. Exceptions can be caught by defining a handler, but there is otherwise no need to detect an exception in a function, solely to return it to that function's caller so that it can do the same – you only catch exceptions you are interested in.

QA Exception syntax

Unhandled exceptions terminate the program.

Trapping an exception:

```
try:  
    code body  
except [exception_list [as var]]:  
    exception handler  
else:  
    statements if no exception  
finally:  
    final statements
```

Optional, not executed if an exception occurs

Optional, always* executed

- Although we use terms like "try block", there is no real or implied scope within each code section.

7

The try block contains code to be tested. Should any of that code (which is often a function call) raise an exception then it can be trapped, and code executed in the except block to handle it. If an exception list is specified, then the handler code will only be executed if the exception is in the list, otherwise the stack will be unwound until a handler is found. Within the except block, the exception raised will be available if the optional as statement was used. The exception object has a number of useful attributes available, which vary depending on the subclass of exception.

The else block will only be executed if everything in the try block worked OK.

The finally block is guaranteed to be always executed, regardless of what happened in the code body (but see later).

Q\Multiple exceptions

It is common to wish to trap more than one exception.

- Each with its own handler.
- Or multiple exceptions with the same handler.

```
filename = "foo"
try:
    f = open(filename)
except FileNotFoundError:
    errmsg = filename + " not found"
except (TypeError, ValueError): ←
    errmsg = "Invalid filename"
...
if errmsg != "":
    exit(errmsg)
```

For example, `TypeError` would be raised if `filename` was not a string.

Remember, `exit()` raises a `SystemExit` exception!

Statements within the `try` block, particularly when functions are called, could raise different exceptions depending on circumstances. In this case, it is possible to test for more than one exception in a structure not unlike a case statement, with multiple `except` tests.

The same handler code can be executed by supplying a tuple of exceptions to the `except`.

QA Exception arguments

Each exception has an `args` attribute:

- Stored in a tuple.
- The number of elements, and their meaning varies.
- Other attributes may be available.

Access the exception using the 'as' clause:

```
import sys
try:
    f = open("foo")
except FileNotFoundError as err:
    print("Could not open", err.filename, err.args[1], file=sys.stderr)
    print("Exception arguments:", err.args, file=sys.stderr)
```

```
Could not open foo No such file or directory
Exception arguments: (2, 'No such file or directory')
```

When an exception is raised, an exception object is created and, like other classes, exception has attributes. New with Python 3 is the `__traceback__` attribute, which shows the stack trace.

Different types of exceptions are actually subclasses of the exception class, and different subclasses have their own attributes. For example, as the slide shows, subclass `FileNotFoundException` includes an attribute `filename`. This was introduced in Python 3.3, in previous versions `IOError` was used. The Python exception class hierarchy is shown on a later slide.

One way of understanding the syntax of the `except` statement is to think of it as being a special kind of function call. One parameter is passed - the exception object, and we specify which type we will accept. The name of that exception object within the "function" is the name given following `as`. Object attributes can then be interrogated using this name.

The syntax for the `as` clause changed at Python 3, it replaces a comma in Python 2.

QA The finally block

The **finally** block is (almost*) always executed.

- Even if an exception occurs.
- *`os._exit()` inside the `try` block ignores the `finally` block.

The **finally** block is executed **before** stack unwind.

```
def my_func():
    try:
        f = open("foo")
    finally:
        print("Finally block", file=sys.stderr)

    try:
        my_func()
    except OSError:
        print("An OS error occurred", file=sys.stderr)
```

Finally block
An OS error occurred



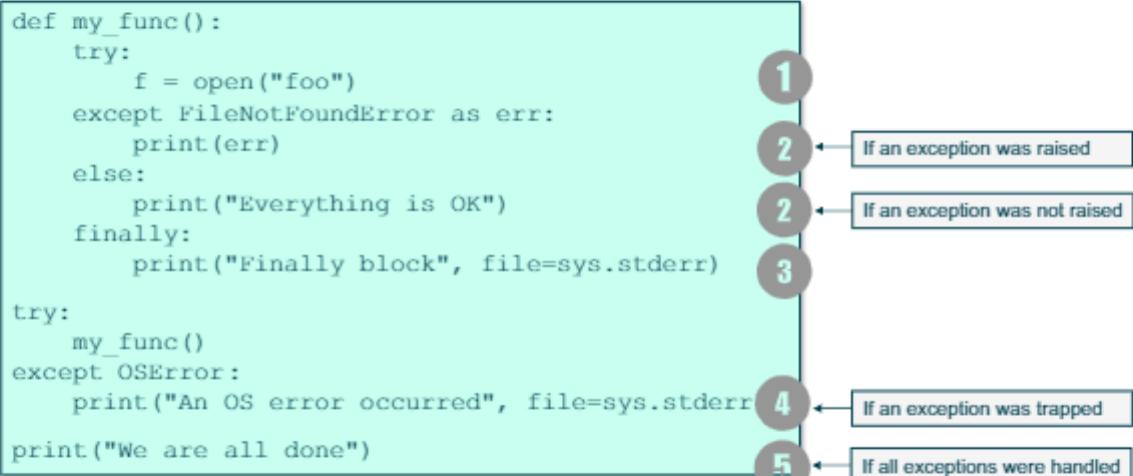
The idea of the **finally** block is that this can contain tidying/teardown or exit code which must always be executed. It is particularly useful as an alternative to a destructor, since, as we saw earlier, `__del__` might not be called.

It is also often used to release locks used, for example, in multithreading. If a lock is obtained in the `try` block, then we can guarantee that it will be unlocked if we put that code in a **finally** block. There is a danger with this, however. Just because we can release resources in the **finally** block, does not mean that the resource is necessarily consistent - the `try` block might have blown up! There is no automatic "rollback" functionality - you have to code that yourself. In the **finally** block code, do not assume *any* statements in the `try` block worked. Test handles and variables (usually for `None`) before trying to use or free them.

We cannot access the exception (if there is one) from within a **finally** block, and if the **finally** block itself raises an exception, then the original one is lost.

QA Order of execution

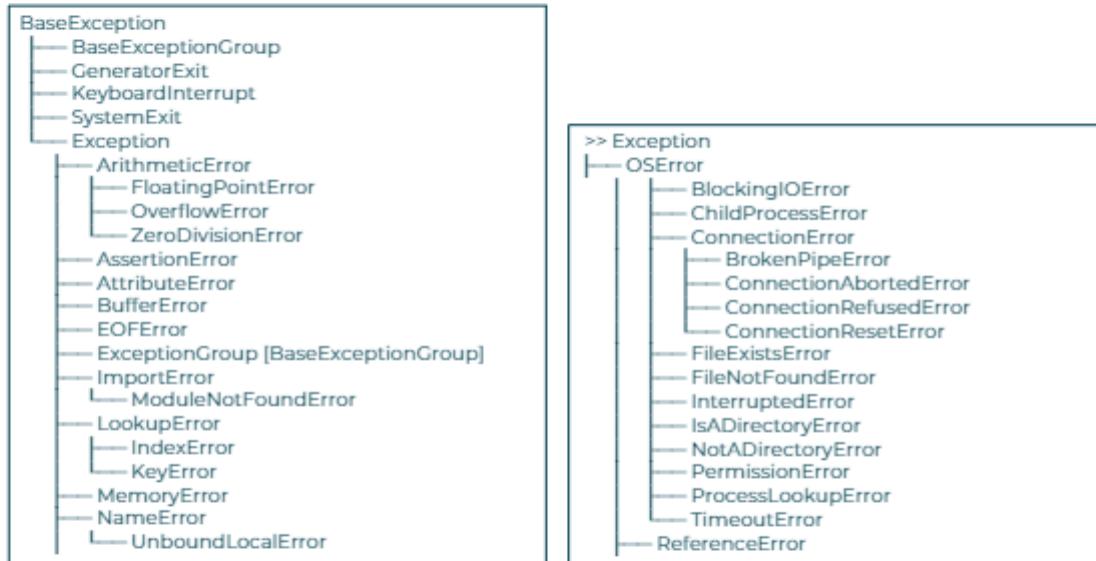
Either the except block or the else block is executed before the finally block.



Python attempts to execute the code in the **try** block. During execution, if an exception occurs, an **except** block is searched for to handle that exception. If none is found, then the stack is unwound until one is found, or the default exception handler is executed and the program ends. If an exception handler is found then that is executed, followed by the **finally** block (if there is one).

If an exception is not raised, then the **else** block is executed (if there is one), then the **finally** block.

QA The Python 3 exception hierarchy (1)



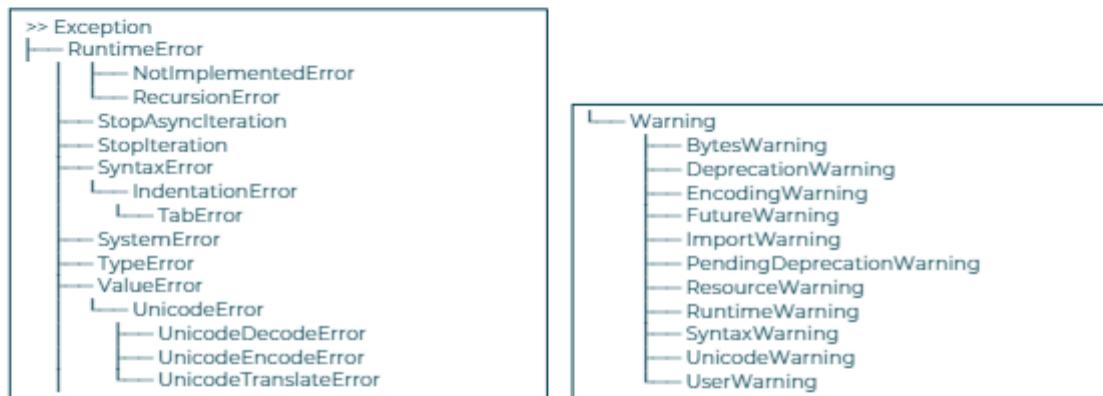
Although objects of any type may be thrown — not just those within the hierarchy shown above — it is sensible to use classes already defined in the standard exception hierarchy. If a new kind of exception is needed, a class for it can be derived from an existing class. For example, the database interface has its own exception hierarchy.

Be aware there could be changes to this hierarchy at every major release. At Python version 3.3, exceptions EnvironmentError, IOError, VMSSError, and WindowsError are aliases of OSSError, and are kept for backward compatibility.

It is a temptation to trap all errors, but that is a bad idea. There are those errors that we know are likely to occur and those we do not expect. Let the unexpected crash our program - at least the program then halts before it can do any damage.

Trapping all errors can lead to a false sense of security and can mask real bugs that we did not expect.

QA The Python 3 exception hierarchy (2)



Note that, although they are technically part of the exception hierarchy, warnings cannot be trapped using the `try...except` mechanism, unless turned into an exception using a filter.

QA assert

Raise an exception based on a boolean statement.

- `AssertionError` is raised if the boolean is `False`.
- May be associated with additional data.

```
assert expression [, associated_data]
```

```
def my_func(*arguments):
    assert all(arguments), 'False argument in my_func'
    ...
my_func('Tom', '', 42) AssertionException: False argument in myfunc
```

Not usually a good idea in production code.

- Comment out `assert` statements for production.
- Or run with `-O` (oh), or set `PYTHONOPTIMIZE` to 0.
- Sets `__debug__` to false.

The Python built-in `assert` has been inspired by the C/C++ macro of the same name, although it behaves differently. It is designed to provide sanity tests within code in a simpler form than having to type a full test. It is designed to be used during development, so usually you would not explicitly test for an assertion failure - crashing the program is probably what you need.

That is not so good for an end-user, and we would not expect to find assertions in production code. Most people would just comment out `assert` calls after testing, but there are other ways of ignoring them.

The `-O` command-line option (`O` for Optimise) will ignore all `assert` statements:

```
python -O scriptname
```

This sets the builtin variable `__debug__` to false (default is true). The `__debug__` variable cannot be assigned, so it cannot be altered in a conventional way.

The same effect can be made by setting the environment variable `PYTHONOPTIMIZE` to a value such as 0 (this is the optimization level). An empty `PYTHONOPTIMIZE` environment variable will raise `AssertionError` exceptions, any value will ignore them.

In the example, we are using the built-in `all()` to test that all the

arguments to the function are True. This is a common assertion to make but remember that zero is False, yet might be a legitimate value.

Q^The raise statement

Throw a standard exception object, with data.

- Syntax change at Python 3.

```
def my_func(*arguments):
    if not all(arguments):
        raise ValueError('False argument in my_func')

try:
    my_func('Tom', '', 42)
except ValueError as err:
    print('Oops:', err, file=sys.stderr)
```

Oops: False argument in my_func

If no exception is specified:

- Repeat the current active exception.
- If no current exception, raise TypeError.

The **raise** statement syntax change at Python 3. We used to throw strings, but in Python 3, the system has been tidied and we throw standard exceptions. We can get the same effect by passing a string argument to, for example, ValueError. We can write to the `__cause__` attribute (not normally set) using, for example:

```
raise some_exception from another_exception
```

We can also create our own class and derive it from exception, or one of the other subclasses. This is discussed on the next slide.

Q A Raising our own exceptions

Define our own exception class

```
class MyError(Exception):
    pass

def my_func(*arguments):
    if not all(arguments):
        raise MyError('False argument in my_func')

try:
    my_func('Tom', '', 42)
except MyError as err:
    print('Oops:', err, file=sys.stderr)
```

An empty class derived from exception

Oops: False argument in myfunc

- In Python 3 we no longer raise string exceptions.

*

In early versions of Python, we passed a string to **raise**, and caught it using **except** (similar to C++). This has been corrected in Python 3 - we can only raise exceptions, not strings.

The syntax is still very simple, and it will not be unusual to see several customised exception classes in an application. The example shown is empty, and that will normally be the case.

The full syntax of the **raise** statement (note: it is not a built-in) includes the suffix with `traceback(traceback)`.

The traceback part would be used to carry forward data from a previous exception, for example:

```
except FileNotFoundError as err:
    raise MyError('something went wrong') from
err
```

It is also possible to create your own tracebacks using the **with_traceback()** exception method.

QA Getting tracebacks

`sys.exc_info()`

- Returns a tuple of type, value, and a *traceback* object
- The traceback object includes attributes such as the line number where the exception occurred, and the stack trace.

```
try:  
    open("some file name")  
except FileNotFoundError as err:  
    type, val, tb = sys.exc_info()  
    print("Exception lineno:", tb.tb_lineno)
```

The `traceback` module

- Includes utilities to trace back through an exception cascade.
- `traceback.print_exc()` is short for `traceback.print_exception(*sys.exc_info())`

Exceptions can occur in heavily nested function calls, and even exceptions can be nested. Several modules in the standard library can help us.

sys has three variables, `sys.last_type`, `sys.last_value`, and `sys.last_traceback`, conveniently retrieved as a tuple by `sys.exc_info()`. The traceback object can be used, although it is generally not accessed directly, but via the **traceback** module.

traceback has several useful methods that act on the current exception, but the most commonly used is `traceback.print_exc()`, which prints the exception stack-trace in the same way as the interpreter. For example:

```
class Exc1(Exception):  
    pass  
def func1():  
    try:  
        open("some file name")  
    except OSError as err:  
        raise Exc1(err)  
import sys, import traceback  
try:  
    func1()  
except:  
    traceback.print_exc()
```

The exception information is shown on the console terminal, which might not be desirable, so `traceback.format_exc()` returns a string in the same format which can be used for your own error logging.

QA

SUMMARY



Write error messages to stderr

Most modern languages support exception handling

- It is particularly suited to object orientation

Exceptions are built-in to Python

- Many built-ins raise exceptions
- Exceptions are not necessarily an error

Handle it!

- Trap code with try:
- Handle with except:
- Also support else: and finally:

We can also raise our own exceptions

Use assert for boolean tests, but not for production code

»

QA Context managers - with

Context managers execute entry and exit code

- Special methods `__enter__` and `__exit__`
- `__exit__` may handle exceptions, or close resources.
- Used with `with`.

```
with context_object as variable:  
    BLOCK
```

File objects are context objects

- Means we do not need finally blocks.

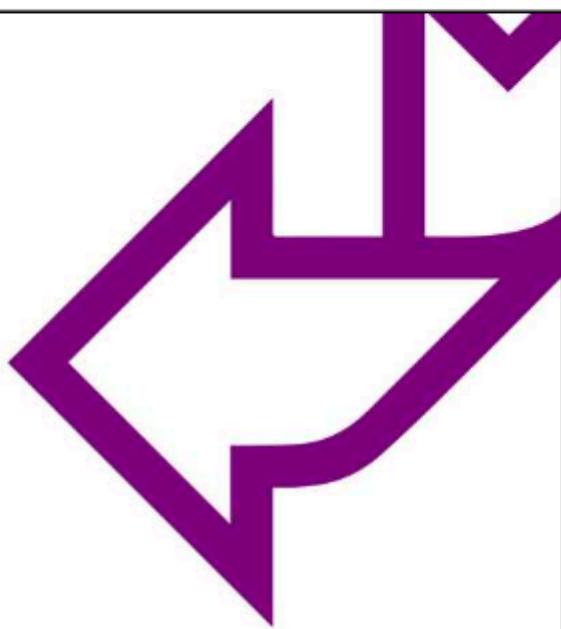
```
with open('spam.txt', 'rt') as fh_in:  
    for line in fh_in:  
        print(line, end='')  
    print(fh_in)  
    <_io.TextIOWrapper name='spam.txt' encoding='cp1252'>
```

Context managers were introduced in Python 2.5. Their advantage is that we can initialise and destroy objects within its context. A destructor might not get called immediately, but the `__exit__` method will be called on exit from the context, even in the event of an exception.



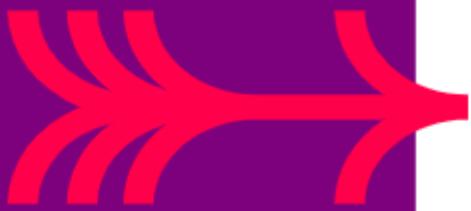
Python 3 Programming

Testing



QA

TESTING



Contents

- What is software testing?
- Types of testing
- Manual vs automated
- Create a simple calculator app
- Docstrings
- Docstrings testing
- Docstrings automated testing
- Unit testing
- The assert statement
- Test runners
- Test scripts
- The Pytest module
- Pytest output
- Summary



QA What is Software Testing?

Checks whether the Application meets the expected requirements

- Check for **Functional** requirements (When I click this button, an email is sent).
- Check for **Non-Functional** requirements (email is sent within 1 seconds and in a secure way).

Program code and logic is error free

- Logical errors.
- Arithmetic calculations...

Application errors

- Different modules work well together.
- System as a whole works.
- Works on the client system.

3

Software testing is the process of evaluating and verifying that a software application does what it is supposed to do and matches requirements. It is simply to confirm that it is defect free and there is no missing functionality.

QA Why software testing is so important?

Testing helps to solve problems early because:

- bugs are expensive to fix in production.
- lives could be affected (Airline, Transport...).
- reputation could be affected.
- performance and security are critical.

Examples of disasters caused by not testing:

- <https://dzone.com/articles/the-biggest-software-failures-in-recent-years>

There are many benefits of software testing, but the prime goal is to provide the best features and experience with no bugs or side effects. This in turn builds your reputation, enhances customer satisfaction, and leads to repeat business.

It can also lead to cost savings with early defect detection, economical fixes, fewer design changes and lower maintenance costs. Comprehensive testing can also lead to better quality code with lower failure.

In this digital web enabled world, safety and security is so important. Think about buying goods online, sharing personal details, or critical infrastructure and services. Testing can improve security against hackers, malicious attacks and thefts. It can even save lives by ensuring the software in the aviation world works correctly.

QA What are the different types of test?



Tests can be categorised as these general types:

Manual or Automated
Unit Testing
Integration Testing
System Testing
Acceptance Testing



Investigate Unit testing using several Python modules:

Document Testing using `doctest`
Test Framework using `unittest`
Test Framework using `pytest`

QA Manual or automated tests

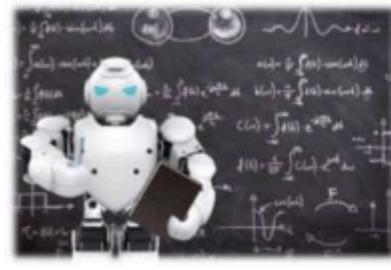
Manual testing

- Can NOT be designed and performed by anyone.
- Requires deep analysis of the requirements and specialised knowledge.
- Exploratory testing.



Automated testing

- Manually checking code may not be possible. For example, when testing:
 - Performance
 - Parallel execution
 - Load tests
 - Penetration test...
- Unit testing is often used to test code.
- It is the duty of the coder to write these.



You may not have realised, but you probably have already performed testing on your code! When you run your application, most coders will check to see if key features work, and if they do, they might consider a follow-on test. That is known as exploratory testing and is a form of manual testing, usually without a plan.

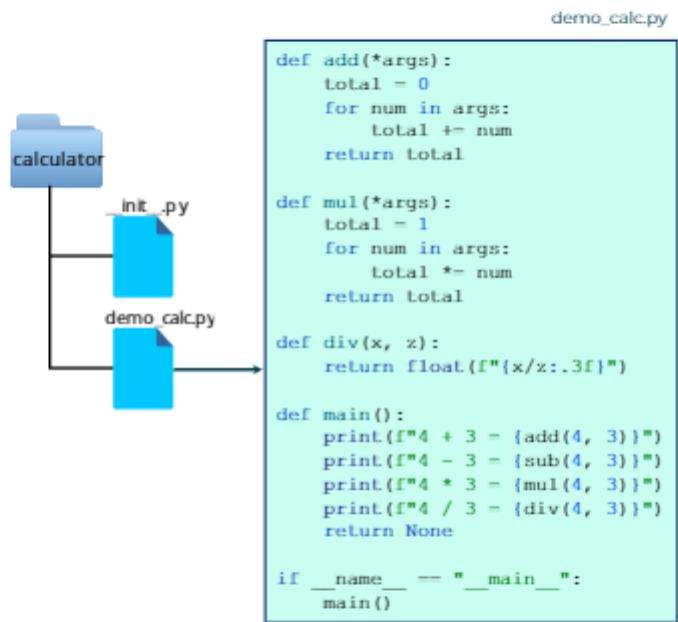
As the application scales, the list of features, and expected inputs and outputs increase. And every time the code changes you will have to manually re-apply all the tests. And possibly add more. Clearly not workable.

The solution is to automate your test plan using scripts. Luckily, Python already comes with a set of tools and libraries to automate your tests for your applications. We will discover how to use the **doctest** module to do DocString testing, and the **unittest** and **pytest** modules to perform automated unit testing.

Q& A simple calculator app

Demonstration

- Create a new Package called calculator.
- Notice the `__init__.py` file.
- Create a new script called `demo_calc.py`.
- Define add, multiply and divide functions.
- Add and multiply functions should allow multiple parameters.
- Define `main()` function to call functions.



To practice our testing capabilities, we will create a simple calculator application **demo_calc.py** with add, multiply, and divide operations.

Test the program by creating a `main()` function that calls all the functions with two simple parameters 4 and 3. Execute the program and confirm that all is well.

You may have done some of these steps in an earlier chapter.

QA A simple calculator app

Manual testing on the REPL or command prompt.

The image contains two side-by-side screenshots of computer windows. The left window is titled 'IDLE Shell 3.10.6' and shows Python code being run at a REPL prompt. The code imports the 'sys' module, appends a path to 'calculator', imports 'demo_calc', and then calls its 'add', 'mul', and 'div' functions with parameters 4 and 3. The right window is titled 'Select Command Prompt' and shows the output of running 'python demo_calc.py'. It displays the results of the three calculations: addition (4 + 3 = 7), multiplication (4 * 3 = 12), and division (4 / 3 = 1.333).

```
>>> import sys
>>> sys.path.append(r'C:\labs\projects\ProjectA\calculator')
>>> import demo_calc
>>> demo_calc.add(4, 3)
7
>>> demo_calc.mul(4, 3)
12
>>> demo_calc.div(4, 3)
1.333
>>>
```

```
C:\labs\projects\ProjectA\calculator> python demo_calc.py
4 + 3 = 7
4 * 3 = 12
4 / 3 = 1.333
C:\labs\projects\ProjectA\calculator>
```

Test the app manually by executing the script in your command prompt or importing the script at the REPL prompt and calling the functions manually. You may have to import the **sys** module and append the location of the script to the sys.path list.

You might want to manually test the add and multiply functions with multiple numeric parameters.

You did remember to put comments in your script or is there a better way to document your script?

QA

DOCUMENT STRINGS AND TESTING



Document and test your code – Docstrings:

- Documents the script.
- Documents the functions.
- PEP 008 compliance.
- Provides help().
- It can be tested!



If you are interested in writing your documentation at the same time as the code and want to incorporate test cases at the same time, then Python's **doctest** module should be considered.

It provides a simple testing framework that allows you to document your script with test examples and automate the testing of these examples – and helps keeps your code and documentation synchronised.

Some coders would say that documentation is just as important as code, and some may even argue it is more important. But all would agree that anything that helps testing is a good thing!

QA Docstrings – format and location

Docstrings are triple quoted strings that document your code, as you code:

- module docstrings placed before any code.
- function/class docstrings at start of block before any code.
- can be simple single line or multi-line.

```
'''  
    Calculator program with add, multiply,  
    and divide functions.  
'''  
  
def add(*args):  
    """ Returns the sum of all parameters """  
    total = 0  
    for num in args:  
        total += num  
    return total  
  
def mul(*args):  
    """ Returns the product of all  
    parameters. """  
    total = 1  
    for num in args:  
        total *= num  
    return total  
  
def div(x, z):  
    """ Returns x divided by z, as a float  
    to 3 decimal places."""  
    return float(f"{x/z:.3f}")
```

Comments are useful for describing code and in Python we use a # symbol for single line comments. Unfortunately, these comments are ignored by the interpreter, unavailable at runtime and can often become out of date!

Step forward, **docstrings** – these triple quoted (single or double) strings allow multi-line comments that remain within your code at runtime. They are displayed with the built-in help() function and modules such as MkDocs and Sphinx can be used to generate project documentation.

The special attribute __doc__ for packages, modules, classes and functions.

Docstrings can be added to packages, modules, classes, methods and functions and are described in the PEP 257 document.

QA Docstrings – for documentation

The screenshot shows the Python IDLE shell window. The title bar says "IDLE (Python 3.10.1)". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main area displays the following help output:

```
>>> help(demo_calc2)
Help on module demo_calc2:

NAME
    demo_calc2 - Calculator program with add, multiply and divide functions

FUNCTIONS
    add(*args)
        Returns the sum of all parameters.

    div(x, z)
        Returns x divided by z, as a float to 3 decimal places.

    main()
        Manually Test the functions.

    mul(*args)
        Returns the product of all parameters.

FILE
    c:\Users\Projects\ProjectA\calculator\demo_calc2.py

>>> help(demo_calc2.add)
Help on function add in module demo_calc2:

add(*args)
    Accepts multiple numeric parameters and returns the sum.
```

Use `help()` to see the docstrings:

```
>>> help(demo_calc2)
>>> help(demo_calc2.add)
```

Or the special attribute `__doc__`:

```
>>> print(demo_calc2.__doc__)
>>> print(demo_calc2.add.__doc__)
```

Use `pydoc` module to display in text or HTML:

```
c:> %PYTHONPATH%\pydoc demo_calc2.py
c:> %PYTHONPATH%\pydoc -w demo_calc2.py
```

Use the built-in `help()` function to display the embedded docstrings. You can also display the docstring for a specific function/method by using the dot notation – `help(demo_calc.add)`.

Alternatively, within a script, you can print out docstrings using the special `__dot__` special attribute which can be combined with package, module, class, function or method names.

The **pydoc** module, from the Python standard library, can also be used on the command line to print out the docstrings in text or html using the `-w` option.

QA Docstrings – for USAGE examples

Docstrings can also have embedded USAGE examples:

- Same format as testing at REPL prompt.
- Can have multiple examples in each docstring.

```
def add(*args):
    """ Returns the sum of all arguments
    >>> add(4, 3)
    7
    >>> add(10, 20, 30)
    60
    """
    total = 0
    for num in args:
        total += num
    return total
```

```
def mul(*args):
    """ Returns the product of all parameters.
    >>> mul(4, 3)
    12
    """
    total = 1
    for num in args:
        total *= num
    return total

def div(x, z):
    """ Returns the result of x divided by z,
    as a float to 3 decimal places.
    >>> div(4, 3)
    1.333
    """
    return float(f"{x/z:.3f}")
```

demo_calc3.py

You can also embed USAGE examples of each function in the docstrings. These examples are in the same format as the calls you would make at the REPL (>>>) prompt and include input parameters and expected output. The function must be deterministic which means it should always return the same output for the same inputs.

This helps the user understand how to call the function and what is expected out for given parameters, and how to manually test the function is working and matches the documentation.

But there is another benefit of USAGE examples in docstrings...

QA Docstrings – automated testing

Docstring USAGE examples testing can be automated:

- Import the **doctest** standard library module.

```
demo_calc3.py

def add(*args):
    """ Returns the sum of all arguments
    >>> add(4, 3)
    7
    """
    def mul(*args):
        """ Returns the product of all
        parameters.
        >>> mul(4, 3)
        12
        """
        def div(x, z):
            """ Returns x divided by z, as a
            float to 3 decimal places.
            >>> div(4, 3)
            1.333
        """
        def main():
            print(f"4 + 3 = {add(4, 3)}")
            print(f"4 * 3 = {mul(4, 3)}")
            print(f"4 / 3 = {div(4, 3)}")
            return None

        if __name__ == "__main__":
            import doctest
            doctest.testmod()
            main()
```



```
def add(*args):
    """ Returns the sum of all arguments
    >>> add(4, 3)
    7
    """
    def mul(*args):
        """ Returns the product of all
        parameters.
        >>> mul(4, 3)
        12
        """
        def div(x, z):
            """ Returns x divided by z, as a
            float to 3 decimal places.
            >>> div(4, 3)
            1.333
        """
        def main():
            print(f"4 + 3 = {add(4, 3)}")
            print(f"4 * 3 = {mul(4, 3)}")
            print(f"4 / 3 = {div(4, 3)}")
            return None

    if __name__ == "__main__":
        import doctest
        doctest.testmod()
        main()

# Tests passed: 4 / 5
```

The **doctest** module provides a framework for simple and quick automation of acceptance tests for integration and system testing. These are important as integration testing are used to confirm that the different components of your application all work together. And system testing is used to confirm that you have ticked off the specifications for your project.

Docstrings and doctests can be embedded at the package level down to the class, function, and method level. At the higher package level, they can be used for integration testing and at the lower function level they are suitable for unit testing – check the code and documentation as you code!

And the document tests can even be written before the code! How about that for a good idea!

QA Docstrings – automated testing

Docstring USAGE examples testing can be automated:

- Apply an error in your docstring and test again.

```
demo_calc3_errors.py

def add(*args):
    """ Returns the sum of all arguments.
    >>> add(4, 3)
    8
    >>> add(10, 20, 30)
    60

def mul(*args):
    """ Returns the product of all arguments.
    >>> mul(4, 3)
    12
    """
    def div(x, y):
        """ Returns x divided by y, as a float,
        to 3 decimal places.
        >>> div(4, 3)
        1.333
        """
        return ...

def main():
    print(f"4 + 3 = {add(4, 3)}")
    print(f"4 * 3 = {mul(4, 3)}")
    print(f"4 / 3 = {div(4, 3)}")
    return None

if __name__ == "__main__":
    import doctest
    doctest.testmod()
    main()
```

The screenshot shows a Python code editor with a file named `demo_calc3_errors.py`. The code contains three functions: `add`, `mul`, and `div`, and a `main` function. The `add` function has a docstring with an error: the word "product" is misspelled as "produt". The `mul` function has a blank docstring. The `div` function has a correct docstring. A red arrow points from the `add` docstring error to a terminal window below. The terminal window shows the command `python -m doctest demo_calc3_errors.py` and its output. It highlights the expected and actual values for the `add` test, showing a failure because the expected value is 8 and the actual value is 7. It also shows the results for the `mul` and `div` tests, which pass with expected values of 12 and 1.333 respectively. A summary at the bottom of the terminal window indicates 1 failure and 3 tests passed.

```
File "C:\labs\projects\ProjectA\www\scripts\python.exe", line 18, in <module>
  add(4, 3)
Expected:
8
Actual:
7

File "C:\labs\projects\ProjectA\calculator\demo_calc3_errors.py", line 29, in <module>
  div(4, 3)
Expected:
1.333
Actual:
1.333
```

In this example, we apply two errors to the DocStrings and re-run the automated testing.

You will notice in the output that Expected and Actual values are highlighted and a summary of how many failures occurred. This can happen when code is changed and the DocStrings are not updated accordingly.

QA

TESTING USING UNITTEST



Unit testing:

- Unit vs integration testing.
- Assertions.
- Using the **unittest** module.



Testing your documentation versus the code is one form of testing. But how do we test that our application is working completely and do the individual features all work correctly? In this part, we will examine integration testing versus unit testing, and will discover how simple it is to perform unit testing using the Python library module **unittest**.

QA Integration Vs unit testing

How to diagnose a complex application and project?

Integration testing

- 2nd Level of Testing.
- Testing the components work as a complete entity.
- Sometimes called black box testing and often performed by a test team.



Unit testing

- 1st level of Testing
- Testing individual units of code (classes, functions and methods).
- Sometimes called white box testing and performed by the coder.



Test step – call a function with parameters.

Test Assertion – check that it performs correctly.

When we purchase a new car, we all hope that the manufacturer has tested our car in readiness for it to be driven out the showroom. The manufacturer will have tested all components individually to check that they all work – this is called unit testing. They will also have performed additional tests to check that all the components when assembled work together – this is called integration testing.

Integration testing can be more problematic. For example, if the engine does not start when you turn the key (or press a start button), then many things could have failed. It could be the battery in the key fob, the car battery, the starter motor, the alternator, fuel, or electrical wiring problems.

The developer typically tests their units of code (classes, functions, methods) during the coding phase of your application. Whilst the integration testing is often managed by a testing team using scripts and tools.

In the testing world, if you turn to start the engine, this is known as

a Test Step, and if you hear the motor turning and the lights in the console switching on this is called a Test Assertion.

Modern cars will normally inform you if something is not working correctly – this is an example of a unit test built-in to the car's software. For example, you may be informed that you have low fuel or battery level.

QA Unit Testing – using assert

Python has a built-in function called assert() for performing unit testing.

assert

- Convenient method of inserting debugging assertions.
- Raises an exception based on a Boolean expression.
- An AssertionError is raised if boolean is False.
- Can have an optional expression.

```
assert expression1 [, expression2]
```

```
>>> assert demo_calc.add(4, 3) == 7, "Should be 7"
```

Can be ignored when optimisation is requested at runtime by:

- PYTHONOPTIMIZE = 0
- Command line option, python -O

```
If __debug__:  
    assert add(4, 3) == 7, "Should be 7"
```

Python supports tools and modules for performing Integration and Unit testing, but in this session, we will be focusing on performing unit testing.

The Python built-in assert() function has been inspired by the equivalent function in C/C++ although it has a different behaviour. It is designed to provide sanity testing within code rather than a full-blown test function. It is designed to be used during development so that the program would fail and generate an AssertionError if an expression fails. Probably not what the end-user would like to see in production code. So, either remove from production code or test the __debug__ special attribute before using the assert statement.

The __debug__ attribute can be altered by enabling the compiler for optimization by setting the PYTHONOPTIMIZE=0 environment variable or with a command line switch (-O).

QA Unit Testing

Test Cases:

- Write your test cases in a separate file.
- Keeps code and testing separate.
- Could write tests before the code.
- Downside is only the first failure is displayed.

Test Runners:

- Specialised tools for running tests.
- Checking output.
- Debugging and diagnostic tools.
- **unittest** and **pytest** are popular.

```
test_calc.py

"""
    Test Cases for Calculator app.:
"""

from calculator import demo_calc

def test_add():
    assert demo_calc.add(4, 3) == 7, "Error should return 7"
    assert demo_calc.add(10, 20, 30) == 60, "Error should
return 60"
    return None

def test_mul():
    assert demo_calc.mul(4, 3) == 12, "Error should return 12"
    return None

def test_div():
    assert demo_calc.div(4, 3), "Error should return '1.333'"
    return None

def main():
    """
        Execute Test functions
    """
    test_add()
    test_mul()
    test_div()
    print("Everything passed")
    return None

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

Rather than keeping your assert statements in your production code, it may be wiser to have them in a separate Python file. These are called a **test case**, and you could put all the test cases for your program in this file. This method of structuring your tests is great for simple checks, but what if multiple tests fail – only the first one would be displayed.

The solution is to use a **test runner**, this is a special application for running tests, checking their output and provides additional debugging and analysis tools.

There are several test runner modules to choose from including unittest, pytest and nose2. In this session we will investigate using unittest and pytest.

QA Test Runners – unittest

The **unittest** module:

- Part of Python standard library.
- Provides a testing framework and test runner.
- Popular in commercial and open-source projects.

unittest

- Tests to be defined as methods in a class.
- Requires some knowledge of OOP.
- Uses special assertion methods.
- Inherit from `unittest.TestCase`.

```
tests\test_demo_calc.py

import unittest
from calculator import demo_calc

class TestCalc(unittest.TestCase):
    def test_add(self):
        self.assertEqual(demo_calc.add(4, 3), 7, "Should be 7")
        self.assertEqual(demo_calc.add(4.2, 3.5), 7.7, "Should be 8.7")
        return None

    def test_mul(self):
        self.assertEqual(demo_calc.mul(4, 3), 12, "Should be 12")
        self.assertEqual(demo_calc.mul(102, 3, -2), -612, "Should be -612")
        return None

    def test_div(self):
        self.assertEqual(demo_calc.div(4, 3), 1.333, "Should be 1.333")
        return None

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

The `unittest` module has been a member of the Python standard library since 2.7 and is popular for providing a testing framework and test runner for commercial and open-source projects.

It does require a little knowledge of OOP as the tests are defined as methods in a class. The class must inherit from the **unittest.TestCase** class imported from the module **unittest**. This new class inherits special assertion methods for writing test cases.

The previous test cases can be converted using the following simple steps:

1. Import `unittest`.
2. Create a new class `TestCalc`, that inherits from `unittest.TestCase` class.
3. Change built-in assert statements to `unittest` assertions.
4. Prefix assertion statements with `self` (it's an OOP thing).
5. Execute `unittest.main()` rather than `main()`.

QA Test Runners – unittest and assertions

unittest Assertions:

- Numerous methods to assert test on..
- values
- types
- existence of variables
- Name file starting with 'test'

Writing assertions:

- Tests should be repeatable.
- Deterministic/predictable.
- Relate to your input data.

In this example, we are testing for the function raising an exception.

Method	Equivalent to
.assertEqual(a, b)	a == b
.assertTrue(x)	bool(x) is True
.assertFalse(x)	Bool(x) is False
.assertIs(a, b)	a is b
.assertIsNone(x)	x is None
.assertIn(a, b)	a in b
.assertIsInstance(a, b)	isinstance(a, b)
.assertRaises(a, *args)	a == Exception

```
def div(x, z):                                     test_calc4.py
    """ Returns x divided by z, as a float. """
    if z == 0:
        raise ZeroDivisionError("Divisor must be zero")
    return float(f"{x/z:.3f}")

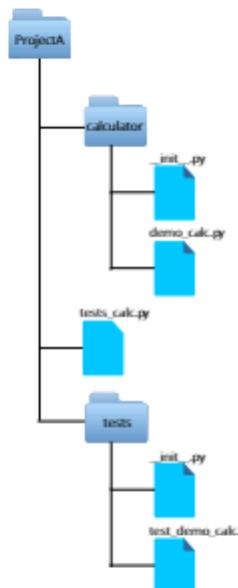
def test_div(self):                                 test_demo_calc2.py
    self.assertRaises(ZeroDivisionError, demo_calc.div,
4, 0)
    return None
```

The unittest module provides many more assertions and features over the built-in assert() function. There are assertions for testing that values are Equal or NotEqual, variables exist or of a particular type, are Boolean or raise Exceptions.

Import unittest at the REPL prompt and use the help(unittest) function to see the complete list of assertions and how to use them. The slide shows an example of testing whether a ZeroDivisionError is raised in your code or not.

When writing test assertions, it is good practice to design them so that they are repeatable, deterministic (always the same result for the same input test) and relate to your input data.

QA Test Scripts – location and executing tests



Standard test cases:

- Located in folder above application folder.
- Needs to be able to import application.
- Name file starting with 'test'.

Complex test cases:

- Create a sub package called tests.
- Split tests into files starting with 'test*.py'.

Running a single test module:

```
C:\_\\ProjectA> python -m unittest tests.test_demo_calc
```

Running a single test case:

```
C:\_\\ProjectA> python -m unittest tests.test_demo_calc.test_add
```

Running all tests:

```
C:\_\\ProjectA> python -m unittest discover
```

21

When writing simple test scripts, locate them in the folder above the application folder as they will need to import the application. Give the script a name starting with 'test'.

Once the script becomes too large than it is advisable to create sub package called tests and split the test script into separate files; the convention is to give the file names starting with 'test_'. The tests package folder should be in your main project folder.

Remember the `__init__.py` file indicates that the calculator and tests folders can be imported as a module from the parent directory.

The test scripts can be executed from the command line in several ways by importing the **unittest** module and then the test module or test function as an argument. Alternatively, if you have named your folder and scripts with the prefix 'test', then you can tell the unittest module to **'discover'** and execute all test folders and scripts.

QA Test Scripts – executing tests in pycharm

Executing your test script in Pycharm:

- Right-click>Run Python Tests>script.
- Or Menu>Run>Run Python Tests>script.
- Test runner executes your test code.
- Test result in console.



The screenshot shows the PyCharm interface with a test script named `test_demo_calc2.py`. The code defines a class `TestCalc` with three test methods: `test_add`, `test_mult`, and `test_div`. The `test_div` method includes a check for a zero division error. The PyCharm Run tool window at the bottom shows the command `python -m unittest test_demo_calc2.TestCalc` was run, and the output indicates 4 tests passed, with a large red 'PASSED' stamp.

```
class TestCalc(unittest.TestCase):
    def test_add(self):
        self.assertEqual(demo.calc.add(4, 3), 7, "Error, should be 7")
        self.assertEqual(demo.calc.add(4.0, 3.5), 7.5, "Error, should be 8.7")
        return None

    def test_mult(self):
        self.assertEqual(demo.calc.mult(4, 3), 12, "Error, should be 12")
        self.assertEqual(demo.calc.mult(100, 3, -2), -600, "Error, should be -600")
        return None

    def test_div(self):
        self.assertRaises(ZeroDivisionError, demo.calc.div, 1, 0)
        return None

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

Run → Python Tests for File... → test_demo_calc2.py
C:\Labs\projects\ProjectX\venv\Scripts\python.exe
Testing started at 13:16 ...
Ran 4 tests in 0.002s
OK
Launching unittests with arguments python -m unittest test_demo_calc2.TestCalc
Test passed 4 (moments ago)

PASSED

To execute your test script, either right-click and select Run Python Tests, or select Run menu option and click Run Python Tests. This will execute `unittest.main()` which is the Test Runner Application which will interpret and execute your tests and display the results of all tests to the console.

QA Test Scripts – executing tests in pycharm

Edit your Test script to report failures.

Execute the Test script.

In the output:

- Execution results.
- Number of failures.
- Detailed about each failed entry.
- Traceback to failed line of code.
- The assertion and expected and actual result.

The screenshot shows the PyCharm interface with a red arrow pointing from the text "Edit your Test script to report failures." to the code editor. The code editor contains a Python test script with two test cases: `test_div()` and `test_div_zero()`. The first test fails with an assertion error, and the second test fails with a `ZeroDivisionError`. A red circle highlights the failure message in the code editor. The bottom right corner of the PyCharm window has a large red stamp that says "FAILED". The "Test Results" tool window on the left shows the test results, indicating 2 failed tests. The console at the bottom shows the detailed failure messages for each test case.

```
def test_div():
    assertEqual(demo_calc.div(4, 5), 1.3333, "Error, should be 1.333")
    return None

def test_div_zero(self):
    self.assertRaises(ZeroDivisionError, demo_calc.div, 4, 0)
    return None

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

Test Results

TestFailed: 2 passed: 0

Failure

File "C:\Users\user\OneDrive\Documents\PycharmProjects\test\test_demo_calc2.py", line 20, in test_div
 self.assertEqual(demo_calc.div(4, 5), 1.3333, "Error, should be 1.333")
AssertionError: 1.3333 != 1.3333 : Error, should be 1.333

Edit your Test script to force it to fail. On the slide we have altered the expected result from the division to be 4 decimal places, rather than 3. And have changed the parameters so that a `ZeroDivisionError` is not raised.

Execute the test runner again and notice the execution results in the console, the number of failures and detail about each failed entry. A traceback to the failed line of code and the expected and actual result of the test will also be displayed. In the navigation test window, you can choose the entire Test Results or individual Test Cases.

QA Testing using pytest

Unit Testing:

- Boilerplate code.
- Using the **pytest** module.
- Common data for testing using fixtures.
- Varying data for testing using parametrization.



24

As good as the **unittest** module is, it still has some shortcomings including its interface. There are several alternative testing frameworks and **pytest** is one the most popular. It is feature rich, has additional plug-ins and is more pleasing to use. It can even run your existing tests out of the box including those written using unittest.

In this session, we will examine, how to create a test runner using Pytest.

QA Boilerplate Vs elegance

Tests should be readable and consist of minimal information to understand the test case.

unittest

- Part of the Python standard library
- Boilerplate code
- Repeated verbose code
- Import module, create class, inherit from TestCase
- Write special methods for each test case
- Use one of the many self.assert* methods

pytest

- Must be installed first
- Simple test functions (no classes or methods required)
- Uses built-in assert() function or any expression that evaluates to True
- Elegant and simple, and nicer output

In computer programming, boilerplate code is often used to describe code that is verbose in nature, that is repeated often with minimal code. Did you notice anything whilst using unittest?

Before we did any assertions, we had to import the **unittest** module, create a new class, inherit from TestCase – and then write a method for each test case using one of the many self.assert* methods.

And we must do that for each test script! Some would say that is unreadable, inelegant, or bordering on boilerplate code!

Alternatively, we could use the **pytest** module that does away with all that and allows us to write simple test functions and use the built-in assert() function or Boolean statements. If you are familiar with assert() then the learning curve is shorter than having to learn and use unittest.

The only convention you must follow is to prefix your test modules with the name 'test'. It also has nicer output!

QA Installing pytest

Pytest is not part of the Python standard library, so needs to be installed.

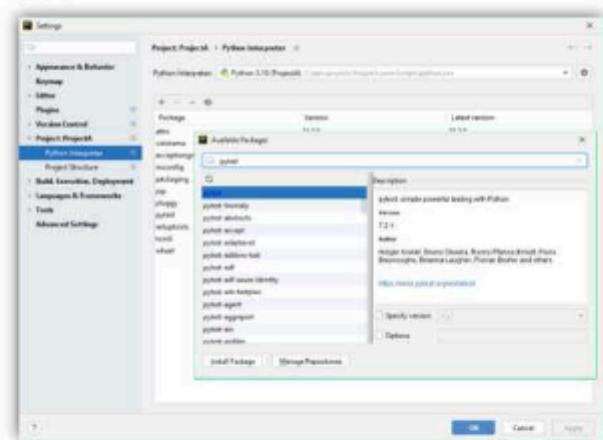
Command line

- Can be installed in a virtual environment (venv) using pip.
- Change into your project folder.

```
c:\labs\projects\ProjectA> python -m venv venv  
c:\labs\projects\ProjectA> .\venv\Scripts\activate  
(venv) c:\_\_\_ProjectA> python -m pip install pytest
```

Pycharm

- File > Settings > Project > Project Interpreter
- Select +
- Search pytest
- Select Install pytest



Pytest needs to be installed first as it is not in the Python standard library. It can be installed from the command lines using pip or from within Pycharm. It can also be installed in a virtual environment.

Pytest: <https://docs.pytest.org/en/7.2.x/>

As an aside, if you are not familiar with Virtual Environments then ask your instructor or use the links below. A virtual environment is a way to isolate a Python interpreter with its own independent set of library modules and versions; so that different projects can manage their own dependencies.

Several Virtual environments are available including venv (python only), conda (python and other languages) and anaconda (commercial environment and package manager) to choose from.

Venv: <https://docs.python.org/3/library/venv.html>

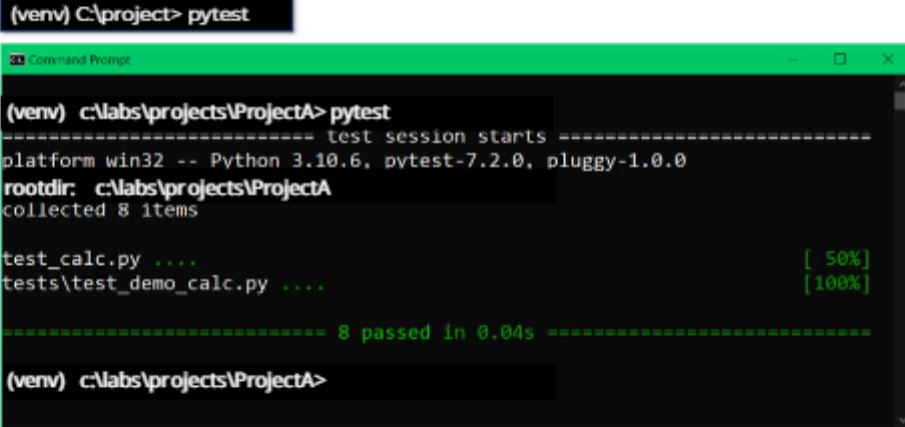
Conda: <https://docs.conda.io/projects/conda/en/stable/>

Anaconda: <https://www.anaconda.com/products/distribution>

QA pytest output

Command line – reporting success

- Execute pytest in your project folder.
- Discover test scripts.
- Nicer output – dot (test passed), F (Failed), E (Exception).



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command entered is "(venv) C:\labs\projects\ProjectA> pytest". The output indicates a successful test session:

```
(venv) c:\labs\projects\ProjectA> pytest
----- test session starts -----
platform win32 -- Python 3.10.6, pytest-7.2.0, pluggy-1.0.0
rootdir: c:\labs\projects\ProjectA
collected 8 items

test_calc.py .... [ 50%]
tests\test_demo_calc.py .... [100%]

----- 8 passed in 0.04s -----
```

The command prompt then returns to the initial state: "(venv) c:\labs\projects\ProjectA>".

Execute pytest at the command line inside your project folder and it will discover all the existing test scripts (with a 'test' prefix).

The report displays the system state, version of Python and pytest and optional plugins. The folder to search for tests and the number of tests discovered.

The output indicates a dot (test passed), an F (test failed), or an E (Exception raised). The overall progress of the test cases are displayed as a percentage.

QA pytest output

Reporting failures

```
PS C:\Users\Projects\ProjectA> pytest test_calc.py
platform win32 -- Python 3.10.6, pytest-7.2.0, pluggy-1.0.0
rootdir: c:\Users\Projects\ProjectA
collected 8 items

test_calc.py F...
tests\test_demo_calc.py .F..


===== FAILURES =====
test_add
=====
def test_add():
    assert demo_calc.add(4, 3) == 7, "Error should return 7"
>   assert demo_calc.add(10, 20, 30) == 50, "Error should return 60"
E     AssertionError: Error should return 60
E     assert 60 == 50
E     +   where 60 = <function add at 0x0000012E9FEDC7A0>(10, 20, 30)
E     +   where <function add at 0x0000012E9FEDC7A0> = demo_calc.add

test_calc.py:13: AssertionError
                               TestCalc.test_div
=====
self = <tests\test_demo_calc.TestCalc testMethod=test_div>

    def test_div(self):
>       self.assertAlmostEqual(demo_calc.div(4, 3), 1.3333, "Error, should be 1.333")
E       AssertionError: 1.333 != 1.3333 : Error, should be 1.333

tests\test_demo_calc.py:25: AssertionError
===== short test summary info =====
FAILED test_calc.py::test_add - AssertionError: Error should return 60
FAILED tests\test_demo_calc.py::TestCalc::test_div - AssertionError: 1.333 != 1.3333 : Error, should be 1.333
===== 2 failed, 6 passed in 0.06s =====
```

Edit your test scripts and add an error and rerun the pytest command. In this instance it displays that there were two failures and gives detailed breakdown of the failure, followed by an overall status report.

QA pytest output

Pycharm – reporting SUCCESS

- Right-click > Run Python Tests > script
- Or Menu > Run > Run Python Tests > script
- Pytest executes your test code

```
def main():
    """ Execute Test functions here """
    test_add()
    test_mul()
    test_div()
    # test_div_zero()
    print("Everything passed")
    return None

if __name__ == "__main__":
    main()

# vim:filetype=python
```

Python tests in test_calc2.py

test_calc2.py::test_add PASSED [25%]
test_calc2.py::test_mul PASSED [50%]
test_calc2.py::test_div PASSED [75%]
test_calc2.py::test_div_zero PASSED [100%]

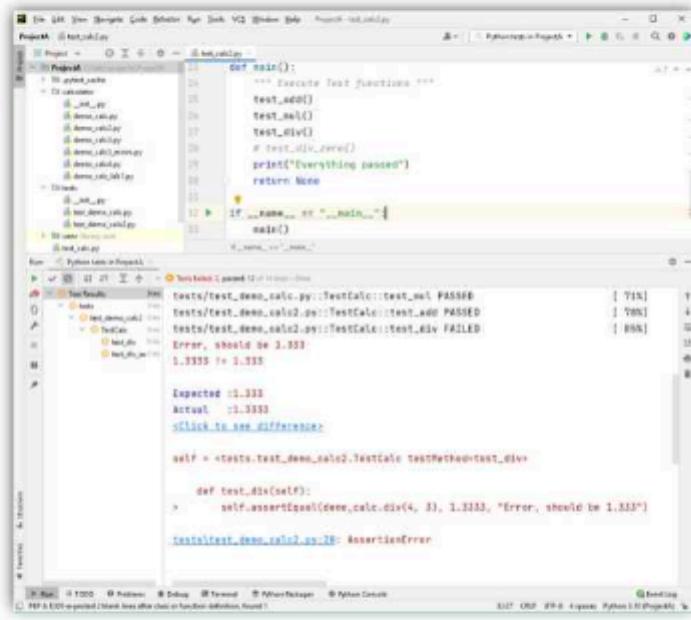
4 passed in 0.02s

Process finished with exit code 0

QA pytest output

Pycharm – reporting FAILURE

- Right-click > Run Python Tests > script
- Or Menu > Run > Run Python Tests > script
- Pytest executes your test code.
- Detailed test report in console.



```
def main():
    """ Execute test functions """
    test_add()
    test_sub()
    test_div()
    # test_div_zero()
    print("Everything passed")
    return None

if __name__ == "__main__":
    main()

# Tests
# =====
# test_add()
# test_sub()
# test_div()
# test_div_zero()

# test_main()
# test_main()

# Test Results
# =====
# TestResult: 2 passed | 2 failed | 0 skipped
# tests/test_demo_calc.py::TestCalc::test_sub PASSED
# tests/test_demo_calc.py::TestCalc::test_add PASSED
# tests/test_demo_calc2.py::TestCalc2::test_div FAILED
# Error, should be 1.333
# 1.3333 != 1.333
#
# Expected: 1.3333
# Actual: 1.3333
# Click to see differences
#
# self = <tests.test_demo_calc2.TestCalc testMethod=test_div>
#
#     def test_div(self):
#         self.assertAlmostEqual(calc.div(4, 3), 1.3333, "Error, should be 1.333")
#
# tests/test_demo_calc2.py:28: AssertionError
```

When tests fail, Pytest will generate a detailed report in the console. In this slide, we have added two obvious errors to the `test_add()` and `test_div()` and both are reported followed by a summary that 2 failed and 6 tests passed.

Pytest can run tests in parallel and can also filter tests by directory, pattern matching the filenames or by category of tests.

QA

SUMMARY



Software testing is important!

- Bugs are expensive to fix in production.
- Could save reputation and lives.
- Could improve performance and security.

Tests can be manual or automated

- Automated is better.

Document your code

- Use DocStrings and Doctest module.

Useful Testing Frameworks

- Write test cases.
- Create test runners.
- Unittest and pytest are popular.

Software testing is important!

QA Managing test data - fixtures

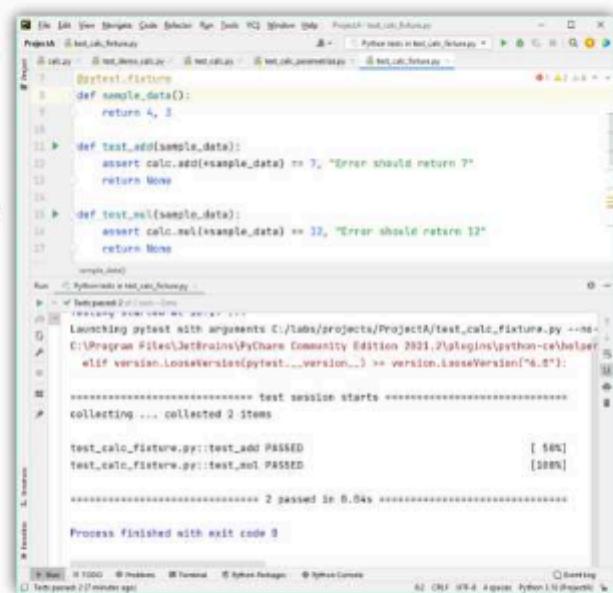
Pytest fixture

- Special function decorated with `@pytest.fixture`.
- Provides data or test double to the test function.
- Test function accepts fixture as a parameter.
- Can simplify multiple tests that use same data.
- Can be centralised in a module called `conftest.py`

```
@pytest.fixture
def sample_data():
    return 4, 3

def test_add(sample_data):
    assert calc.add(*sample_data) == 7,
    "Error should return 7"
    return None

def test_mul(sample_data):
    assert calc.mul(*sample_data) == 12,
    "Error should return 12"
    return None
```



```
def sample_data():
    return 4, 3

def test_add(sample_data):
    assert calc.add(*sample_data) == 7, "Error should return 7"
    return None

def test_mul(sample_data):
    assert calc.mul(*sample_data) == 12, "Error should return 12"
    return None
```

Running Pytest in test_calc_fixture.py

Launching pytest with arguments C:/labs/projects/ProjectA/test_calc_fixture.py --no-capture

C:\Program Files\JetBrains\PyCharm Community Edition 2021.2\plugins\python-ce\helpers\pydev\etc\version.LooseVersion(pytest...version...) >= Version.LooseVersion("6.0"):

----- test session starts -----

collecting ... collected 2 items

test_calc_fixture.py::test_add PASSED [50%]

test_calc_fixture.py::test_mul PASSED [100%]

----- 2 passed in 0.04s -----

Process finished with exit code 0

If you have many tests that share the same test data, then you can create a special function in **pytest** called a fixture. Fixtures are functions that can return a large range of values for your test functions. Like a stunt double, they provide test doubles to your test function, and the test function accepts them explicitly as parameters. In some cases, they can simplify multiple test functions and reduce the amount of boilerplate code.

If you want to make a fixture available to your entire project, then you can place them in a special module called `conftest.py`. Pytest looks for this module in each directory, but if placed in the project parent directory then it will be available to the parent and all sub directories without importing it.

QA Managing multiple test data - parametrized

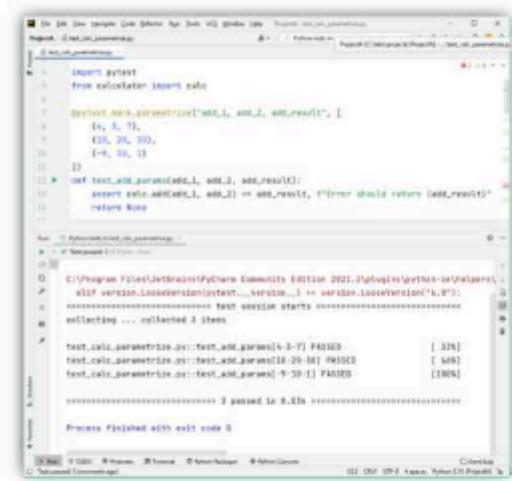
Pytest – combining tests

- Solution to different inputs and outputs.
- Generalises the code.
- Scales better.
- Uses `@pytest.mark.parametrize()`

```
@pytest.mark.parametrize("param1, param2, result", [ values, values ])
```

```
import pytest
from calculator import calc

@pytest.mark.parametrize("add_1, add_2, add_result", [
    (4, 3, 7),
    (10, 20, 30),
    (-9, 10, 1)
])
def test_add_params(add_1, add_2, add_result):
    assert calc.add(add_1, add_2) == add_result, f"Error should return {add_result}"
    return None
```



```
def test_add_params(add_1, add_2, add_result):
    assert calc.add(add_1, add_2) == add_result, f"Error should return {add_result}"
    return None
```

```
test.calculator_py::test_add_params[4-3-7] PASSED [ 3%]
test.calculator_py::test_add_params[10-20-30] PASSED [ 6%]
test.calculator_py::test_add_params[-9-10-1] PASSED [100%]
```

```
Process Finished with exit code 0
```

Fixtures can be useful to reduce duplication in code but are limited to common parameters and outputs. If you have test data with slightly different inputs and outputs, then **pytest** allows you to combine and generalise the test data using `parametrize()`.

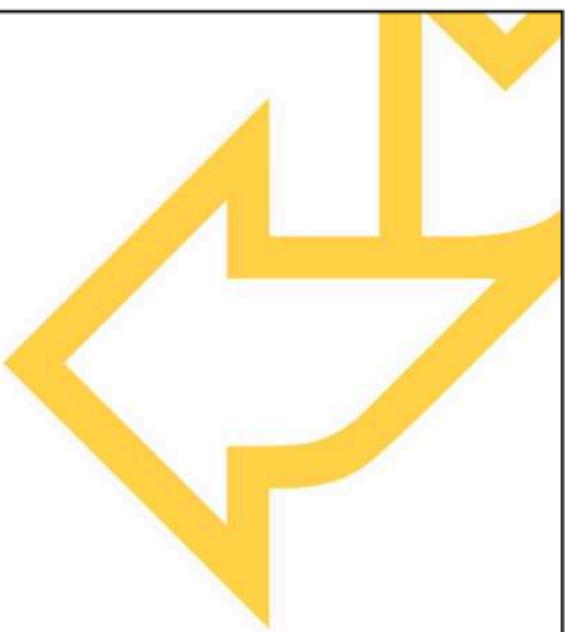
The first parameter to the `parametrize()` function is a comma-delimited string of parameter names which incorporates the input data and expected output. The second parameter is a list or tuple of the values to be tested.

A note of caution. Do not over complicate your parametrization so that it tends to boiler code. Test should be clear and simple.



Python 3 Programming

Multitasking



QA

MULTITASKING



Contents

- Family life
- Creating a process from Python
- Old interface examples
- Using the subprocess module
- subprocess.run
- The subprocess.Popen class
- Running a basic process
- Capturing the output
- Very basic threads in Python
- Using the multiprocessing module
- Queue objects

Summary

Q& Family life

A process is an instance of a program loaded and ready to run.

Every process has a parent, so every process is a child.

Child usually inherits attributes of parent.

- Environment, current directory, security.
- Open files – depends on the Python release (see notes).

Relationship depends on the operating system.

- UNIX has strong family ties:
 - If a parent dies, the child is an 'orphan'.
- Microsoft Windows has few ties between parent and child.
 - Parent must explicitly maintain a HANDLE to the child.
 - Can disown children, but still supports inheritance.

Most modern operating systems allow a user to run several applications simultaneously, so that non-interactive tasks can be run in the background while the user continues with other work in the foreground. The user can also run multiple copies of the same program at the same time.

The two key operating-system object types that have a major role to play in multitasking are the 'process' and the 'thread'. A process is an instance of a running program. Each process owns its own resources (code, data and the like) which are located in its own private address space. Any such resources created by a process are destroyed when the process terminates.

Before creating a process, we should consider the implications. Aside from the obvious performance overhead, creating another process forms a relationship between creator (the parent) and created process (the child).

Depending on the operating system, some items are inherited (copied) by the child process. On Microsoft Windows and UNIX, the

child inherits the Environment block, the current directory, and security ID. Other things may be inherited, and both operating systems allow a degree of control over what exactly is inherited.

Open files are handled differently depending on the release of Python. Prior to Python 3.4, open file handles were inherited – and this is still the case in Python 2. In Python 3.4, open handles are no longer inherited by default, see PEP 446.

QA Creating a process from Python

Process interfaces can be platform specific.

`os.fork()`

- UNIX specific.
- Creates another process - does not run another program.
- `exec` type is required to run another program after fork.
- Requires `os.wait` or `os.waitpid` to avoid a zombie.

`os.system()`

- Passes the command to the shell (`cmd.exe` or the Bourne shell).
- Runs an additional shell process.

`os.spawn` type()

- Some types are not available on Windows.
- Can run in a similar mode to `exec` type.

`os.popen()`

- Run a process connected through pipes.

All these interfaces
are deprecated



The `os` methods shown are system specific, and their behaviour varies wildly between operating systems. They reflect the differing architectures of the operating systems they run on.

On UNIX (and Linux), when we wish to run another program, we first have to create a copy of the *current* process - current program and all. That action is known as a **fork**. In the copy (known as a *child process*), we can overlay the current program with a different one, and this is known as an **exec**. There are various forms of **exec**, depending on whether you wish to supply a different environment block, use the PATH environment variable to find the program, and the way that arguments are passed.

The old DEC operating system VMS did not use that two-stage approach but created a new process running a different program in one go - an action called **spawn**. Windows inherited some of the architectural features of VMS, including **spawn** which, like **exec** on UNIX, also had different forms depending on how we wished to run the program.

Meanwhile, at an attempt to be portable, the C language standard came up with **system()**, which called a shell program to launch

another process. This was not particularly efficient and could not handle asynchronous requests.

These architectural differences are all reflected in these older interfaces, which are now considered to be deprecated, so we will not discuss them further, although, you may see them still used by die-hards.

Older versions of Python on UNIX also had the **commands** module, which was withdrawn at Python 3.

os.startfile() runs on Windows and runs the associated program on the specified file (like double-clicking on the file in Windows Explorer).

QA Old interface examples

Run a process and wait for it to complete.

- Invokes a surrogate shell.

```
import os
status = os.system("hello.py")
print("Child exited with", status)
```

Run a process at the other end of a pipe.

- Returns a file object.

```
for line in os.popen("tasklist").readlines():
    print(":", line, end="")
```

All these interfaces
are deprecated



The simplest way to run another program from Python is to use **os.system**. Unfortunately, this launches a shell whether you need one or not.

os.popen has two parameters, mode which defaults to 'r', and Buffering which defaults to None. There are some portability issues with popen, and a `win32pipe` module also exists specifically for Windows.

To read stderr from popen, you will need the assistance of the shell, typically with the decoration `2>&1` (*not* UNIX csh). If you need to read stdin from another program, then open the pipe with write access. If you need to open both, stdin and stdout, then use **os.pipe**.

QA Using the subprocess module

Unifying process creation:

- Intended to replace `os.system()` and `os.spawn()`.

From Python 3.5, the preferred interface is `subprocess.run()`:

- Meant for the majority of simple tasks.

For more complex tasks use `subprocess.Popen()`:

- Returns a subprocess object.
- Parameters are discussed later.

Other shortcuts are available:

- `call` and `check_call`.
- `getoutput` and `getstatusoutput` (UNIX specific).

We discuss the `multiprocessing` package later:

- Runs processes in a similar way to threads.

At an attempt to resolve the different interfaces used, the **subprocess** module was introduced into the Python standard library at Python 2.4. It was supposed to be unifying with no operating system specific quirks - an aim not entirely achieved.

The older interfaces are still supported but should not be used for new applications. Check specifically, the *Replacing Older Functions with the subprocess Module* section in that documentation page. Since using Python 3 is an opportunity to move to new practices, this is a good time to ditch the old methods.

The `multiprocess` module in the Python Standard Library runs processes using a different approach, which we shall discuss later...

QA subprocess.run()

Run a program and wait for it to complete.

- This API was added in Python 3.5 for the majority of simple jobs.
- It is a wrapper around Popen.
- Returns a subprocess.CompletedProcess object.

```
run(*args, input=None, timeout=None, check=False, **kwargs)
```

args	Command-line to execute (a sequence)
input	Data to be passed to stdin of the program
timeout	A timeout value in seconds. Raise a subprocess.TimeoutExpired exception if exceeded
check	If True, raise a subprocess.CalledProcessError if exit code != 0
kwargs	Optional Popen arguments (see later)

The CompletedProcess object includes:

- `returncode`
- `stdout` only if routed to a PIPE (see later)
- `stderr` only if routed to a PIPE (see later)

The subprocess.run method was introduced in 3.5 to simplify the use of subprocess. It is a wrapper about Popen, which is described over, and can appear intimidating. Popen is more than most people need, and there is a requirement to call a method to wait for the process to complete. The simpler subprocess.run will wait – which is often what you need.

The timeout parameter is something which is often requested and is an add-on which is not available with Popen.

Should you need the power of Popen, then it is still there, all its parameters can be appended.

Note that the template shown is from the help text, not from the documentation.

Q& subprocess.Popen()

Much of the functionality has been moved into Popen.communicate()

- This allows for communication between processes.

```
import subprocess

p1 = subprocess.Popen('dir', shell=True, stdin=None, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
p2 = subprocess.Popen('sort /R', shell=True, stdin=p1.stdout)

p1.stdout.close()
out, err = p2.communicate()
```

QA The subprocess.Popen class parameters

args	Command-line to execute (a sequence)
bufsize=-1	Bufsize 0: unbuffered < 0: default bufsize
executable=None	Program to be executed, rarely needed
stdin=None	Handle used for stdin (can be PIPE)
stdout=None	Handle used for stdout (can be PIPE)
stderr=None	Handle used for stderr (can be STDOUT)
preexec_fn=None	Code to call before the program (UNIX)
close_fds=True	Do not inherit open file handles
shell=False	Use a shell to execute the command
cwd=None	Working directory of the child process
env=None	Environment block of the child process
universal_newlines=False	See any of '\r' or '\n' as newlines
startupinfo=None	Windows only STARTUPINFO struct
creationflags=0	Windows only creation flags

Which buffer size should I use? If in doubt, default the parameter, as usual. The default size of -1 was changed in Python 3.3.1 to use the system's default buffer size.

Open file handles (called *file descriptors* on UNIX) are inherited by a child process in many languages, and by the older interfaces (file locks are not inherited). To prevent that, we often had to resort to low-level interfaces, like **fcntl** on UNIX. Fortunately, the subprocess module switches this off by default (**close_fds**), except for stdin, stdout, and stderr, which is usually what we want.

The Windows startupinfo mostly determines features of the main window for the new process, like x/y starting position, title, etc. The creationflags cover, for example, the priority of the child process. For details of these Windows specifics, see the MSDN and the low-level C function CreateProcess.

Q^ Running a basic process

Run a process and wait for it to complete.

A shell is sometimes required:

- When using shell meta-characters.
- Wildcards, pipes, redirections, etc.
- On Windows, no file association is done unless shell=True.

```
import subprocess
proc = subprocess.run('hello.py', shell=True)
print('Child exited with', proc.returncode)
```

Don't use a shell if you don't need to.

- It can add an unnecessary overhead.

```
import subprocess
proc = subprocess.run([sys.executable, 'hello.py'])
```

Typically:
C:\Python36\python.exe

Here we look at the simplest technique to run another program, which is similar in some ways to `os.system()`. It uses `subprocess.run()`, often with just the command-line as the single argument.

The first parameter to run can be any sequence - including either a string or a list. If using a string, make sure that there is at least one space between each component. For example, the final example of the slide could be written:

```
cmd = sys.executable + ' hello.py '
proc = subprocess.run(cmd)
```

Unlike `system()`, `run` does not use a surrogate shell to run the program unless you ask it to – and this is a `Popen` parameter. You will need a shell if you require shell features, such as globbing (also known as wildcards) - why not use the Python `glob` module instead?

On Windows, users are so used to clicking on a file and expecting "it" to run the right program that they often forget who or what "it"

is. File association, associating a file extension with a particular program, is not done by the operating system, it is done by the application which launches it - Windows Explorer, or `cmd.exe` for example. The **`subprocess`** module does not do file association, so you will need a shell to do the association for you or add the program name yourself (which is more efficient). For Python programs, the full path name is conveniently in **`sys.executable`**. If you need file association on Windows, then use **`os.startfile()`**.

Q& Capturing the output

Use the returned CompletedProcess object:

- Includes stdout, and stderr, but only when using PIPE.
- Use `bytes.decode()` to convert bytes to string.
- Use `string.encode()` to convert string to bytes (for stdin).

```
import subprocess
proc = subprocess.run("tasklist",
                      stdout=subprocess.PIPE,
                      stderr=subprocess.PIPE)

if proc.stderr != None:
    print ("error:", proc.stderr.decode())
print("output:", proc.stdout.decode())
```

- Remember that data has to be stored in memory - too much may crash your program!

This is roughly equivalent to using `back-ticks` or `$ (command)` **substitution** in UNIX shells - capturing the stdout from the child process. It is only useful with relatively small amounts of data, since the whole output is captured in memory before we can proceed.

The CompletedProcess object includes the output from stdout and stderr, but only if the process is run using PIPE for these streams. Both are byte-streams rather than strings, so you might have to **decode** them to manipulate the data.

Q\ Passing data into stdin

Suitable for simple text

- Bytes objects only

```
import subprocess
proc = subprocess.run("stuff.py", input=b"some text")
```

- stuff.py

```
reply = input("Enter stuff: ")
print(f"<{reply}>" )
```

Enter stuff: <some text>

Passing data through stdin is less common but can be useful. The `subprocess.run` interface makes this very simple for single strings by using the `input` parameter.

Notice that native strings cannot be used - a bytes object is required.

Q A Very basic threads in Python

Python threading usually uses the **threading** module.

- Call `threading.Thread(function)`

```
from threading import Thread
import time

def my_func(*args):
    print("From thread", args)
    time.sleep(5)

th1 = Thread(target=my_func, args="1")
th2 = Thread(target=my_func, args="2")
th1.start()
th2.start()
print("From main")
th1.join()
th2.join()
```

```
From thread ('1')
From thread From main ?
('2')
```

- Or create our own class derived from `threading.Thread`

The Python **threading** module is a high-level interface based on the `thread` module. If you have used threads procedurally, for example Win32 threads or pthreads from C/C++, then you will be familiar with the procedural interface.

Alternatively, we can derive our own class from the threading base class:

```
import threading
import time
class MyThread (threading.Thread):
    def run (self):
        print ("From thread", self.name)
        time.sleep(5)

th1 = MyThread()
th2 = MyThread()
th1.start()
th2.start()
print ("From main")
```

```
th1.join()  
th2.join()
```

Python threads support various locking mechanisms: Condition, Event, Semaphore, Locks (and RLock), and Thread local data.

Take note of the output from our simple program, can you see how the output from the threads and main are interleaved? Oops!

QA Synchronisation objects in threading

Several objects are available for thread synchronisation.

Condition variables

- Similar to those used by pthreads.

Events

- Similar to those used by Win32.

Thread local storage

- Enables global variables to be local to a thread.

Locks

- Similar to a mutex, has a concept of ownership.

Semaphores

- A counting lock, e.g., allow up to 3 threads to access a resource.

Timers

- Similar to waitable timers on Win32 and interval timers on UNIX.

The **threading** module contains a ranges of objects that can be used for thread synchronisation.

Condition variables come from the POSIX pthreads runtime library and include a Lock to protect a predicate. They are generally more complex to use than Events.

Events are very easy to use. Threads wait on an event, and another releases them. They include a timeout parameter.

Thread local storage is to enable a thread to share a global variable between functions, but for that variable to be different for each thread. Generally, this is a hack to allow a single threaded program to be converted to multi-threaded! Avoid global variables and you won't need to use this.

Locks are often required and represent the basic locking mechanism. A thread either has ownership of a lock or waits for it.

Semaphore on the other hand are not "owned" by anyone. We put

a limit on the number of threads that can lock a semaphore, if more come along then they wait until a thread releases the semaphore.

Timers can be useful for triggering functions at specific times, or in specific intervals.

Q^ Simple use of lock

To fix the print issue, and to protect a global list:

```
from threading import Lock
csScreen      = Lock()
csSharePrices = Lock()

dSharePrices = []

def GetStockPrice():
    global dSharePrices

    csSharePrices.acquire()
    dPrices = dSharePrices[:]
    csSharePrices.release()
    return dPrices

def Sessions:
    csScreen.acquire()
    print("\nWaiting for requests\n")
    csScreen.release()
```

This shows the use of `Threading.Lock` to create a "CriticalSection" of code (the term *Critical Section* comes from Windows and indicates code which can only run in one thread at a time). While the lock has been acquired, no other thread can access code protected by the lock.

There are two lock objects: `csScreen` protects the `STDOUT` buffer and is acquired and released around each `print()`, and `csSharePrices` protects the global list `dSharePrices`. Notice we are taking a copy of the list by using the slice, otherwise we would be returning a reference to the list which would not be protected.

QA The trouble with threads

They are very difficult to code.

- Sharing variables requires locking mechanisms.
- Subtle timing differences can make debugging difficult.

The Python Global Interpreter Lock (GIL)

- The GIL locks the interpreter.
 - Threads are locked for 100 Ticks (about 100 byte-code instructions).
 - Simplifies and protects the interpreter.
- The GIL does not mean that:
 - Python is not multi-threaded - C modules can multi-thread.
 - You don't need to worry about locking - you certainly do!

"Multi-threading is a way of shooting yourself in both feet"

Guido van Rossum : "Unfortunately, for most mortals, thread programming is just Too Hard to get right.... Even in Python...".

The CPython interpreter, when working with pure Python code, will force the GIL to be released every 100 Ticks which equates to about hundred-byte code instructions. This means that if you have a complex line of code, like a complex math function that in reality acts as a single byte code, the GIL will not be released for the period that statement takes to run.

There is an exception though: C modules! C extension modules (and built in C modules) can be built in such a way that they release the GIL voluntarily and do their own magic.

By the way, don't think it is just Python which is affected by this kind of issue, in Ruby the GIL is called the Global VM Lock.

Because of these issues, we are not taking threading any further here.

Parallel Python is available here: <http://www.parallelpython.com>,

but not currently on Python 3.0. Another alternative is to use Stackless Python, from <http://www.stackless.com>. Originally, the Google "Unladen Swallow" project was to remove the GIL, but that extension has now been dropped.

Q&A Using the multiprocessing module

Uses processes rather than threads.

- Default number of processes is one for each core.
- Also supports process pools, and processes across systems.
- Pipes and queues for synchronised communication.

```
from multiprocessing import Process

def my_func(*args):
    print("From proc", args)
    time.sleep(5)

if __name__ == "__main__":
    p1 = Process(target=my_func, args="1")
    p2 = Process(target=my_func, args="2")
    p1.start()
    p2.start()
    print("From main")
    p1.join()
    p2.join()
```

```
From main
From proc ('2',)
From proc ('1',)
```

"

The **multiprocessing** module is suitable for sharing data or tasks between processor cores. It does not use threading, but processes instead. Processes are inherently more "expensive" than threads, so they are not worth using for trivial data sets or tasks.

Since they run in different processes, then any data items sent must use a kernel object, which again uses more system resources than using shared variables within the same address space. However, shared variables require synchronisation whether they are within the same process or not (technically, in-process synchronisation is cheaper than synchronisation between processes). We shall address that on the next slide.

You will note that the code on the slide is very similar to the threading example. The main noticeable difference (apart from the names) is the inclusion of the `if __name__ == "__main__"` test. This is there because the whole script is repeated for the child processes, much like `fork()` does things (it uses `fork` on UNIX). If you wondered, in the child processes the value of `__name__` is "`__parents_main__`". This if statement is important on Windows,

since it does not have a `fork()` but imports the entire script. So, you can get away with not having it on UNIX/Linux, but it is probably a good idea to always include it for portability. Strictly speaking, if `process` is called from elsewhere, such as a class, then it might not be required.

Just as with threading, it is common to derive a child class from `multiprocessing` in a similar way.

Q& Queue objects

Used by threads and multiprocessing.

- Provides a serialised method of communication.
- multiprocessing also supports JoinableQueue.

```
from multiprocessing import Process, Queue
import os

def my_func(*args):
    queue = args[0]

    word = ""
    while word != "END":
        word = queue.get() ←
        if len(word) == 7:
            print(os.getpid(), ":", word)
```

Get an item
from the queue

Continued on next slide...

The **Queue.queue** module supplies a serialised communication mechanism between threads and the **multiprocessing** module has them built-in. All the locking is done for us - we do not need to worry about atomicity and other nasty details - every operation is atomic.

Queues are ideal for the producer-consumer module, where one set of processes adds data items into the queue and another set removes and processes them. One of the good things about queues is that it does not matter if the data items take different lengths of time to process. Each "thread" gets the next item from the queue when it has nothing else to do.

In the (simple) example code, we just have one producer and two consumers. The code shown above is for the consumer child processes. All it is doing is printing out each 7-character word in the queue.

The multiprocessing module also supports other synchronisation primitives like events and semaphores.

Q& Queue objects example (2)

```
if __name__ == "__main__":
    queue = Queue()
    p1 = Process(target=my_func, args=(queue, "1"))
    p2 = Process(target=my_func, args=(queue, "2"))

    p1.start()
    p2.start()

    for line in open("words"):
        queue.put(line[:-1])

    queue.put("END")
    queue.put("END")

    p1.join()
    p2.join()
    print("All done")
```

Put an item onto the queue

Make sure there is an 'END' marker for each child process

Here is the main part of the example code, the producer. Something which is easy to miss is that we have to send the terminator ('END') to each consumer, otherwise one would end, and the producer would hang waiting for the other.

In this case, with a little more work, we could have used a JoinableQueue instead.

Can you think how we could distribute the workload without using a queue? We would have to divide the 'words' file into two, probably by record number, and give each half to each child. There might or might not have been an even distribution of 7-character words in each half and, depending on the processing to be done on each hit, one child could have finished a lot earlier than the other - possibly resulting in an idle processor.

QA

SUMMARY

Running a program using the older interfaces was platform specific.

- These functions are now considered deprecated.

The subprocess module, and the Popen method, provides a more unified approach.

- Although, there are still platform specific methods.

The communicate method can be used to pass data through pipes.

Threads can create more problems than they solve.

- For true multiprocessing, consider another way.

QA

ASYNCIO



Contents

- Why Async code?
- Async language
- Basics
- Synchronous Vs Asynchronous
- Waiting for external data

Summary

Q&Why async code?

asyncio is a library to write concurrent code using the `async/await` syntax

- Used as a foundation for multiple Python asynchronous frameworks that:
 - Provide high performance network and webservers.
 - Provide Database connection libraries.
 - Provide Distribution of tasks via queues.

asyncio provides a set of high-level APIs to:

- Run python coroutines and have full control over their execution.
- Perform network IO and IPC.
- Control subprocesses.
- Distribute tasks via queues.
- Synchronise concurrent code.

xx

Ironically, good async code gives the look and feel of concurrency,
but are allowing processes to run as needed

QA Async Syntax

Firstly, `asyncio` needs to be imported

```
import asyncio
```

Async statements will change the way your programs operate:

- **async** – a piece of code which tells python that the object can be used asynchronously .

```
async def test():
    some code
```

- **await** – this tells the code to pause and wait for something to happen.

```
async def test():
    await y()
    return y
```

When this await is called, the rest of the program can carry on while waiting for this coroutine to end.

Q& Simple example

```
import asyncio

async def y():
    await asyncio.sleep(2)
    return "data"

async def x():
    print("starting")
    data = await y()
    print(data)
    print("end")

asyncio.run(x())
```

starting
data
end

async tells python it can be paused

await tells the coroutine to pause
asyncio.sleep lets this pause, but other coroutines continue

Another await which lets this coroutine pause while y() runs

asyncio.run() tells python that all the async coroutines can be paused

24

Run and see the pause – link to data collection.

Q& A Synchronous Vs Asynchronous

```
import time

def count():
    print("one")
    time.sleep(1)
    print("two")

def main():
    for _ in range(3):
        count()

if __name__ == "__main__":
    s = time.perf_counter()
    main()
    elapsed = time.perf_counter() - s
    print(f"executed in {elapsed:0.2f} seconds")
```

```
One
Two
One
Two
One
Two
executed in 3.21 seconds.
```

Q& A Synchronous Vs Asynchronous

```
import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"executed in {elapsed:0.2f} seconds.")
```

```
One
One
One
Two
Two
Two
executed in 1.05 seconds.
```

26

While using `time.sleep()` and `asyncio.sleep()` may seem banal, they are used as stand-ins for any time-intensive processes that involve wait time. (The most mundane thing you can wait on is a `sleep()` call that does basically nothing.) That is, `time.sleep()` can represent any time-consuming blocking function call, while `asyncio.sleep()` is used to stand in for a non-blocking call (but one that also takes some time to complete).

Q^Waiting for external data

```
import requests
import asyncio

async def loading():
    print("Loading....")
    data = requests.get('https://jsonplaceholder.typicode.com/todos/')
    response = ""
    if data.status_code != 200:
        response = "Data Not Collected"
    else:
        response = "Data Collected"
    print(response)
    data = data.json()
    return data

async def newData():
    data = await loading()
    return data

data = asyncio.run(newData())
```

27

Type this out with the cohort, discussing the new imports and link heavily to any next steps such as frameworks (Django / Flask).

QA

SUMMARY



- `async` allows for code to be paused to allow for processes to return content.
- `await` is the keyword used to tell a coroutine to pause.
- Unlike `time.sleep()`, `asyncio.sleep()` lets other processes continue, rather than all subroutines having to wait for it to finish.
- Used in conjunction with the fetching of external data, it allows for the building of Single Page Applications within frameworks such as Django or Flask.

**Want to find out more?
QA.COM**



QALtd QA-Ltd QALtd QALimited

V2.5