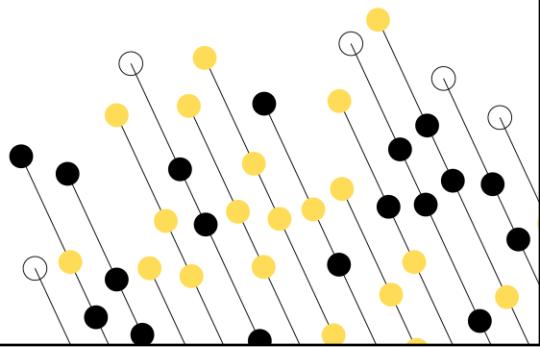


Module 1 - Introduction to Python 3

1: Introduction to Python 3

2: Fundamental Variables

3: Flow Control

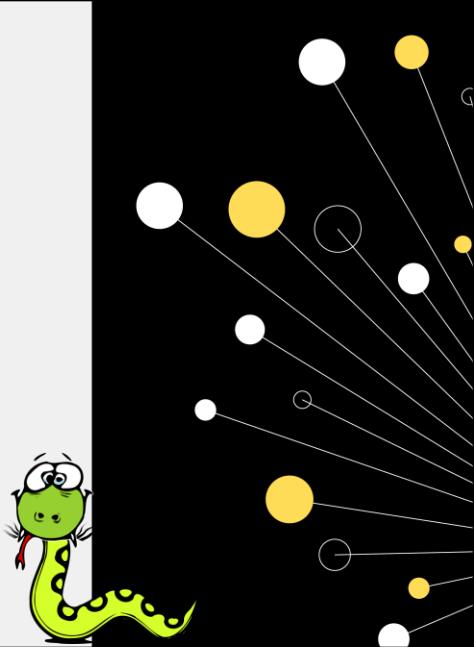


Learning objectives

By the end of this chapter, you will be able to:

- Describe what Python is and why it's widely used
- Compare Python versions and popular implementations
- Understand the philosophy behind Python via the Zen of Python
- Discuss Python's performance and community support
- Run Python interactively, via scripts, or through IDEs
- Access help, use PEPs, and explore built-in documentation
- Understand the structure of a Python script
- Use modules, functions, and built-in features effectively

QA



This chapter introduces **Python 3** and explores why Python continues to be one of the most popular programming languages. We'll look at its key benefits and the 19 guiding principles—known as the Zen of Python—that shape the language's design.

Python is a **general-purpose, object-oriented scripting language** with multiple interfaces, including user-friendly web-based environments like **Jupyter Notebooks**. As an extensible language, it supports a wide range of libraries and modules, making it ideal for developing fast and efficient solutions.

We'll focus on the **basics** in this chapter—but that's all you need to start your Python journey.

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

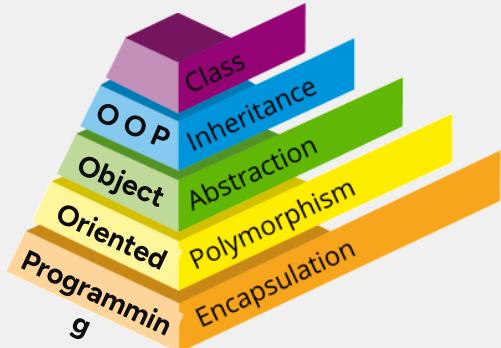
It is free

- Open source: Python license is less restrictive than GPL

It is portable

- UNIX, Linux, Windows, OS X, Android, etc...

QA Ported to the Java and .NET virtual machines



Python is a **clean, object-oriented scripting language** designed for readability and ease of use. It was originally developed by **Guido van Rossum** in the late 1980s and has since evolved through contributions from a large global community.

The name "Python" doesn't come from the snake, but rather from **Monty Python's Flying Circus**, a British comedy series that van Rossum enjoyed. This light-hearted and playful origin reflects the language's emphasis on simplicity and fun.

While **Python 2** was copyrighted by BeOpen.com and Stichting Mathematisch Centrum in Amsterdam, Python 3 is maintained and copyrighted by the **Python Software Foundation** (founded in 2001). Python is **free and open source**, but it is **not controlled by the Free Software Foundation** and does **not include any GNU code**, although it is GPL-compatible and can interface with GNU components if needed.

Useful links:

- 🔗 Python: <http://www.python.org>
- 🔗 Jython: <http://www.jython.org>

 IronPython: <http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython>

Python Versions

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

QA



4

The most noticeable change in Python 3 is that **print** is no longer a statement—it's now a **built-in function**. This means parentheses are required, e.g., `print("Hello")`. For those familiar with Python 2, this can take some getting used to.

While some **Python 3 scripts may run on Python 2.6+**, and vice versa, **this is purely coincidental** and should not be relied upon. Compatibility is not guaranteed.

From version 2.6 onward, many Python 3 features were **backported** into Python 2. Tools like the -3 warning flag and the **2to3.py** conversion utility were introduced to ease the transition. However, **Python 2 reached end-of-life** with the final release of 2.7 on July 4, 2010. Maintenance officially ended in 2019.

Guido van Rossum, Python's creator, stated - “If you're starting a brand-new thing, you should use 3.0.”

That said, due to performance concerns, **Python 3.0 was short-lived**, quickly replaced by version 3.1 within months. Python 3 is now the standard.

The king is dead. Long live the king!

Python Implementations

Several Variants of Python

- Most common is written in C and known as CPython
- Java based version named Jython
- C# (.Net) based version named Iron Python

Python based version: PyPy

- Just-in-time compiler, fewer threading issues
- Remarkable performance improvements x3
- Latest version is v7.3.10 (Python 3.10)
- Might be the default Python one day

Platform Agnostic

- No platform specific functions in the base language

QA



5

There are several versions of Python, each differing in subtle ways. The most widely used—and the one assumed for this course—is **CPython**. Written in C for speed and portability, CPython runs on many platforms. You can even download and compile the source code yourself. The latest version is **3.13.3** (as of April 2025).

The Java-based implementation, Jython, uses Java's native services and is well-suited for integrating with Java systems, as it can directly use Java classes. However, Jython has not yet released a version compatible with Python 3.

IronPython serves a similar role for .NET environments. Its latest stable version is **3.4.2**, compatible with Python 3.4, and targets .NET Framework 4.6.2, .NET Core 3.1, and .NET 6.

All these implementations—CPython included—have active developer communities that share ideas and occasionally even code. A pure Python virtual machine, **PyPy**, is also under development, and other VMs have benefited from its innovations.

Python isolates platform-specific features in the `os` module, which is deliberately non-portable. Despite this, cross-platform compatibility is never fully guaranteed—differences like filename syntax still vary between UNIX, Windows, and macOS.

Why use 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

QA Libraries may be coded in C or C++



6

Like most modern languages, Python has a garbage collector—so we don't need to worry too much about memory management. Unicode refers to multi-byte character sets, essential for displaying international characters, the Euro symbol (€), and even emojis. It supports virtually every human language, making Python web-ready and globally compatible.

One of the strengths of community-driven projects like Python is the wide range of free add-on libraries. These include powerful tools like NumPy for numerical computing and DISLIN for data visualization.

Python also has strong support for JSON (JavaScript Object Notation), widely used in web development as a lightweight alternative to XML. Many web frameworks and APIs—such as those for Twitter—rely heavily on JSON.

These add-ons are possible thanks to Python's ability to interface with C. Most APIs are designed in C, and Python can use C-written libraries. To integrate third-party software, developers often create a C-Python wrapper around it.

Zen of Python

19 Guiding Principles

```
IDLE Shell 3.10.6
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!
>>>
```

QA

...and some entertainment

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



7

It would be remiss not to mention the **this** module—a playful Easter egg that reveals 19 guiding principles by Tim Peters, known as The Zen of Python. These principles offer insight into the philosophy that shapes the Python language.

To view them, try this at the IDLE prompt:

```
>>> import this
```

For some light-hearted fun, consider downloading **Pygame** and diving into classics like Flappy Bird, Super Potato Bruh, Sudoku, and of course, Snake! You can even create your own games.

Visit www.pygame.org — “Takes the C++ out of Game Development.”

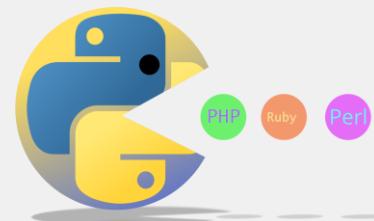
Python Performance

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

But...

- Better memory management
 - Equivalent to Perl and Ruby - depending on the benchmark
 - Better than PHP and better than Tcl
- QA
- Packages often use compiled extensions
 - Google Unladen Swallow project improved the base



8

Like many scripting languages—such as Java, C#, Perl, Ruby, and PHP—Python is compiled at runtime into an internal format known as bytecode. It's unrealistic to expect this to match the speed of C programs compiled into native executables like PE COFF (on Windows) or ELF (on UNIX). Nevertheless, some impressive benchmarks show that Python's performance can be surprisingly competitive in certain scenarios.

It's also worth noting that writing fast, safe, and efficient C code requires significant expertise, whereas Python enables safer code development with less effort and faster turnaround. As we'll see later, Python modules are usually compiled into bytecode only once and then stored—though this bytecode is not portable across systems. You can find more benchmark information at: <http://shootout.alioth.debian.org/gp4/python.php>

IronPython and Jython generally perform similarly to CPython, and all of these VM implementations are under continuous development with ongoing optimizations. Unladen Swallow project, named after a famous line from Monty Python and the Holy Grail, was merged into the mainline and contributed to performance enhancements (see PEP 3146).

Python 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 & Interest Groups

- **PyCon** - EuroPython moves around European cities

QA U.S.A. - PIGgies



9

The Python Software Foundation (PSF) website is the go-to starting point for anything related to Python. It offers a wealth of resources, including downloads, tutorials, documentation, and a wide range of modules.

Python enthusiasts gather at various conferences held around the world each year, including EuroPython and PyCon. You can find details about these events via the python.org website.

Please note that “Python” and the Python logo are registered trademarks of the Python Software Foundation.

Interacting with a Python Session

Command Line

- Interactive
- Available in Windows and Linux Shell
- Limited Debugging
- Interactive help() and documentation

```
c:\labs>set PATH=%PATH%;C:\Program Files\Python\Python313\

c:\labs>python
Python 3.13.1 (tags/v3.13.1:0671451, Dec  3 2024, 19:06:28) [MSC v.194
2 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
>>> 41 + 1
42
>>> quit()
c:\labs>
```

IDLE – Integrated Development Learning Environment

- Graphical, Interactive and portable
- Limited debugging features
- Interactive help() and documentation
- Built-in text editor with syntax highlighting
- Command History – Previous=Alt-p and Next=Alt-n

```
File Edit Shell Debug Options Window Help
Python 3.13.1 (tags/v3.13.1:0671451, Dec  3 2024, 19:06:28) [MSC v.1942 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello world")
Hello world
>>> 41 + 1
42
>>> quit()
>>>
```

Your program is still running!
Do you want to kill it?

OK Cancel

10

QA

When Python is launched interactively from a shell—whether it's a UNIX-style shell like Bash or the Korn shell, or Microsoft's cmd.exe—you're presented with the Python shell prompt: `>>>`. This is often called the **REPL** prompt, which stands for **R**ead, **E**valuate, **P**rint, **L**oop. It allows you to quickly test or evaluate Python code before incorporating it into a script.

The REPL continues running until you call the `exit()` or `quit()` functions, or press `<CTRL+Z>` on Windows or `<CTRL+D>` on UNIX/Linux systems.

One of the most useful initial commands is the built-in `help()` function, which provides interactive documentation and guidance. Python can be configured using optional environment variables, though these are rarely needed. Full details can be found in the official documentation.

IDLE is a simple GUI for writing and debugging Python code. It comes bundled with Python and is handy for development. (And yes—Eric would approve: “Nudge nudge, wink wink, say no more!”). Note that IDLE may encounter issues with personal firewalls. On Linux, it also requires Tk to be installed.

On macOS, IDLE controls differ slightly:

- <CTRL+P> – Previous statements
- <CTRL+N> – Next statements

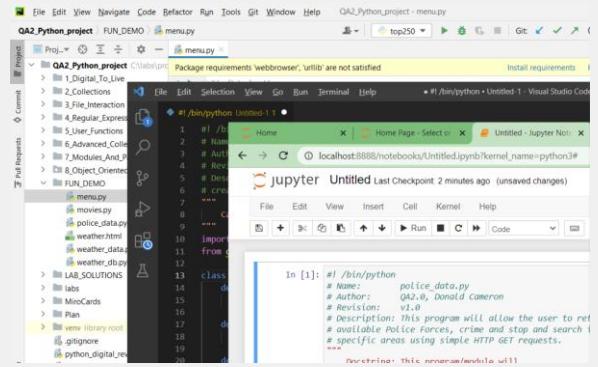
IDEs – You have a choice!

Integrate Development Environments

- IDLE, Pycharm, VSCode, Eclipse..
- User Friendly GUI
- Built-in text editor and debugger
- Integrated console
- Coloured Coded highlighting
- Code Search
- Integrate with Online Repository and CI/CD

Web Based IDE – Jupyter Notebooks

- Supports 100+ languages
- Suited for Data Science and Machine Learning



QA

11

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

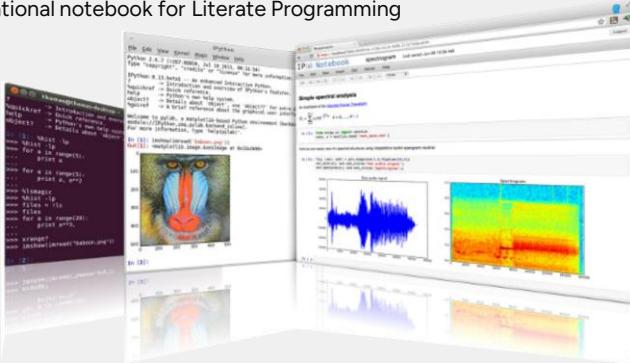
Even beginner Pythonistas often transition quickly to more feature-rich IDEs such as **PyCharm** or **VS Code**. These environments offer powerful editors, integrated consoles, advanced debugging tools, and seamless integration with version control systems and online repositories like GitHub and Bitbucket.

Additionally, **web-based interfaces** such as **Jupyter Notebooks** are popular—particularly within the data science community—for their ability to combine code, visualisations, and narrative text in a single interactive document.

IDEs - IPython and Jupyter

Interactive Computing and Data Visualisation

- Jupyter Notebooks – fork of IPython (2014)
- Named derived from Julia, Python and R
- Computational notebook for Literate Programming



QA

"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

12

IPython, or Interactive Python, is a powerful command-line shell designed for interactive computing. It supports **introspection** (examining object types and properties), rich media (text, images, graphs), **shell syntax**, **tab completion**, and **command history**.

IPython also offers a **browser-based notebook interface** that allows the integration of code, text, mathematical expressions, and interactive data visualisations such as plots. It's widely used alongside math and data science libraries like **NumPy**, **SciPy**, and **matplotlib**.

In 2014, IPython creator **Fernando Pérez** launched a spin-off project: **Project Jupyter**. IPython now functions as the **kernel and shell** within Jupyter's notebook-style interface. Jupyter is **language-agnostic**, and its name reflects the core languages it originally supported: **Julia**, **Python**, and **R**.

As a result, the **Jupyter Notebook** (formerly the IPython Notebook) has become a web-based, interactive environment for writing, running, and visualising code. It supports over **100 kernels**, with IPython being one of them. This approach aligns with the Literate Programming style first introduced by Donald Knuth in

1984.

Python Scripts

Python command line options

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

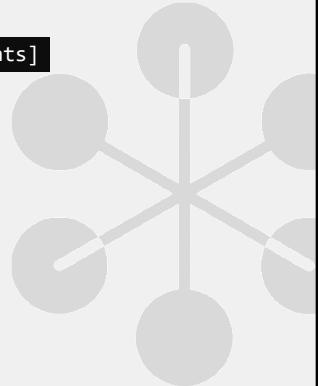
Python scripts are compiled into byte-code

- 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

```
hello.py
```

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

```
Hello world!
```



QA

13

Python programs (often called scripts) and modules typically have the **.py** file extension. When a Python module is executed, it is automatically saved in a semi-compiled bytecode format with a **.pyc** suffix to speed up future loading. This bytecode is updated if the source file has a newer timestamp but is not portable across platforms. For a full list of command-line options, use: python -h.

When naming your scripts, avoid using names that clash with existing Python modules. A helpful strategy is to adopt a naming convention—such as prefixing script names with a code or identifier. We'll explore modules in more detail later.

On UNIX and Linux, it's good practice for Python scripts to begin with a shebang line (#!), followed by the path to the Python interpreter. This allows the script to be executed directly from the command line. While this line is ignored on Windows, it's still common to include it for portability. On UNIX-like systems, you'll also need to assign execute permission using **chmod u+x scriptname**. Python modules do not require a shebang and only need read permission.

The **-c** option lets you run a short Python command from the command line.

Using a dash (-) instead of a script name or -c will read code from standard input. The **-i** option runs a script and then drops you into interactive mode so you can keep using the script's variables or functions. The **-m** option runs a module as a script. The command **python -V** (uppercase V) displays the version of Python.

Python Help

Online Help

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

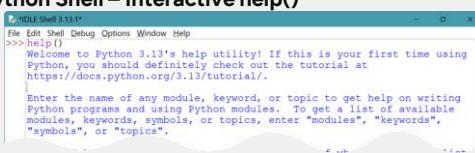
UNIX/Linux man pages

```
$ man python
```

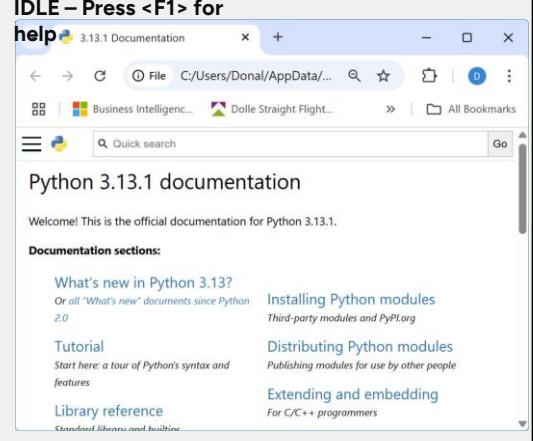
Python pydocs

```
C:\ pydoc sys
```

Python Shell – Interactive help()



IDLE – Press <F1> for help



The screenshot shows the Python 3.13.1 Documentation page. At the top, it says "IDLE – Press <F1> for help". Below that is the title "Python 3.13.1 documentation". The page content includes sections like "What's new in Python 3.13?", "Documentation sections", and links to various Python modules and features.

QA
14

Python documentation is readily available online. For example, the module reference can be found at:

<http://www.python.org/doc/current/lib/modindex.html>

And the Python 3 language reference is available at:

<http://docs.python.org/3.0/>

IDLE includes built-in access to documentation, and on UNIX/Linux systems, man pages are typically bundled (try `man -k python` to explore extensions).

The **pydoc** tool is a handy way to view documentation from the command line:

- On UNIX/Linux: run `pydoc module_name`, e.g., `pydoc sys`
- On Windows: use `pydoc.py` located in the Lib folder,
e.g., `C:\Python32\Lib\pydoc.py sys`

Windows releases often include a Tkinter-based GUI version of pydoc (usually in the Tools folder), though you may need to create a shortcut for easy access.

PEP – Python Enhancement Proposals

Language design document

- Evolution, Information and 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



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

QA

15

Python Enhancement Proposals (PEPs) are design documents that outline the evolution, features, and key decisions behind the Python language. They serve as a communication tool for the Python community, explaining the rationale for new features, when they were introduced, and how they were implemented.

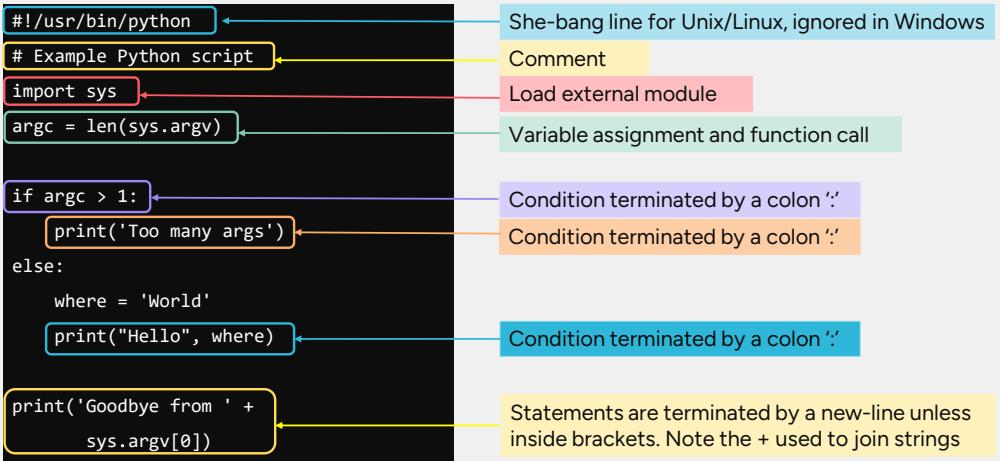
The comprehensive index of PEPs can be found at:

<http://www.python.org/dev/peps/>

They are also included in the Python 3 documentation, with many help texts linking directly to relevant PEPs.

One of the most important PEPs is **PEP 8**, the official **Style Guide for Python Code**. It defines best practices for **code layout**, **readability**, **consistency**, and overall coding conventions. PEP 8 should be read, followed, and referenced frequently to write clean and maintainable Python code.

Anatomy of a Python Script



The diagram shows a Python script with various parts highlighted and annotated:

- `#!/usr/bin/python` is labeled "She-bang line for Unix/Linux, ignored in Windows".
- `# Example Python script` is labeled "Comment".
- `import sys` is labeled "Load external module".
- `argc = len(sys.argv)` is labeled "Variable assignment and function call".
- `if argc > 1:` is labeled "Condition terminated by a colon ':'".
- `print('Too many args')` is labeled "Condition terminated by a colon ':'".
- `else:` is labeled "Condition terminated by a colon ':'".
- `where = 'World'` is labeled "Condition terminated by a colon ':'".
- `print("Hello", where)` is labeled "Condition terminated by a colon ':'".
- `print('Goodbye from ' + sys.argv[0])` is labeled "Statements are terminated by a new-line unless inside brackets. Note the + used to join strings".

QA

16

When a Python script is executed, the code is first compiled into bytecode and then run—similar to languages like Java and C#. This compilation step is fast and usually not noticeable. All the lines shown are optional, and this example omits details such as function definitions and global variables.

The first line, often called a **shebang**, tells UNIX which interpreter to use. It begins with `#`, which is also a comment—so on Windows systems, it's ignored.

The **import** statement loads an external module. This is typically placed near the top of the script. In this case, `sys` is imported to provide access to system-specific parameters and functions. We'll cover modules in more detail later.

A **variable assignment** is shown without explicitly declaring a type—Python handles typing dynamically. The if condition ends with a colon, and code blocks are defined by indentation. This rule applies to all control structures, including loops. The **print()** function outputs text to the standard output stream (STDOUT), usually the terminal, and adds a new line by default.

Most Python statements end with a new line. But if a new line appears within

brackets, it's treated as whitespace. You can also escape a new line using a backslash (\). Technically, you can put multiple statements on one line separated by semicolons—but just because you can, doesn't mean you should. But we can C what you are doing and will disapprove!

Modules

Most Python programs load other Python code

- Standard Library modules bundled with Python
- Downloaded or modules written locally
- Import with and without Namespace prefix:

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

Examples:

- **os**: Operating system specific code
- **sys**: Interface to the runtime environment
- **NumPy, DISLIN, SciPy**: Scientific libraries
- **__builtins__**: Python's built-in functions (automatically imported)

QA

```
>>> help(__builtins__)
```



17

Each module has its own namespace, so variables and functions defined within one won't interfere with those in another, even if they share the same name.

Python comes with a rich set of **built-in** modules, and they are used frequently. A list of standard modules can be found in the online documentation. Additional modules can be downloaded from <https://pypi.org/> - check the sidebar for those specifically compatible with Python 3.

Python locates its modules from directories listed in the **PYTHONPATH** environment variable, or from its built-in lib directory.

To view help for a module in IDLE, type **help(module_name)**, or enter **help()** to access the help prompt. From there, type modules to see a list of available modules, and then enter the name of a module for detailed documentation.

We'll also explore how to create our own modules later on.

Function and builtins

A function is a named block of program code

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

```
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 online documentation
 - The Python Standard Library > Built-in Functions
 - Summary list at the end of this chapter



QA

18

Functions are blocks of code designed to perform specific tasks. Python includes many built-in functions that significantly enhance the language's capabilities. These functions are always available—they don't require any external modules to be imported—and execute very efficiently.

While a list of common functions is provided at the end of this chapter, the most comprehensive and up-to-date reference is the official Python documentation.

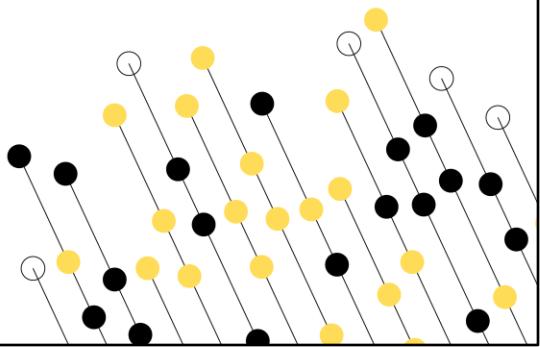
To view the full list of built-in functions directly in Python, use:

- `>>> help("builtins")`
- `>>> help(__builtins__)`

Review & Lab

Answer review questions

Your instructor will guide you which lab exercises to complete



Review Questions

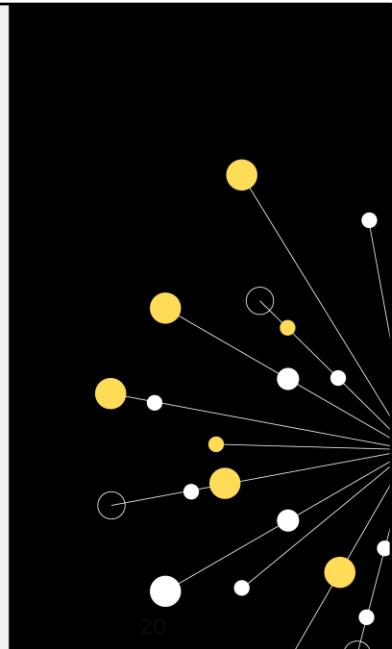
1. Which of the following is NOT a valid way to run a Python program?

- a. From the command line as a script
- b. Interactively via the REPL
- c. Using an IDE like VSCode or PyCharm
- d. Compiled as a .java file

2. What does the import statement in Python do?

- a. Compiles the script into machine code
- b. Loads a built-in function into memory
- c. Imports an external module into the program
- d. Executes the main function of the script

QA



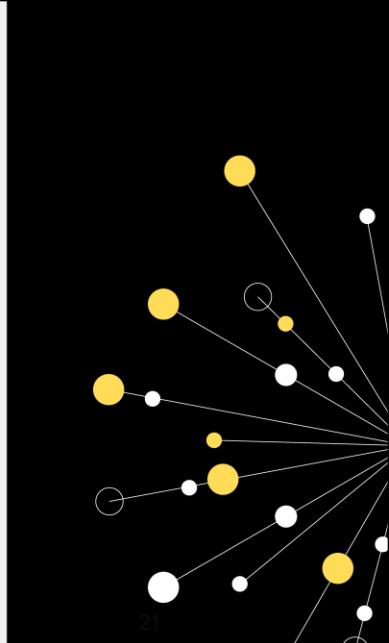
20

Answers on next page..

Review Questions

3. Why is proper indentation important in Python?
 - a. It helps the compiler optimize the code
 - b. It is required to define code blocks
 - c. It improves speed and memory usage
 - d. It's only needed for comments
4. Which of the following is true about Python's built-in functions?
 - a. They must be imported from external modules
 - b. They are slow and memory-intensive
 - c. They are available by default and run quickly
 - d. They require internet access to use

QA



21

Correct answers:

1. Which of the following is NOT a valid way to run a Python program?
 - b. Compiled as a .java file.

Explanation: Python is not compiled as .java files. It is interpreted and can be run from scripts, interactively, or via IDEs.
2. What does the import statement in Python do?
 - c. Imports an external module into the program

Explanation: The import statement is used to bring external modules or libraries into your Python program.
3. Why is proper indentation important in Python?
 - b. It is required to define code blocks

Explanation: Python uses indentation to define blocks of code (e.g., for loops, if statements, functions). It's a core part of the language syntax.
4. Which of the following is true about Python's built-in functions?
 - c. They are available by default and run quickly

Explanation: Built-in functions like print(), len(), and type() are always

available in Python and are optimized for performance.

Summary

Python is a fully functional, free programming language

Extensive online documentation is available

Python scripts can be run:

- As standalone scripts
- Interactively via the REPL
- From an IDE like VS Code or PyCharm

Python's syntax is unique and whitespace-sensitive

- Proper indentation is required — by design!

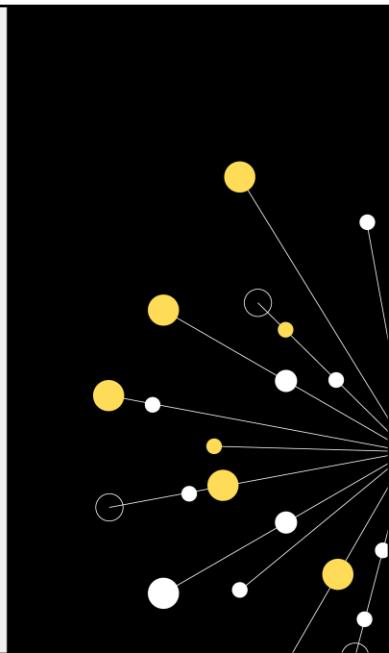
External modules are commonly used

- Load them with the import statement

Built-in functions are:

- Fast, always available and very useful

QA



Python Builtin Functions (page 1 of 2)

fn_name(args)	fn_name(args)	fn_name(args)
<code>abs(x)</code>	<code>bytearray([arg[, encoding[, errors]]])</code>	<code>dir([object])</code>
<code>all(iterable)</code>	<code>bytes([arg[, encoding[, errors]]])</code>	<code>divmod(a, b)</code>
<code>aiter(iterator)</code>	<code>callable(object)</code>	<code>enumerate(iterator[, start=0])</code>
<code>any(iterator)</code>	<code>chr(i)</code>	<code>eval(expression[, globals[, locals]])</code>
<code>anext(iterator)</code>	<code>classmethod(function)</code>	<code>exec(object[, globals[, locals]])</code>
<code>ascii(object)</code>	<code>compile(source, filename, mode[, flags[, dont_inherit]])</code>	<code>filter(function, iterable)</code>
<code>bin(x)</code>	<code>complex([real[, imag]])</code>	<code>float([x])</code>
<code>bool([x])</code>	<code>delattr(object, name)</code>	<code>format(value[, format_spec])</code>
<code>breakpoint(function)</code>	<code>dict([arg])</code>	<code>frozenset([iterable])</code>

QA

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 Builtin Functions (page 2 of 2)

fn_name(args)	fn_name(args)	fn_name(args)
getattr (object, name[, default])	isinstance (object, classinfo)	min (iterable[, args...], *[, key])
globals ()	issubclass (class, classinfo)	next (iterator[, default])
hasattr (object, name)	iter (o[, sentinel])	object ()
hash (object)	len (s)	oct (x)
help ([object])	list ([iterable])	open (file[, mode='r'[, buffering=None [, encoding=None[, errors=None [, newline=None, closefd=True]]]]]))
hex (x)	locals ()	ord (c)
id (object)	map (function, iterable, ...)	pow (x, y[, z])
input ([prompt])	max(iterable[, args...], *[, key])	
int ([number string[, radix]])	memoryview (obj)	

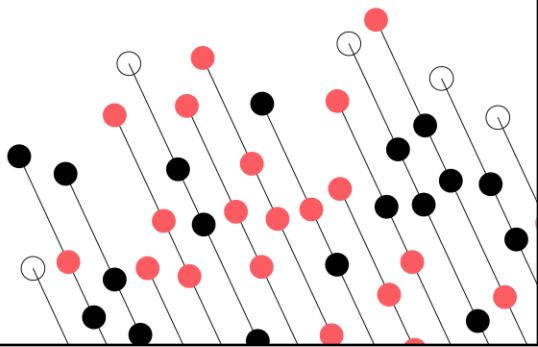
QA

Module 2 - Fundamental Variables

2: Fundamental Variables

3: Flow Control

4: String Handling

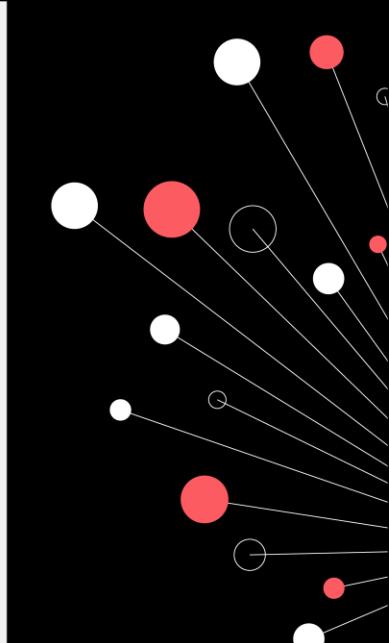


Learning objectives

By the end of this chapter, learners will be able to:

- By the end of this chapter, you will be able to:
- Understand that everything in Python is an object with a type
- Create and use variables with dynamic typing
- Apply type-specific methods (e.g. on strings, lists)
- Use augmented assignments (e.g. `+=`, `*=`)
- Work with core types: `int`, `float`, `str`, `bool`, etc.
- Create and manipulate lists and tuples
- Use dictionaries for key-value storage

QA



This chapter introduces learners to how Python handles **data and variables**, emphasising its **object-oriented nature, dynamic typing**, and **core data types**.

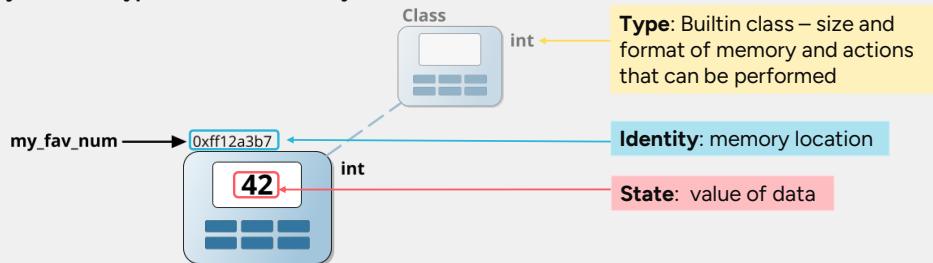
By the end, learners will understand how to **store, manipulate, and organise data** efficiently using Python's built-in tools.

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



QA

Object orientation is central to Python, and understanding its principles is key—even if you don't plan to write object-oriented (OO) code yourself. In Python, **everything is an object**, and **variables are simply names that reference those objects** in memory. The idea of a "type" in many programming languages is roughly analogous to a class in Python—it defines what kind of object you're dealing with and how it behaves. While we don't need to worry about how objects are laid out in memory, Python does—knowing the size and format is essential for it to manage memory properly.

Fortunately, Python provides many built-in classes, such as strings, files, and exceptions, so we don't always need to define our own. When we create an object, it is an instance of a class, which gives it structure and behaviour. The object is stored in memory, and we typically assign it to a variable for reuse. You can check an object's class using `isinstance()` or `type()`. For example, `isinstance(name, str)` will return True if `name` is a string. The unique identity of an object can be obtained with `id()`, though this is rarely needed.

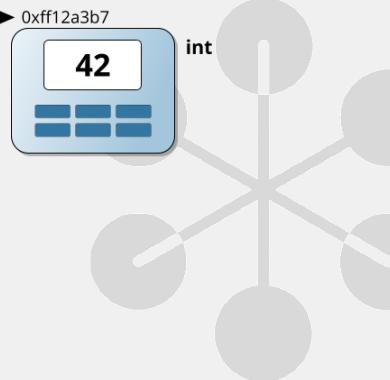
As for whether classes and types are the same—that's a bit nuanced, and we'll revisit the idea later when we explore duck typing.

Python Variables

Python variables are references to objects

```
my_fav_num = 42
```

my_fav_num → 0xff12a3b7



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 Scope – Life and Visibility:

- Local or Global – local by default if created within a user function
- Display local variables with `print(locals())`
- Display global variables with `print(globals())`

QA

28

Like many scripting languages, Python defines variables automatically and without explicit type declarations—they are untyped until assigned. In reality, variables are references to objects. Assigning a string to a variable, for example, means that variable now points to a string object. Uninitialized variables reference a special object called **None** (similar to **NULL** or **undef** in other languages).

Since everything in Python is an object, class-specific functionality—such as changing the case of a string—is accessed via method calls on the object itself. A method call works like a function call, but with the object automatically passed in as context. We'll see examples of this soon.

For immutable objects (like strings and numbers), multiple variables can reference the same underlying value without conflict.

Unlike many scripting languages, Python variables defined inside a function are local by default. If you want to modify a global variable inside a function, you must declare it explicitly using the `global` keyword—this helps avoid accidental side effects.

Finally, deleting a variable removes its name and decreases the object's reference count. When no more references exist, Python can reclaim the memory.

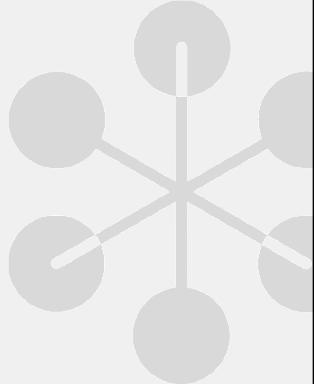
Variable Naming

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.



QA

29

Variable names in Python follow standard naming rules familiar from many programming languages. Names are case-sensitive and must not conflict with Python keywords (`help('keywords')`). Beyond the rules, Python also follows naming conventions—especially involving underscores:

- A **single leading underscore** (e.g., `_var`) signals that a name is intended for internal use within a module or class. It's not truly private (as in Java), but serves as a hint to other programmers. It also prevents the name from being imported via `from module import *`—though this practice is discouraged by PEP 8 anyway.
- A **double leading underscore** (e.g., `__var`) triggers name mangling, where the interpreter changes the name to include the class name (e.g., `_ClassName__var`). This avoids naming conflicts in subclasses and makes the attribute effectively private to the class.
- Names with **both leading and trailing double underscores** (e.g., `__init__`) are reserved for special system-defined names. You should avoid using these for your own variables or methods.

These naming conventions help improve clarity, prevent conflicts, and support encapsulation in object-oriented design. We'll explore these in examples throughout the course.

Type Specific Methods

Actions on objects are done by calling methods

- A method is implemented as a function - a named code block.
- Objects can be variables (e.g. `my_var`) or literals (e.g. "hello world").
`object.method([arg1[, arg2...]])`

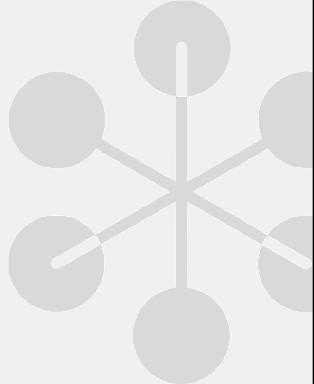
Which methods may be used?

- 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>

QA



30

Nearly everything in Python is an object, and operations or attribute queries are typically performed by calling **methods**—functions associated with objects—directly from the object itself. This approach has key advantages: for example, it ensures you're operating on the correct type, as there's no other way to access that method.

In the following examples, assume the variables are:

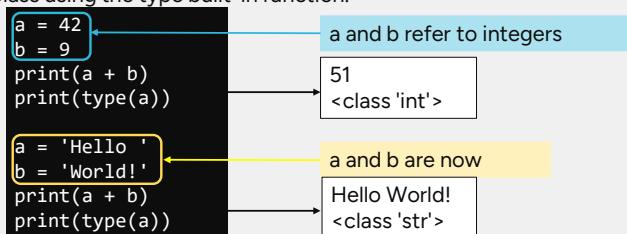
- **name**: a string object
- **names**: a list object
- **mydict**: a dictionary object
- **myfile**: a file object

The method names used are mostly self-explanatory, but full documentation is available via Python's online help. You can use `help(object)` to view the docstring and all methods of the object's class. Alternatively, `print(object.__doc__)` displays just the class-level docstring.

Operators and Type

An operator carries out an operation on an object

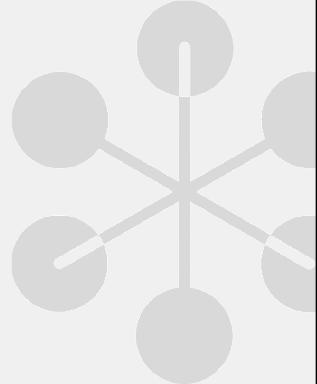
- Produces a result which does not (usually) alter the object.
- The operation depends on the Class (type) of the object.
- List the Class using the type built-in function.



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

QA

31



An **operator** is typically a symbol (see the slide after the summary) that performs an operation on one object (**unary**) or between two objects (**binary**). The result is a value that can be passed to a function (like print) or used in an assignment.

Arithmetic operators such as `+`, `-`, `*`, `/`, and `%` have familiar meanings with numbers. But what do they do with other types—like strings?

In Python, operators can behave differently depending on the **type** of object they're working with. Some meanings may be unexpected—for instance, `%` performs string formatting when used with strings. That's why it's essential to know the object type you're working with, and the `type()` function helps with that.

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.

QA

32

Augmented assignments (also known as **compound assignments** in some languages) originate from C. The expression `a += b` means "increment `a` by the value of `b`".

For example:

```
>>> stein = 1  
>>> pint = 1  
>>> stein += pint # stein becomes 2, pint remains 1
```

These operators are a concise alternative to writing:

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

Instead, we use:

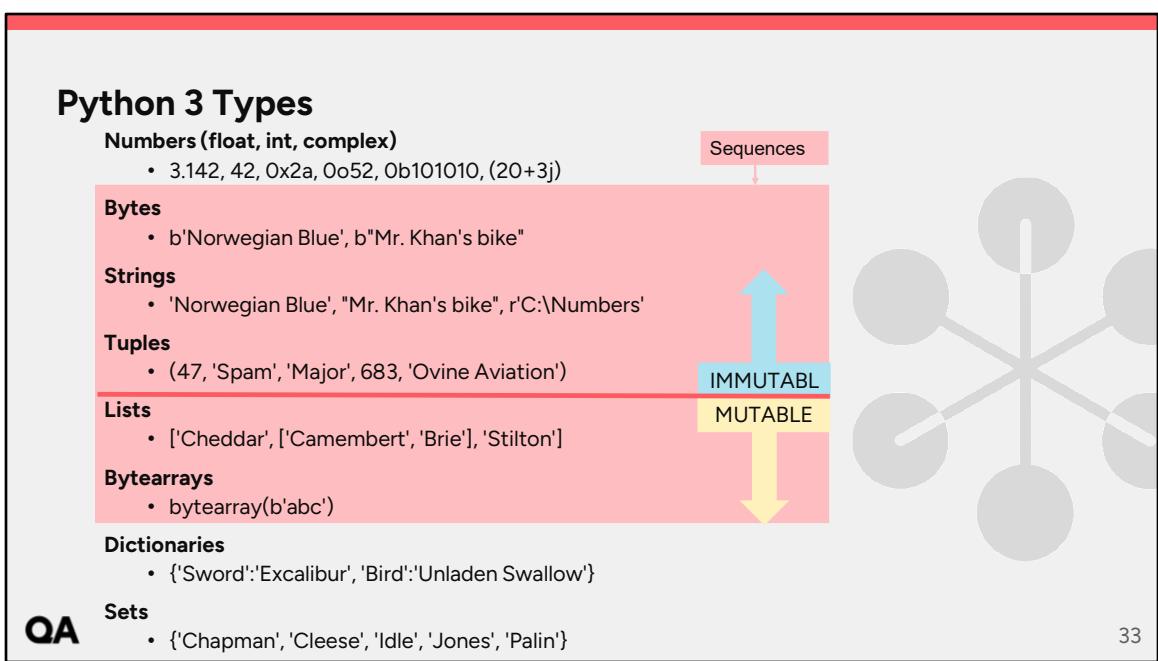
```
>>> total += subtotal  
>>> geometric *= progression
```

The meaning of these operators depends on the object type.e.g., for strings:

```
>>> a = 'Hello'  
>>> b = 'World!'  
>>> a += b # Result: 'Hello World!'
```

This creates a new string object (optimized in CPython). Python does not support

the ++ or -- operators like C. Instead use += 1 or -= 1.



Some Python types are intuitive, but others need explanation. For instance, octal (base-8) numbers used to begin with a leading 0o prefix (zero, lowercase "o"). Hexadecimal (base-16) has a leading 0x, and binary (base-2) 0b.

Strings are text objects, and once a string variable is created, its methods can be called directly. Numbers, strings, and tuples are immutable—their values can't be changed, though the reference to them can. This allows Python to optimize memory by reusing literal values.

Strings, lists, and tuples are **ordered collections** (or sequences), and are covered in later chapters.

Dictionaries are key-value collections, similar to associative arrays in awk/PHP, or hashes in Perl/Ruby.

Sets, introduced in Python 2.4, are collections of unique values. There's also an immutable version called frozenset.

Bytes are immutable sequences of 8-bit values; bytearray is the mutable

version. Lists, tuples, dictionaries, and sets are all considered part of Python's **collections**, and are explored in detail in their own chapter.

Switching Types

Sometimes Python switches types automatically

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

13.36728198

6

Num gets automatic promotion

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

QA

34

Like many modern languages, Python automatically converts between numeric types—typically from narrower to wider types (e.g., int to float). To force integer division (truncated result), use the `//` operator. With other types, especially when mixing strings and numbers, conversions must be explicit.

A common beginner mistake is mixing incompatible types, like trying to add a string and a number. Python doesn't assume your intent—`+` could mean addition or string concatenation—so it throws a `TypeError`. You must tell Python what conversion to perform using functions like `str()`, `int()`, or `float()`.

Some developers adopt naming conventions (e.g., Hungarian notation) to hint at types—prefixing strings with `s`, integers with `i`, and lists with `l` (e.g., `sName`, `iCount`, `lNames`). This can help, but it's not common practice in Pythonic code.

When calling `str()`, `int()`, or `float()`, you're actually invoking constructors for those classes—creating new objects of the specified type. To check memory usage, use `sys.getsizeof()`:

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

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

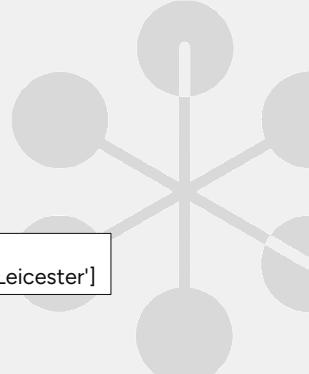
Python Lists introduced

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

QA

35

Lists in Python are objects that hold a sequential collection of other objects, known as elements. Elements are accessed using square brackets [], with positions starting from zero—a convention shared by most programming languages. This zero-based indexing likely stems from early languages like BCPL and C, where memory addressing began at zero. Mathematicians might simply call it natural!

Python lists are **mutable**, meaning they can be modified—unlike tuples or strings. They're also dynamic, allowing elements to be added or removed at any position.

We'll explore lists in more detail in the Collections chapter.

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 ().
- Indexed like lists starting from 0 on the LHS or -1 on the RHS.

```
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
```

QA

36

Tuples are **immutable**—once created, they cannot be modified. For example, trying to append an item results in an error:

```
TupleVar.append('Vikings')  
AttributeError: 'tuple' object has no attribute 'append'
```

Assigning values to a tuple is called packing, while extracting them into variables is unpacking. However, because tuples are immutable, you cannot reassign individual elements after creation.

At first, tuples and lists may seem similar, but tuples offer performance advantages due to their fixed size. When flexibility isn't needed, tuples are often more efficient than lists. While parentheses are commonly used to define tuples, they're optional in many contexts.

For example:

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

This is equivalent to using parentheses. We'll explore tuples further in the Collections chapter.

Python Dictionaries introduced

A Dictionary object is an "ordered" collection of objects

- From Python 3.6, Python dictionaries are ordered in Insertion Order

- 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'
```

QA

London



37

Dictionaries in Python are similar to associative arrays in awk and PHP, or hashes in Perl and Ruby. They store key: value pairs and are defined using curly braces {} or with the dict() constructor:

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

Keys are typically strings, and values can be any valid Python object—such as a list, tuple, or even another dictionary. No special syntax is needed to access values via their keys.

Historically considered unordered, dictionaries became **ordered by insertion** in CPython 3.6, and this is now a guaranteed language feature from Python 3.7 onwards.

You can retrieve all keys using **.keys()**, and all values using **.values()**.

Dictionaries—and their related type, sets—are covered in more depth in the Collections chapter.

Review Questions

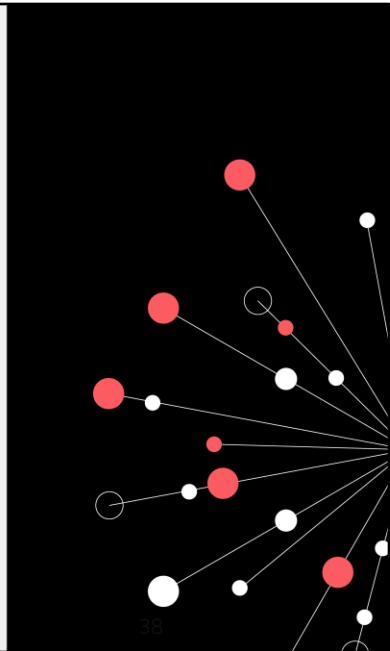
1. What is the correct statement about Python variables?

- a. They must be declared with a type
- b. They store the actual value
- c. They are references to objects in memory
- d. They must start with an uppercase letter

2. Which of the following types is immutable in Python?

- a. List
- b. Dictionary
- c. Set
- d. Tuple

QA



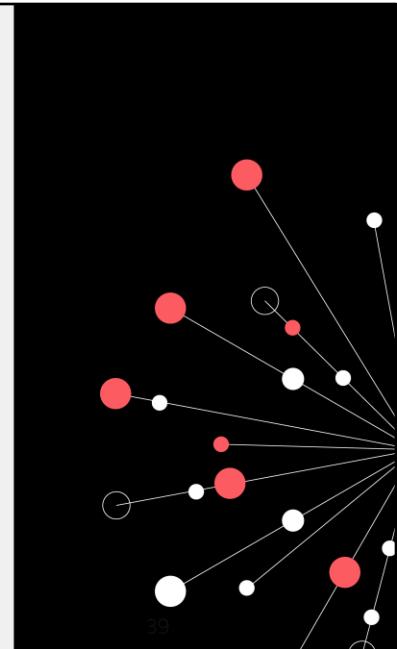
38

Answers on next page..

Review Questions

3. What does the += operator do in Python?
 - a. Adds two variables and deletes the second
 - b. Appends to a list only
 - c. Combines addition and assignment in one step
 - d. Only works with integers
4. What will type("Hello") return?
 - a. 'String'
 - b. <type 'Text'>
 - c. <class 'str'>
 - d. object(str)

QA



Correct answers:

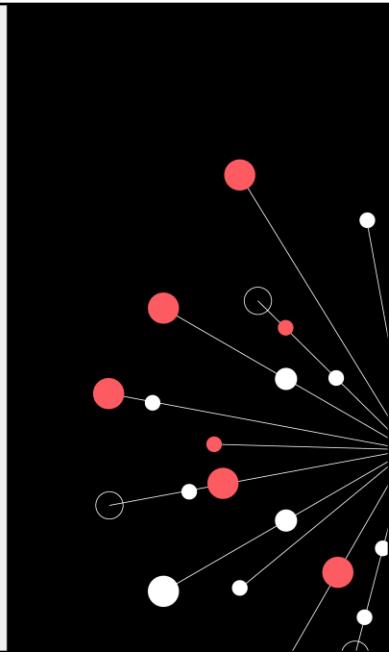
1. What is the correct statement about Python variables?
 - c. They are references to objects in memory
2. Which of the following types is immutable in Python?
 - d. Tuple
3. What does the += operator do in Python?
 - c. Combines addition and assignment in one step
4. What will type("Hello") return?
 - c. <class 'str'>

Summary

Everything is an object in Python

- Variables are simply names referencing these objects.
 - Follow good naming conventions
- Variables are dynamically typed — determined at assignment.
- Use `type()` and `isinstance()` to inspect the class/type of an object.
- Common built-in types include:
 - **Numbers** (int, float, complex)
 - **Text** (str)
 - **Collections**: list, tuple, dict, set, frozenset
 - **Binary**: bytes, bytearray
- Mutable vs Immutable:
 - **Mutable**: list, dict, set, bytearray
 - **Immutable**: int, float, str, tuple, frozenset, bytes
- Augmented assignment operators (`+=`, `-=`, `*=` etc.)
- Conversions between types (e.g. `str()`, `int()`) must be explicit.

QA

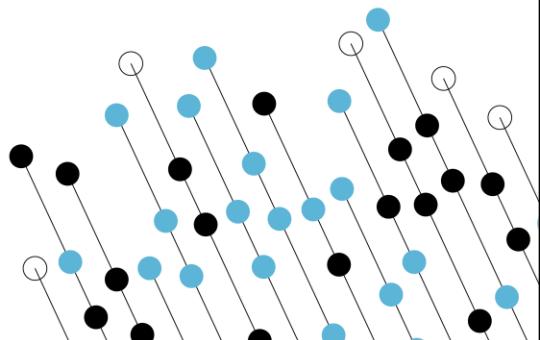


Module 4 – String Handling

4: String Handling

4: Collections

6: Regular Expressions

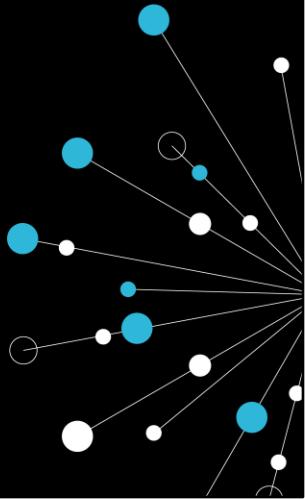


Learning Objectives

By the end of this chapter, you will be able to:

- Explain how Python represents strings as Unicode text.
- Use escape sequences and triple quotes to handle special and multi-line text.
- Apply string concatenation and understand why strings are immutable.
- Use key string methods for searching, testing, and transforming text.
- Format strings using `str.format()` and f-strings.
- Split and join strings effectively for text processing.
- Perform slicing operations on strings and other sequence types.

QA



Python Strings

Strings in Python 3 are Unicode

- Unicode: a universal numbering system for characters
 - Unique code for every character for every language (letters, numbers, symbols)
- Multi-byte characters
- \unnnn for a two-byte Unicode character
- \Unnnnnnnn for a four-byte Unicode character
- \N{name} for a named Unicode character

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()
```

QA

```
>>> euro="\u20ac" ← Two-byte Unicode char
>>> euro
'€'
>>> euro="\N{euro sign}" ← Named Unicode char
>>> print(euro)
€
```

Python 3 introduced a major revision of string handling. All strings are now **Unicode** objects, capable of representing characters from any language. Internally, Python stores them as efficiently as possible: for example, ASCII characters only use one byte each, even though the object is Unicode. This optimisation, introduced in Python 3.3, significantly reduces memory usage.

By default, Python source files use **UTF-8 encoding**, but this can be changed with a pragma at the top of the script: `# -*- coding: latin-1 -*-`

A full list of supported encodings is available in the documentation, and the current default can be checked with `sys.getdefaultencoding()`.

The old Python 2 `u"..."` prefix was dropped in Python 3.0–3.2 but restored in 3.3 for backward compatibility. In modern Python it has no effect.

You may also encounter `bytes` and `bytearray`, which represent raw single-byte data. These are mainly used for low-level C interfaces and for compatibility with Python 2. Some libraries may define their own string types—for example, PyQt's `QString`.

Printing Strings in Python

Printing strings in Python

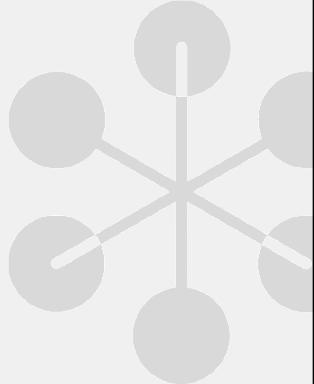
- `print()` is one of the most commonly used functions:
- Display a **comma-separated list of objects**, automatically converting
`print(object1, object2, ...)`

Key named arguments (can appear in any order)

- **sep**: separator between items (default is space)..
- **end**: string appended after output (default is newline \n)
- **File**: file-like object to write to (default is sys.stdout)
- **Flush**: whether to flush the output buffer (default is False since 3.3)
`print("The answer is", 42, "always", sep=':', end='')`
`print("(I think)")`

QA

The answer is: 42: always(I think)



44

The built-in `print()` function is one of the most frequently used functions in Python. It first appeared in Python 2.6, replacing the old `print` statement in Python 3.0 (so both forms existed in 2.6 and 2.7). Unlike the old statement, the function form always requires parentheses, and it also introduced several useful named arguments.

The `flush` argument was added in Python 3.3.

When we say that objects are stringified, we mean they are converted to strings in the same way as the `str()` built-in function. As we'll see later, classes can override this behaviour by defining their own method for string conversion.

Finally, `print()` can also send output to a file-like object, which we will cover in a later chapter.

Escape Sequences in Strings

Escape sequences add special meaning to characters:

- \n → newline
- \t → tab
- ...and many more

Escapes can also remove special meaning:

- \\ → literal backslash
- \' → literal single quote
- \" → literal double quote

Raw strings (prefix r"..."")

- treat \ as a normal character, with no escaping

QA

Escape	Meaning
\newline	Newline is ignored
\\	Backslash \
\'	Single Quote
\"	Double Quote
\a	Bell (BEL)
\b	Backspace (BS)
\f	Formfeed (FF)
\n	Linefeed (LF)
\N{name}	Unicode Char by Name
\r	Carriage Return (CR)
\t	Horizontal Tab (TAB)
\xxxxx	Unicode Char – 16 bit
\xxxxxxxxx x	Unicode Char – 32 bit
\xhh	Char – hex value
\ooo	Char – Octal value

45

Python supports a wide range of escape sequences for representing special characters – see table in slide.

Triple-quoted strings (""""...""") will be discussed later.

Raw strings (r"..."") are especially useful when you don't want escape sequences processed.

- Example: in regular expressions, \1, \2, etc. are used for back-references. Without a raw string, these might be misinterpreted as \x01, \x02, etc.
- Note: the last character inside a raw string cannot be a lone backslash (\).

String Concatenation

Adjacent literals are concatenated.

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

Line continuation

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

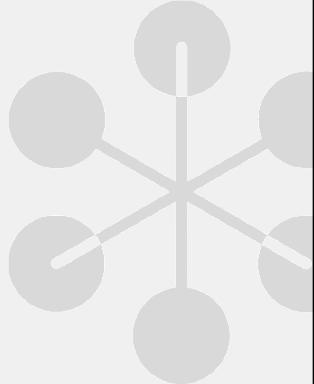
```
>>> name = first + 'bloggs'
```

But remember that strings are **immutable**.

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

Very inefficient Code

Use str.join() instead



QA

46

Much like **awk**, string literals in Python can be implicitly joined simply by placing them next to each other. Whitespace between them is optional. This is useful for splitting long string literals across multiple lines. However, if the split involves a line break in your code, you must escape the newline with a backslash to prevent it from being treated as the end of the statement.

Unlike awk, this technique does not work with variables - an operator must be used. Python, like C++ and Java, **overloads the + operator** for string concatenation.

Although using + for concatenation is convenient, it should not be overused, especially inside loops. Each + operation **creates a new string object**, copying the contents each time, which can be inefficient. A more efficient approach for building larger strings is the join() method, which we'll cover later.

Quotes in Python Strings

- Single ('') and double ("") quotes behave the same
- Choose one style to avoid escaping:

- Use " if the string contains ''
- Use ' if the string contains ""'

```
print('hello\nworld') ←→ print("hello\nworld")
```

- For strings with **embedded quotes** or **newlines**:
 - Use triple quotes (''' or """).

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

Triple Quoted Strings
Improve readability!

QA

47

In Python, **single ('') and double ("") quotes have no difference in meaning** - both can contain special characters. The choice usually depends on the content of the string:

- If the string contains **single quotes** (e.g. SQL statements), wrap it in double quotes.
- If the string contains **double quotes** (e.g. HTML attributes), wrap it in single quotes.
- If the string contains **both quotes**, or spans multiple lines with whitespace characters such as tabs and newlines, use **triple quotes** (''' or """).

Triple-quoted strings automatically handle embedded newlines and quotes. This makes them especially useful when dealing with **user-provided data** (e.g. from forms) that may include unexpected characters or even malicious content such as SQL injection attempts.

With triple quotes, be cautious of **trailing special characters**, especially backslashes and quotes. For example: file = """C:\QA\Python\PYTHB\""" does not work since the final backslash escapes the quote. Use a double backslash

instead - """C:\QA\Python\PYTHB\\"""

!

String functions and methods

Some useful string functions

- Most of these can be used with other types

Function	Meaning
<code>int("42")</code>	Convert string to integer
<code>str(42)</code>	Convert integer to string
<code>len("42")</code>	Return num of chars in string
<code>min("abc")</code>	Smallest Unicode char – a
<code>max("abc")</code>	Largest Unicode char – c
<code>sorted("cba")</code>	Return sorted list – ['a', 'b', 'c']
<code>chr(0x20ac)</code>	Returns Unicode char - €

- Overloaded * operator – string replication

```
>>> name = "spam" * 4      'spam spam spam spam'
```

^Monty Python reference

QA

Some useful string methods

- Over 40 methods available

Method	Meaning
<code>str.upper()</code>	Return upper-case string
<code>str.lower()</code>	Return lower-case string
<code>str.title()</code>	Return title-case string
<code>str.find()</code>	Search for string
<code>str.replace()</code>	Replace string
<code>str.rstrip()</code>	Strip trailing chars
<code>str.ljust()</code>	Left justify string

48

Python provides a wide range of **built-in functions and methods** for working with strings. These tools make it easy to analyse, transform, and format text. Some of the most useful **built-in functions** include `len()`, which returns the number of characters in a string, and `sorted()`, which lists the characters in order. The `min()` and `max()` functions can be used to find the smallest and largest characters based on their Unicode values, while `ord()` and `chr()` convert between characters and their numeric codes. These are general functions that work on many data types, but they are particularly handy for understanding how Python represents and compares text internally.

In addition to these functions, strings also have a rich set of **methods** that act directly on string objects. Common examples include `upper()`, `lower()`, and `title()` for changing the case of text, and `strip()` for removing extra spaces from the start or end. The `replace()` method substitutes one substring for another, while `split()` and `join()` are powerful for breaking text into lists and re-combining it. Searching within strings is easy using `find()` or `count()`, and methods like `startswith()` and `endswith()` help validate user input or file types.

Together, these functions and methods allow Python programmers to clean,

manipulate, and analyse text efficiently. Understanding how and when to use them forms a core part of writing readable and effective Python programs.

Searching and Testing Strings

Searching for substrings using 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.

Test Methods	Code Snippet	Annotations
<code>startswith()</code>	<code>text = "hello world"</code>	Count occurrences of string
<code>endswith()</code>	<code>print(text.count("o"))</code>	Test string starts with "hell"
<code>isalnum()</code>	<code>if text.startswith("hell"):</code>	Test string only contains alpa chars, fails because of space
<code>isalpha()</code>	<code> print("It's hell in there")</code>	
<code>isdigit()</code>	<code>if text.isalpha():</code>	
<code>islower()</code>	<code> print("string is all alpha")</code>	
<code>isspace()</code>	<code>text = '\t\r\n'</code>	2 It's hell in there string is whitespace
	<code>if text.isspace():</code>	Test string only contains whitespace
	<code> print("string is whitespace")</code>	

QA

49

Several built-in methods are available in Python for testing the content or type of a string. Most of these return a Boolean value (True or False), although some, such as `count()`, return a number instead. While many of these checks could be written in other ways, one common alternative is to use **Regular Expressions (REs)**. However, regular expressions are often slower and rarely easier to read than Python's built-in methods.

Here's an example of equivalent code using Regular Expressions - we'll explore the 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")
```

String Formatting using str.format

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:

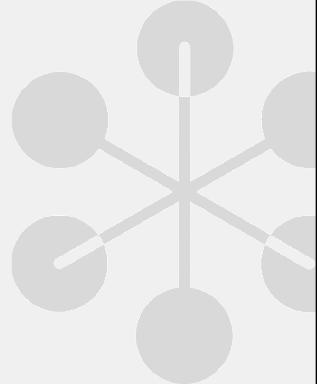
```
{ position : fill align sign # 0 width . precision type }
```

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

- Positions are optional if sequential (Python 3.1)

We can also specify index numbers or key names

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



QA

50

Many languages use printf or sprintf; Python instead provides **string formatting**. Python 3 introduced a new, clearer formatting style (also available from Python 2.6). For the older style, see the slides after the summary.

The **format specification fields** have the following meanings:

Field	Description
fill	Any character except {
align	< left, > right, ^ centre, = pad
sign	sign+ force sign, space for negatives only
#	Prefix integers with 0b, 0o, or 0x
0	Pad numbers with zeros
width	Minimum field width
.precision	Max width (strings) / decimal places (numbers)
type	Integers: b, c, d, n, o, x, X Floats: e, E, f, g, G

All fields are optional - by default, the type is a string.

Named fields can also be used for substitution:

```
var1 = 42
```

```
var2 = "hello"
```

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

Example – Using str.format

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(), start=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



QA

The format specifiers shown in the slides are type specifiers - they define how each argument should be displayed.

The example format string uses the following specifiers:

Specifier	Meaning
{2d}	Right-justified, at least two digits, space-padded
{<10s}	Left-justified string, at least 10 characters wide
{06.2f}	Right-justified, at least six characters, zero-padded. One of the six is the decimal point, and the value is rounded to two decimal places

The second parameter in enumerate(..., 1) sets the **starting index** for enumeration (default is 0).

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]))
```

Gm (gigametre) is a lesser-used metric unit equal to one million kilometres. In

this example, it represents each planet's **mean distance from the Sun**.

String Formatting using methods

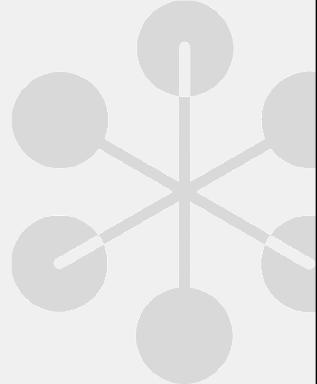
Simpler and more readable

Method	Meaning
<code>str.center()</code>	Return title-case string
<code>str.ljust()</code>	Search for string
<code>str.rjust()</code>	Replace string
<code>str.zfill()</code>	Strip trailing chars

```
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>
```

QA



52

In many cases, explicit format strings aren't needed - the standard string formatting methods are often sufficient. The examples shown should be largely self-explanatory.

Python also provides numerous other string methods. For example, `string.title()` converts text to **title case**, capitalising the first letter of each word.

Literal String Interpolation – f-strings

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")
```

QA

53

A new and more powerful system, defined in **PEP 498** and introduced in **Python 3.6**, provides a cleaner approach - similar to Ruby's syntax.

Single or double quotes can be used, and even triple quotes for multi-line strings. You simply place a Python expression (typically a variable) inside **curly braces {}** within the string.

This is known as an **f-string**, or *formatted string literal*.

Literal String Interpolation – with expressions

Any valid Python expression can be used in f-string braces:

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

2nd of 4 is Harry

And can also be combined with raw strings:

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

Use f-strings with Unicode text – they are not supported in byte objects

QA

54



Function calls, arithmetic operations, and any valid Python expressions can be placed inside the braces of an f-string. However, it's best to Keep It Simple — complex expressions inside strings can be hard to read and even harder to debug. In practice, f-strings are most effective when used with short, clear expressions.

A raw string can be combined with an f-string by using either the `fr` or `rf` prefix. This is useful when working with patterns that include backslashes, such as regular expressions or file paths.

The `fr'` example simply demonstrates the syntax; in real code, you should use `os.path.join()` to construct file paths correctly. Raw and f-strings are most commonly encountered in regular expression work, which we'll explore later in the course.

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])
print(text[-7:-1])
```

bir
an Blu

- Start and end positions may be defaulted.

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

QA



55

Sequential objects like **strings**, **tuples**, and **lists** can be *sliced* to extract portions of data. To access a single element, use **square brackets** with the index inside. Indexing starts at **0** from the left, or **-1** from the right for negative indexing.

To extract a range, specify the **start** and **end** indices separated by a colon:

```
sequence[start:end]
```

The slice includes elements up to but not including the end index.

If the start index is omitted, Python begins at the first element.

If the end index is omitted, it slices to the end.

Example: `string[:-1]` # removes the last character

Python 2.3 introduced an optional **third index**, the **step**, allowing you to skip elements:

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

What do you think this does? (*Try it!*)

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

Splitting and Joining Strings

String to list - split

```
string.split([ separator[, max_splits]])
```

- If **separator** is omitted, split on one or more whitespace characters.
- If **max_splits** is omitted, split the entire string.
- Use **string.splitlines()** to split text into lines (e.g., from files).

Sequence to 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] = "brian"
elems[4] = "superuser - NOT the Messiah"
line = ":".join(elems)
print(line)
```

brian::0:superuser - NOT the Messiah:/root:/bin/sh

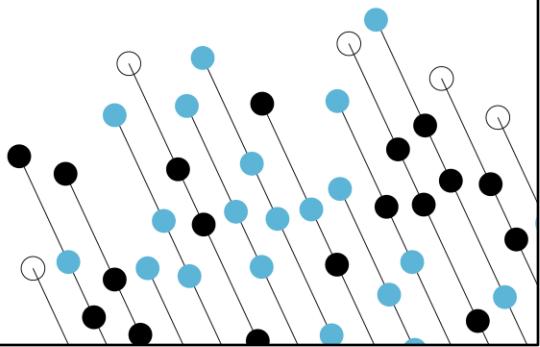
QA

56

Review & Lab

Answer review questions

Your instructor will guide you which lab exercises to complete



Review Questions

1. Which of the following correctly creates a multi-line string?

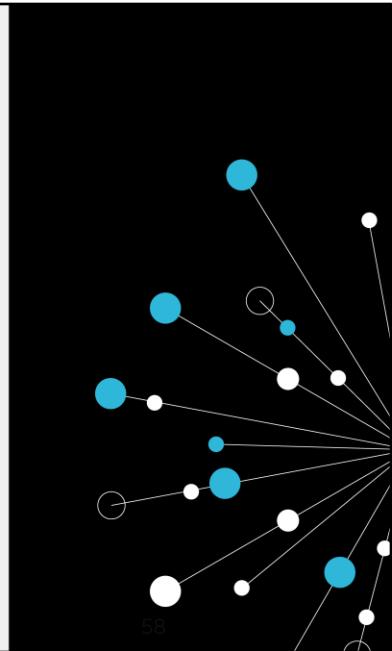
- a. 'This is a string\nwith multiple lines'
- b. "This is a string with multiple lines"
- c. """This is a string with multiple lines"""
- d. str("This is a string, with multiple lines")

2. What is the output of the following code?

```
print("Python"[-1])
```

- a. TypeError: Literal 'str' object is not subscriptable
- b. n
- c. o
- d. P

QA



58

Answers on next page..

Review Questions

3. What does the following code print?

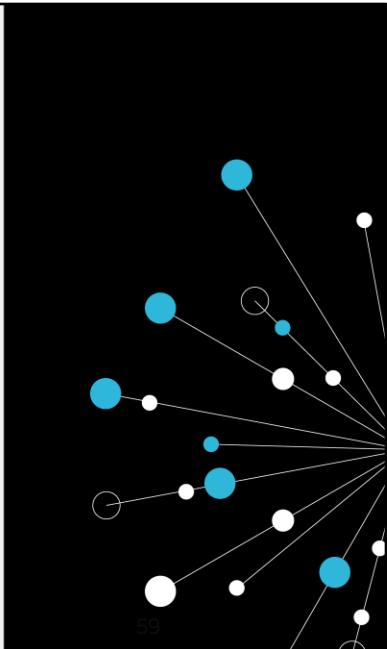
```
print("hello world".capitalize())
```

- a. Hello world
- b. HELLO WORLD
- c. hello world
- d. Hello World

4. Which statement about f-strings is true?

- a. They can include variables and expressions inside {}
- b. They can contain literal text
- c. They automatically escape all special characters
- d. They can be used with byte objects

QA



59

Correct answers:

1. Which of the following correctly creates a multi-line string?

- c. """This is a string with multiple lines"""

2. What is the output of the following code?

- b. n

3. What does the following code print?

- a. Hello world

4. Which statement about f-strings is true?

- a. They can include variables and expressions inside {}

Summary

Python 3 strings are Unicode

- No difference between ' and ", use whichever improves readability
- Triple quotes allow multi-line text
- Escape characters like \r, \n, and \t can be included inside strings

Variables are not interpolated inside quotes

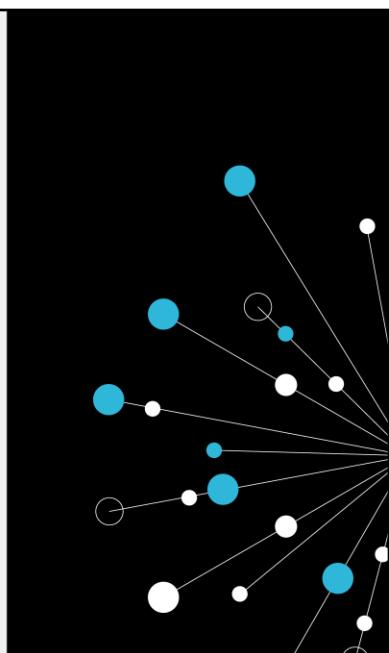
- Use f-strings or format() for insertion.

Common string methods:

- format() – for string formatting
- split() – to divide a string into a list
- join() – to combine a sequence into a string

Strings can be sliced using [start:end+1], just like other sequences.

QA

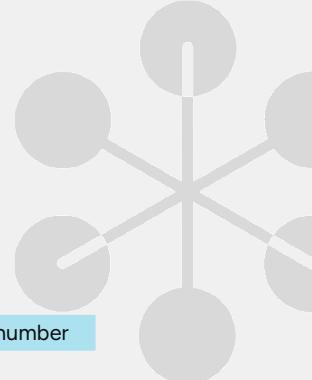


String Formatting - Examples

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

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

```
file   = sys.argv[0]
perms = "0{:o}".format(os.stat(file).st_mode & 0o7777)
    Octal number
            0640
```



QA

The `format()` method is particularly useful for **displaying floating-point numbers** in a wide range of formats.

In the first example, the number is printed with a **total width of 11 characters** (including the decimal point) and **8 digits after the decimal point**, followed by **engineering notation**. Only one value is supplied because both format specifications refer to **position zero**.

In the later examples, **positional indexes are omitted**, so the arguments are **automatically numbered** (a feature introduced in Python 3.1).

Text fields can be **left- or right-aligned**, as shown in the second example, where two columns are **left-aligned and padded with spaces**.

Conversion to **hexadecimal** (or **octal**, using `o`) is also straightforward. One example demonstrates **returning the formatted string** rather than printing it directly.

Finally, values commonly stored in **octal**, such as **UNIX file permissions**, can be

displayed and manipulated more easily using `format()`.

String Formatting in Python 2

Format strings using % operator – like printf in other languages

```
"format_string" % (argument_list)
```

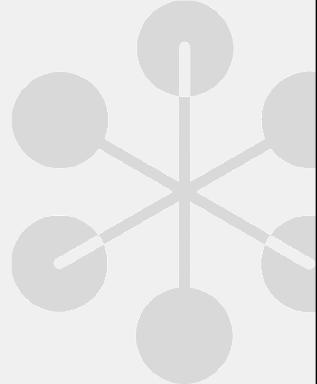
format_string

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

Specifier	Format as	Specifier	Format as
%s	String	%u	Unsigned int
%c	Character	%o	Octal
%d	Decimal	%x	Lowercase hex
%i	Integer	%X	Uppercase hex
%f	Float	%e	Engineering

argument_list - text or variables to be plugged-in.

QA



This older style of string formatting is still supported in Python 3, but it is **deprecated** and should be **avoided in new code** - and be removed in a future version of the language.

Many languages use the **printf / scanf** family of functions (originally from C). Python 2 adopted a similar approach by **overloading the % operator**. When mixing text, variables, and escape sequences, this provided more flexibility than print.

The general syntax of a format specifier is: %[flag][width][.precision]letter
Where:

flag	specifies padding and sign options
width	defines the total field width
.precision	sets the number of decimal places for floats
letter	indicates the data type (e.g., d, f, s, e)

To print a literal percent sign, use **%%**. Only the most common specifiers are typically used - refer to the online documentation for the full list. Floating-point values are **rounded automatically**, though the rounding algorithm may differ

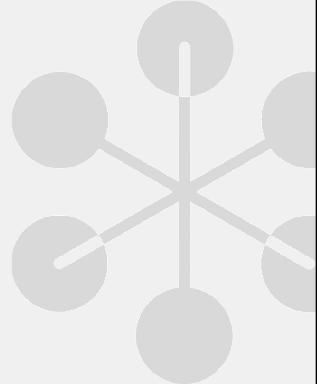
from what you expect. For critical cases (such as financial calculations), it's best to implement a **custom rounding method** rather than rely on %.

String Formatting in Python 2 - Examples

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

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

```
file   = sys.argv[0]
perms = '%lo' % (os.stat(file).st_mode & 0o7777)
    0640
```



QA

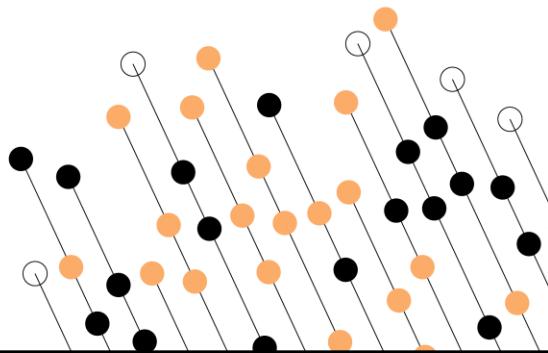
These are the same examples used in a previous slide, but using Python 2 syntax and the % operator to format strings. Those with experience of the printf from C and Java will find the syntax quite familiar.

Module 5 - Collections

5: Collections

6: Regular Expressions

7: Data Storage and File Handling



Learning objectives

By the end of this chapter, learners will be able to:

- Know the differences between the different collection types
- Create and query tuples
- Create, add and remove items from lists
- Sort and slice lists
- Create sets and carry out basic set operations (intersection, union, etc.)
- Create dictionaries and use keys to retrieve associated values

QA



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

Python types – a 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')`
- And bytearray (mutable) and bytes (immutable) used for binary data

Not all collections are sequences

- A set is an unordered collection of unique objects
- Dictionaries have key-value pairs, with their order preserved (Python 3.7)
`{'Arthur': 'King of the Britons', 'Lancelot': 'Brave Knight'}`



QA

66

Strings, Lists, and Tuples are **ordered collections** of objects, often referred to as **sequences**.

The bytes and bytearray types (introduced in Python 2.6) are used for raw binary data. They behave similarly to strings - supporting many of the same methods and allowing element access via indexing `[]` - but store 8-bit unsigned integers in the range 0–255.

Dictionaries are key-value collections, comparable to associative arrays in AWK and PHP, or hashes in Perl and Ruby.

Sets, introduced in Python 2.4, store unique, unordered elements and support standard mathematical set operations.

The collections module in the Python Standard Library provides enhanced container types, such as deque, Counter, and OrderedDict (added in Python 3.1).

From Python 3.6 (CPython), dictionaries maintain insertion order, and as of Python 3.7, this behaviour became an official language feature.

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)

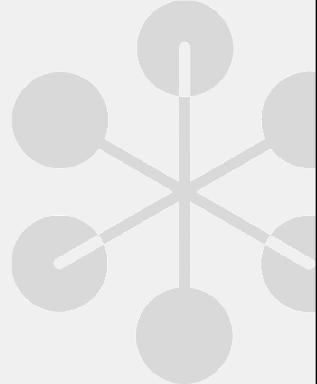
Dictionaries implement `min`, `max`, and `sum` on keys.

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`



QA

67

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 types.

The `sum()` function supports an optional second argument, `start`, which specifies an initial value for the sum (default is 0).

The `len()` function returns the **number of characters** in a string, and the **number of bytes** in 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

Useful tuple operations

- Swap references:

```
a, b = b, a    0 1 2
```

- Set values from a numeric range:

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

- Repeat values:

```
mytuple = "a", "b", "c"  
another = mytuple * 4  
('a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c')
```

- Be careful of single values and the trailing comma:

```
thing = ("Hello")  
print(type(thing))  
<class 'str'>  
thing = ('Hello')  
print(type(thing))  
<class 'tuple'>
```

Careful of trailing commas

QA

68

Tuple elements can be **assigned to named variables**, allowing easy unpacking and swapping. In the example shown, two variables (a and b) are swapped - but the same technique can be used with any number of variables. This is simply a **tuple assignment**, not a modification of the tuple itself. The number of variables on the left must match the number of values on the right. Parentheses are optional, so the statement:

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

can also be written as:

`a, b = b, a`

The next example assigns three variables from a `range()`:

`gouda, edam, caithness = range(3)`

Here, `gouda = 0`, `edam = 1`, and `caithness = 2`.

Tuples (as well as strings and lists) can be **repeated** using the `*` operator. For example,

`another = mytuple * 4`

creates a tuple with the contents of `mytuple` repeated four times.

Be aware that using parentheses around a single item does *not* create a tuple - a

trailing comma is required:

```
single = (42,) # Correct  
not_tuple = (42) # Just an integer
```

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']
```

QA



69

A **list** is an object that contains a **sequential collection of elements**, each of which can be accessed by its index position (starting from zero) inside square brackets [] — a concept familiar from many other languages.

Lists are **mutable**, meaning they can be modified after creation. They are also dynamic, allowing elements to be added, removed, or reordered at any position within the list.

Lists can be **concatenated** (joined together) using the + operator.

Although Python lists are often compared to arrays in other languages, this analogy only goes so far. Python lists are **more flexible**, as they can store elements of different types.

However, this flexibility introduces a performance overhead compared to low-level C arrays, which are much faster. For performance-critical applications requiring array-like efficiency, Python provides the **array** module in the standard library - offering objects similar to C-style arrays.

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])
print(mytuple[-4])
mylist = list(mytuple)
print(mylist[1:])
print(mylist[:2])
```

(`'spam', 'tea'`)
`bacon`
['`bacon', 'spam', 'tea', 'beans'`]
['`eggs', 'bacon'`]

start:end+1



- List elements may be removed using del

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

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

QA

70

Sequential composite objects, such as **strings**, **tuples**, and **lists**, can be **sliced** to extract a subset of their elements.

When only a single item is needed, use **square brackets** with an **index value** inside. Indexing starts from **0** on the left and **-1** on the right (counting backward from the end).

To slice a range of elements, specify the **start index**, a **colon**, and the **end index plus one** - meaning the slice includes elements up to but not including the second index.

If the start index is omitted, Python assumes 0 (the first element).

For example: `string[:-1]` returns all characters except the last one, effectively trimming it. If the end index is omitted, the slice extends to the end of the sequence. Strings, tuples, and lists all support this same slicing behaviour.

The `del` statement can remove a slice, a comma-separated list of elements, or even the entire list. Note that `del` is a statement, not a function - so no parentheses are required around the object being deleted.

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

QA

71

PEP 3132 introduced a useful simplification to tuple assignment - known as **extended unpacking**.

Previously, when assigning to multiple variables from a tuple, the number of variables on the left had to exactly match the number of values on the right. In Python 3, you can prefix a variable with an **asterisk (*)** to collect multiple remaining values into a list during unpacking.

This variable can appear anywhere on the left-hand side, not just at the end, and it allows you to handle tuples of unknown or variable length without manual slicing.

For example:

```
a, *b, c = (1, 2, 3, 4, 5) # a = 1, b = [2, 3, 4], c = 5
```

A similar use of * for unpacking has existed since Python 2, particularly in function argument lists - a concept we'll explore later.

Adding items to a list

On the left:

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

Insert before Index 1

On the right:

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

Both statements extend list by two strings

- 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']
```

Both statements insert string before index 2

QA

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

72

Lists can be extended in **any direction** and from **any position**.

In the first example, 'Cheshire' and 'Ilchester' are added to the **front** of the cheese list.

The second example demonstrates two ways to add items to the **end** of a list - using the `+=` operator or the `extend()` method.

Although `extend()` is theoretically more efficient, the performance difference is usually negligible in practice.

The third example uses the `append()` method, which adds a single item to the end of a list — often all that's needed.

Finally, there are two ways to insert elements at a specific position:

- Using `insert(index, item)`, which adds one item at the given position.
- Using slicing assignment, which allows multiple items to be inserted at once.

Removing items by position

Use `pop(index)`

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

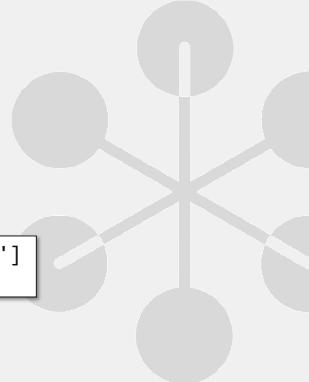
```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg']
saved = cheese.pop(1)
print("Saved1:", saved, ", Result:", cheese)

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.~~

- But does not return the deleted item
- And can delete more than one item by using a slice



QA

73

The **pop()** method removes an item from a list - by default, the last (rightmost) element.

You can specify an index to remove a particular item, for example:

cheese.pop(0)

removes the first (leftmost) element.

An advantage of `pop()` over `del` is that `pop()` returns the removed item, whereas `del` simply deletes it. However, `del` can remove multiple elements at once, while `pop()` removes only one.

Removing items from the end of a list (the default `pop()` behaviour) is generally more efficient, since deleting elements from the middle or beginning requires rebuilding the list's internal structure.

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
```

QA

74

Sometimes, we may not know the **position** of an item but want a simple “**search and remove**” operation - and that’s exactly what the `remove()` method does.

Note that list elements don’t have to be unique - `remove()` will delete only the first (leftmost) occurrence of the specified value.

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** - modifies the list

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']
```

QA

75

Python uses the **Adaptive, Stable Mergesort** algorithm (also known as Timsort, created by *Tim Peters*). The **sorted()** function, introduced in Python 2.4, **returns a new sorted list**, leaving the original sequence unchanged.

In contrast, the **sort()** method **sorts the list in place** and **returns None**.

The **key** parameter specifies a single-argument function that determines the value used for comparison. This is often written as a lambda function - a small, inline anonymous function. The key function is called once for each element, and its return value defines the sorting order.

In one example, sorting strings representing numbers results in textual comparison, whereas sorting numeric values compares their actual numeric value.

The **sorted()** built-in can sort any iterable — including **tuples, strings, and lists** - but always returns a list.

The **sort()** method applies only to mutable sequence types, such as **lists** and **bytearrays**.

Miscellaneous list methods

Method	Meaning
<code>list.count('value')</code>	Return the number of occurrences of 'value'
<code>list.index('value')</code>	Return index position of leftmost 'value'
<code>list.reverse()</code>	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']
```

QA

76

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), but throws a `ValueError` exception if the item is not found.

List Methods

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

QA

77

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

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})
```

QA

78

A **set** can be thought of as a **list without order**, or as **one half of a dictionary** (the **keys**). In fact, the **dict.keys()** method behaves much like a set in Python.

Sets are ideal for **lookup operations**, where you only need to test membership using the `in` operator, and for performing set operations such as **union**, **intersection**, and **difference**.

Keep in mind that **sets are unordered**, so any original sequence order is lost.

Use the `add()` method to insert a single element, or `update()` to add multiple elements at once.

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 methods:

Method	Meaning
<code>len()</code>	Function returns number of elements in set
<code>set.discard()</code>	Remove element if present
<code>set.pop()</code>	Reverse and return element from set
<code>set.clear()</code>	Remove all elements
<code>set.update()</code>	Add elements from another set

QA

79

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>	<code>&=</code>	Update the set, with common elements
<code>difference_update</code>	<code>--</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.

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))
```

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

```
['Cornish Yarg', 'Cheshire', 'Cheddar', 'Stilton', 'Oke']
```

- 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']
```

QA

80

Sets can be used for many purposes, including **membership testing** with the **in** operator. In this example, we use sets to take advantage of their unique properties.

All **items in a set are unique**, so converting a list to a set automatically removes duplicates - though the original order is lost.

A common use case is deduplicating data before sorting.

Before sets were introduced, this was often done using a dictionary with “throw-away” values. Once a list has been converted to a set, you can perform **set operations** such as **union**, **intersection**, and **difference** directly on the data.

When performing set subtraction, the elements in the right-hand set don’t need to exist in the left-hand set. For example, attempting to remove ‘Brie’ from a set that doesn’t contain it is not an error — it’s simply ignored.

Set operators and methods

Set Operators (Remember Venn Diagrams)

Operator	Method	Returns Set of..
&	s6.intersection(s7)	Items common to both sets
	s6.union(s7)	All items from both sets (no duplicates)
-	s6.difference(s7)	Items only in the left but not the right
^	s6.symmetric_difference(s7)	Items in either but not both

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

print(s6 & s7)	{42, 123}
print(s6 s7)	{66, 27, 42, 23, 56, 123}
print(s6 - s7)	{66, 23}
print(s6 ^ s7)	{66, 23, 56, 27}

QA

81

Python supports several **set operations** that behave just like the **Venn diagrams** you may remember from school.

- **Intersection (&)** – items common to both sets
- **Union (|)** – all items from both sets (no duplicates)
- **Difference (-)** – items in the left set but not in the right
- **Symmetric difference (^)** – items in either set, but not both

Each of these has an equivalent **set method**, for example:

`set1.intersection(set2)` or `set1.union(set2)`.

The method versions can accept any iterable as a parameter - there's no need to convert the data to a set first.

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'
```

QA

82

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)
```

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']
UK : ['London', 'Wigan', 'Macclesfield', 'Bolton']
US : ['Miami', 'Springfield', 'New York', 'Boston']
```

Wigan

Springfield

QA

83

This example shows a simple **dictionary** with two keys, 'UK' and 'US', each containing a **list** as its value.

Dictionaries can be extended dynamically by simply assigning a value to a new key - for example, adding 'FR' creates a new entry automatically.

Originally, dictionary order was not guaranteed, so key order could appear random. From **Python 3.7**, however, **insertion order** is preserved as part of the language specification.

To access values in a nested (multi-dimensional) structure, add the key or index inside square brackets - just as in other languages. Keys can be **string literals** (remember to use quotes) or **variables**.

Finally, the example demonstrates iterating through a dictionary, a technique that will be explored in more detail shortly.

Removing items from a dictionary

Use `del` or `pop()` to remove a single key/value pair:

```
del dict[key]    dict.pop(key[, default])
```

- The `pop` method deletes but also returns the key/value pair or default
- Both raise a `KeyError` exception 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'
```

Also:

- `dict.pop('key', False)` returns `False` if the key is not found in iteration.
- `dict.clear()` removes all key/value pairs from the dictionary.



QA

84

Deleting a **key** from a dictionary also removes its **associated value**.

The **del** statement deletes the key–value pair but **does not return** the value. In contrast, the **pop()** method **removes and returns** the value for a given key.

Both `del` and `pop()` will raise a `KeyError` if the key does not exist.

However, **pop()** can take an optional default value, which is returned instead of raising an exception. The **popitem()** method removes and returns an arbitrary key–value pair as a tuple.

Although dictionaries now preserve **insertion order**, `popitem()` is most useful when iterating and clearing a dictionary. It also raises a `KeyError` if the dictionary is empty.

Dictionary Methods

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

QA

85

The return values from `keys()`, `values()`, and `items()` are **view objects** — dynamic windows into a dictionary’s data. They do not create separate copies but instead reflect the **current state** of the dictionary, automatically updating if the dictionary changes.

The `copy()` method creates a **shallow copy**, meaning it duplicates only the top-level structure of the dictionary. If the dictionary contains nested objects (such as other dictionaries or lists), those inner objects are **shared**, not duplicated. A **deep copy** would instead create new, independent copies of all nested objects.

Dictionaries were originally **unordered**, but since **Python 3.7**, the insertion order of keys is **guaranteed** as part of the language specification.

If you need explicit control over ordering — for example, to reorder or sort items — use `OrderedDict` from the `collections` module.

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

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

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

QA

86

View objects, returned by the `items()`, `keys()`, and `values()` methods, can be used in iterator for loops and can also be converted into other collections such as **lists**.

Set operations (`&`, `|`, `^`, and `-`) are supported on `dict_keys` and `dict_items` view objects when used with sets, but **not** on `dict_values`.

Dictionary view objects were introduced in **Python 3** and also made available in **Python 2.7** for compatibility.

Review Questions

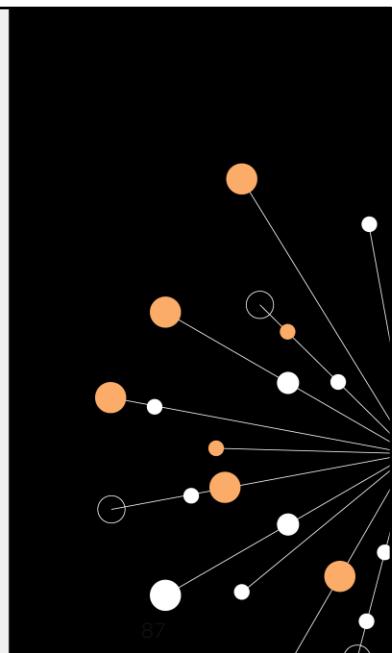
1. Which one of the following collections is immutable?

- a. list
- b. dict
- c. set
- d. Tuple

2. What is the main characteristic of a Python set?

- a. It maintains insertion order
- b. It allows duplicate elements
- c. It stores unique, unordered elements
- d. It maps keys to values

QA



Answer on next page.

Review Questions

3. What does the items() method of a dictionary return?

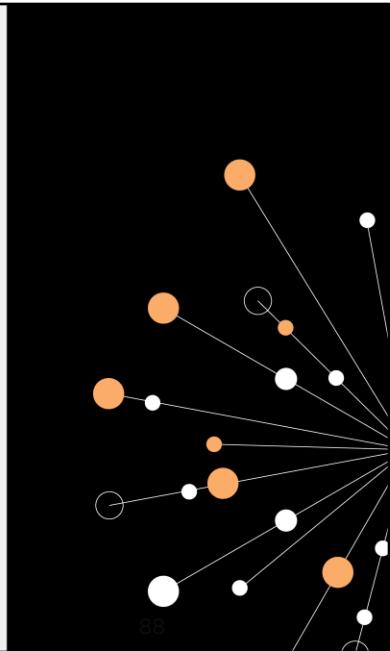
- a. A list of keys
- b. A list of values
- c. A view of key-value pairs
- d. A shallow copy of the dictionary

4. Given the code below, what will be the output?

```
a = {"Arthur": "King", "Lancelot": "Brave"}  
b = {"Arthur": "King", "Galahad": "Pure"}  
print(a.keys() & b.keys())
```

- a. {'Arthur', 'King'}
- b. {'King'}
- c. {'Arthur'}
- d. {'Lancelot', 'Galahad'}

QA



88

Correct answers:

1. Which one of the following collections is immutable?

- d. Tuple

2. What is the main characteristic of a Python set?

- c. It stores unique, unordered elements

3. What does the items() method of a dictionary return?

- c. A view of key-value pairs

4. Given the code below, what will be the output?

- c. {'Arthur'}

Explanation: & gives the intersection of the dictionary keys

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].

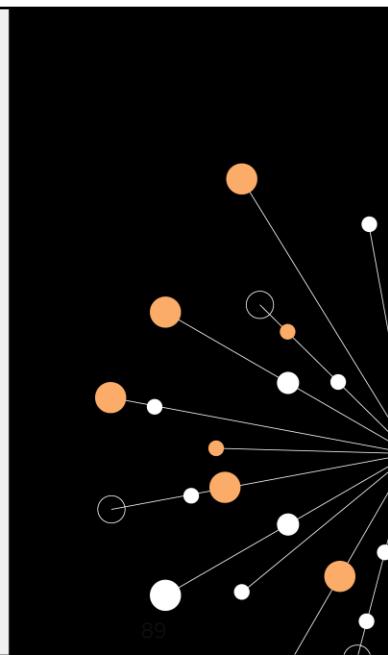
Dictionaries are like associative arrays:

- Mutable & accessed by a unique key.
- Ordered (in insertion order) from Python 3.6.

Sets are like a special version of dictionaries:

- Mutable, unordered & can be combined with other sets.
- They do not have keys but objects are unique.

QA

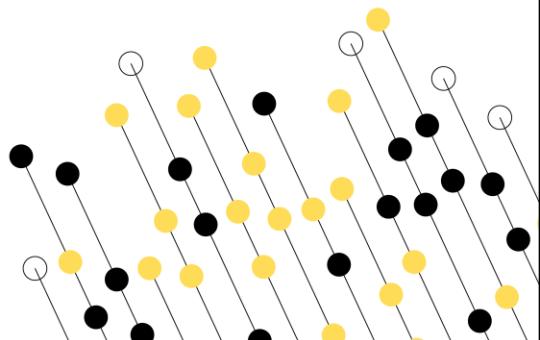


Module 6 – Regular Expressions

6: Regular Expressions

7: Data Storage and File Handling

8: Functions



Learning objectives

By the end of this chapter, you will be able to:

- Understand and use Regular Expressions.
- Use a range of core character classes including:
 - Line Anchors
 - Single character classes
 - Quantifiers
 - Capturing groups and back references
 - Alternation and non-capturing groups
- Use functions from the `re` module to:
 - Match and validate strings.
 - Match and substitute strings.
 - Precompile patterns for improved performance.

QA



This chapter introduces

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.

QA

How to Regex



Editor, Keyboard, Cat & Music

Regular expressions aren't a programming language—they're simply a set of characters and symbols used to find patterns in text. It sounds simple, but they can be incredibly powerful once you understand how they work. Their roots actually go back to neuroscience. In 1943, McCulloch and Pitts described how the human nervous system worked, and in 1951, Stephen Kleene introduced regular sets using algebraic notation to represent these networks—coining the term regular expressions.

In 1968, Ken Thompson, creator of UNIX, updated the QED text editor to support string pattern matching, which laid the foundation for tools like ed, ex, and eventually vi. Thompson also created grep, short for g/re/p, meaning "globally search for a regular expression and print."

Over time, tools like egrep, awk, and later programming languages like Perl embraced and extended regex capabilities. As variations emerged, a standard was needed, leading to the POSIX specification in 1986. Today, most programming languages include regular expression support, either built-in or through libraries.

Although regular expressions can seem cryptic at first (unless you're a well-trained cat), they are an invaluable skill for any developer working with text or data.

Python regular expressions

Supports Extended Regular expressions - with extensions

- Requires the `re` module (in the standard library).
- Notable features:
 - Can pre-compile an expression for performance.
 - Support Multi-line pattern matches.
 - 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.

QA

93

```
extract_num-
ber_and_(destination, source) int
(destination, *unsigned char **source); { extract_num-
ber(destination, *source); *source += 1; } #ifndef EXTRACT_NUM-
BER_AND_INCR
#define EXTRACT_NUMBER_AND_INCR(extract_number_and_incr, &src) #define EXTRACT_NUM-
BER_AND_INCR(dest, src) extract_number_and_incr(&dest, &src) #endif /*

not EXTRACT_MACROS */ #endif /* DEBUG */ If DEBUG is defined, Regex prints
many voluminous messages about what it is doing (if the variable 'debug' is nonzero). If
linked with the main program in 'iregex.c', you can enter patterns and strings interactively.
And if linked with the main program in 'main.c' and the other test files, you can run the all-
ready-written tests. #ifndef DEBUG /* We use standard I/O for debugging. */ #include <stdio.h>
/* It is useful to test things that "must" be true when debugging. */ #include <cassert> static int
debug = 0; #define DEBUG DEBUG #define PRINT(x) if (debug) print(x)
#define DEBUG_PRINT(x) DEBUG(PRINT(x)) if (debug) print(x)
#define DEBUG_PRINT3(x1, x2, x3) if (debug) print(x1, x2, x3)
#define DEBUG_PRINT4(x1, x2, x3, x4) if (debug) print(x1, x2, x3, x4)
#define DEBUG_PRINT_COMPILED_PATTERN(p, s) if (debug) print_partial_compiled_pattern(s, p)
#define DEBUG_PRINT_DOUBLE_STRING(w, s1, s2, s2z) if (debug) print_double_string(w, s1, s2, s2z)

extern void printchar(); /* Print the fastmap in human-readable form. */
void print_fastmap(fastmap*
char *fastmap; { unsigned was_a_range = 0; unsigned i = 0; while (i < (1 << BYTEWIDTH)) { if (fastmap[i] == 1) {
(was_a_range = 0; printchar(i - 1); while (i < (1 << BYTEWIDTH) && fastmap[i]) { if (was_a_range == 1; i++;) if
print_partial_compiled_pattern(start, end) unsigned char start; unsigned char end; { int mcnt, mcnt2; un-
signed char *p = start; unsigned char *pend = end; if (start == NULL) { printf("(null)\n"); return; } /* Loop over
pattern commands. */
while (p < pend) { switch ((re_opcode_t)*p++) { case no_op: printf("(no_op)\n");
break; case exactn: mcnt = *p++; printf("%c", (char)*p); break; case anychar: printf("(anychar)");
break; case charset: case charset_n: { register int c; printf("(charset%c", (re_opcode_t)*p - 1);
== charset_n, not _0); assert((c = *p < pending) && (c <= 0 < c * *p; c++)) { unsigned bit;
aligned_charset_type p1 = p1 + 1; p1 |= 0x80; for (bit = 0; bit < BYTEWIDTH; bit++) if
((map, byte & (1 << bit)) != 0) { printf("%c", (char)*p); break; } } case begin-
line_if: { /*beginline_if*/ break; } case endline_if: { /*endline_if*/ break; } case on_failure_-
jump: extract_number_and_incr(&mcnt, &p); printf("(on_failure_jump(%d)\n", mcnt); break;
case on_failure_keep: string_jump(&mcnt); extract_number_and_incr(&mcnt, &p); printf
("(on_failure_keep_string_jump(%d)\n", mcnt); break; case dummy_failure: jump(
extract_number_and_incr(&mcnt, &p); printf("(dummy_failure_jump(%d)\n", mcnt); break;
case push_dummy: printf("(push_dummy_failure)\n"; break; case may-
be_pop_jump: { extract_number_and_incr(&mcnt, &p); printf
("maybe_pop_jump(%d)\n", mcnt); break; } case pop_failure_-
jump: extract_number_and_incr(&mcnt, &p); printf("(pop_failure_-
jump(%d)\n", mcnt); break; case jump_past_alt:
extract_number_and_incr(&mcnt, &p); printf("(~C, Yuck\n");

93
```

Python supports **Extended Regular Expressions (E.R.E.)** through the `re` module in the Standard Library, along with several powerful extensions. While Python uses E.R.E. syntax, those familiar with tools like grep, vi, sed, and awk should find it relatively easy to pick up. One key difference is that **Basic Regular Expressions (B.R.E.)** require backslashes to escape braces (quantifiers) and parentheses (grouping), whereas in E.R.E. and Python, this is not necessary.

How Python Regular Expressions Compare to grep, sed, and awk:

- Multiline matching:** Python allows patterns to span multiple lines, enabling complex searches without manually managing line state.
- Dynamic substitution:** Python's substitution supports using variables, including list or dictionary lookups based on captured groups in the match.
- Function-based replacements:** You can replace matched text with the result of a function call, giving you highly flexible transformations.
- Self-referencing patterns:** Python allows matching patterns where parts of the match refer back to earlier groups—such as matching a line that starts and ends with the same word or phrase.

Keep in mind that compiling and searching with Regex expressions incurs a

performance cost. For simple tasks, it may be faster and more readable to use built-in string operators like `==`, `!=`, `in`, or methods like `.startswith()` and `.endswith()`.

Core Character Classes I

Line Anchors

- Start `"^the"`
- End `"ing$"`

Escape Char Class

- Usual `r"\\".`
- Alternative `"[.]"`

Single Char Class

- Any 1 char `"^.ing$"`
- Any 4 chars `"^....$"`
- Char set `"^ [rdz] ing$"`
- Char ranges `"^ [A-Z] ing$"`
- Char ranges `"^ [a-zA-Z] ing$"`
- NOT Char `"^ [^dz] ing$"`
- Multiple sets `"[aeiou][aeiou][aeiou][aeiou]"`

Quantifiers

- 0 or more ":[0-9]*:"
- 0 or once ":[0-9]?:"
- 1 or more ":[0-9]+:"
- Exact ":[0-9]{10}:"
- Min, max ":[0-9]{10,20}:"
- At least ":[0-9]{10,}:"
- At most ":[0-9]{0,10}:"

QA

94

The examples above highlight some of the most common **regular expression meta-characters**—a great starting point for beginners.

Line Anchors

Use `^` to match the **start** of a string and `$` for the **end**. Similar functionality exists in Python string methods.

Character Classes

`.` matches **any single character** (usually Unicode).

`[]` defines a **character set** (e.g., `[aeiou]`), and `[^]` negates it (e.g., `[^0-9]`).

Use ranges like `[a-z]` (must be in increasing order).

Escape special characters with `\` or use them inside `[]` (except `^` and `-`, which have special roles).

Quantifiers

`? = 0 or 1 occurrence`

`* = 0 or more`

`+ = 1 or more`

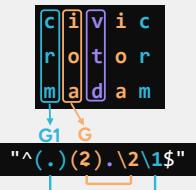
`{n} = exactly n times, {n,m} = between n and m, {n,} = n or more`

Regex is **greedy** by default—it matches the longest possible string.

Core Character Classes II

Capturing groups/back references

- Match 5 char palindromes



- Match strings start/end with same capital



QA

Alternation (OR)

"eric|graham|michael|terry|john"



Alternation plus non-capturing groups

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

- Faster, no back references, just for alternation

- Will match the following strings:

"a **bottle** of **lager**"

"a **glass** of **wine**"

"a **bottle** of **beer**"

95

This slide introduces some additional powerful features of **extended regular expressions**.

The **alternation operator (|)** matches one of several options:

- For example: eric|graham|michael|terry|john.

Parentheses (()) group parts of a pattern. This allows:

- Repetition using quantifiers (e.g., (ab)+)
- Restricting alternation to part of a pattern (e.g., a glass of (wine|beer))

Backreferences allow you to match the **same text** as an earlier capture group:

- Example: ^([A-Z]).*\1\$

^ = start of string

([A-Z]) = capture a capital letter

.* = any characters

\1 = match the same capital as before

\$ = end of string

- This pattern matches strings that start and end with the same capital letter—useful in detecting simple palindromes.

Non-capturing groups use (?:) to group without saving the match:

- Example: 'a (?bottle|glass|keg) of (?lager|wine|beer)'

- This is more efficient when you don't need to reference the groups later

Often used for validating data

What would these match?

1. `"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}"`
2. `"07[0-9]{9}"`
3. `"[0-3][0-9]/[0-1][0-9]/[0-9]{4}"`
4. `"£[0-9]+(\.[0-9]{2})?"`
5. `"https://[A-Za-z0-9.-]+\.[A-Za-z]{2,}"`



QA

96

Regular expressions are often used for validating data and input.

For example, the examples on the slides are used to validate:

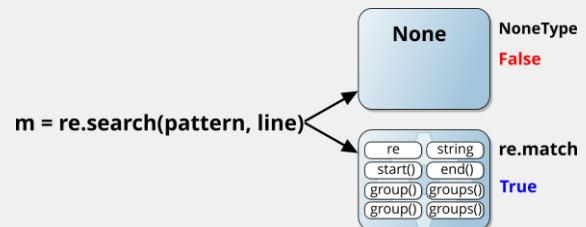
1. Match an email like string, for example, info@qa.com
2. UK Mobile Number, for example, 07123456789
3. A Date in UK format, for example, 25/12/2030
4. A Price with a currency, and pounds and pence, for example, £18.50
5. A simple URL, for example, http://qa.com and https://qa.com

re Match Object

Module **re** provides several functions

- For searching, matching and validation
 - `search()`
 - `match()`
 - `fullmatch()`
 - Precompiled using `compile()`
- Returns:
 - None** (`False`) or **re.match** (`True`) object
 - Exception `re.error` on failure
 - Match object provides several methods:

```
testy = "The quick brown fox jumps over the lazy dog"
m = re.search(r"(quick|slow).*\b(fox|camel)", testy)
if m:
    print(f"Matched {m.groups()} between {m.start()} and {m.end()}")
```



Matched ('quick', 'fox') between 4 and 19

97

QA

Importing the **re** module makes the regular expression functions available, with **search()** and **match()** being the most common, and **fullmatch()** (added in Python 3.4) used when the entire string must match. Each of these functions returns either a `MatchObject` or `None`. A `MatchObject` evaluates as `True` in a Boolean context, while `None` evaluates as `False`. If the regex itself is invalid, Python raises a `re.error` exception (improved in Python 3.5).

Capturing groups are created with parentheses `()`. The text inside a group can be referenced later in the pattern using a back-reference such as `\1`. From the `MatchObject`, `groups` can be accessed with `m.group(n)` for an individual group, or `m.groups()` to return a tuple of all captured groups. Calling `m.group()` without a parameter returns the full matched string.

Other useful methods include **start()** and **end()**, which give the start and end positions of the match in the input string. The `MatchObject` also provides attributes such as `string` (the original input) and `re` (the compiled regex used). For efficiency, regular expressions can be pre-compiled with **re.compile()**.

re Substitution

Module `re` provides several functions

- For search and substitution
- Returns a modified string
 - `sub(pattern, replacement, string[, count, flags])`
- Returns a tuple of (modified string, num of changes)
 - `subn(pattern, replacement, string[, count, flags])`
 - Optional count determines occurrence to modify
 - Precompiled using `compile()`

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

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

QA



Python for Python Programmers
Python for Perl Programmers

98

The `re.sub()` function performs substitution in a string, similar in spirit to the awk scripting language. It returns the modified string after replacing all matches of the pattern. A related function, `re.subn()`, does the same but also returns the number of substitutions as part of a tuple. Both functions apply substitutions globally, scanning the string from left to right.

Another useful regex function is `split()`, which you will see on the next slide.

Regular expressions can also be compiled into a `RegexObject` using `re.compile()`. Compiling a pattern is more efficient when the same regex is used repeatedly, such as inside a loop. Once compiled, the object provides methods like `match()`, `fullmatch()`, `findall()`, `search()`, `split()`, `sub()`, and `subn()` to work with the text directly.

re Split

Similar in functionality to str.split()

- Uses a RE for the separator instead of a string
- `re.split(pattern, string[, max_splits=0, flags=0])`
 - Default for max_splits is zero which is unlimited
- Only use the re version if you need alternative separators

```
import re
line = "root:;0.0:superuser,/root;/bin/bash"
elems = re.split('[:;.,]', line)
print(elems)
```

```
['root', '', '0', '0', 'superuser', '/root', '/bin/bash']
```



QA

99

Python string objects provide a built-in **split()** method, which splits a string using a specified delimiter and returns a list of substrings. This is often useful when processing lines of text from a file. While strings in Python are immutable, the resulting list is mutable, meaning it can be modified and later joined back into a string with the `join()` method.

The `re` module extends this functionality with **re.split()**, which allows a regular expression to define the delimiter. This makes it much more flexible, similar to the split functions in languages like Perl and PHP. For example, `re.split(r'[:;.,]', text)` will split a string using any of the characters `: ; . ,` as delimiters. An optional argument lets you specify the maximum number of splits to perform.

From Python 3.1, **re.split()** also accepts a `flags` parameter, which can modify how the pattern is applied - for instance, using **re.IGNORECASE** to make splitting case-insensitive.

Shorthand Character Classes

A character class describes a set of characters

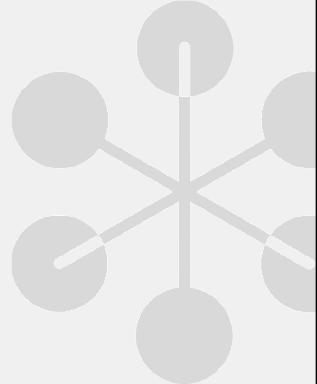
- For example, [a-z], [^A-Z], and [aeiou]

Python also supports several shorthand character classes.

- Preferred, concise, portable and easier to read

Class	Shortcut	Class	Shortcut
Digit	\d	Not Digit	\D
Word Char	\w	Not Word Char	\W
Whitespace	\s	Not Whitespace	\S
Word Boundary	\b	Not Word Boundary	\B
m = re.search(r'^ttyp\d\$', port)			\Z
if m:		print(port)	
		ttyp0	
		ttyp9	

QA



100

Python's re module supports the same single-character classes and sets familiar from **BRE** (Basic Regular Expressions) and tools like **sed** and **awk**.

In addition, Python provides **shorthand character classes** using escape sequences:

- \d → digit
- \s → whitespace
- \w → word character

These can also be used **inside character sets**.

- Example: [\da-fA-F] matches a hexadecimal digit.

Limitations: As of Python 3.13.1, the re module does **not** support **POSIX character classes** (e.g., [:digit:], [:space:], [:alnum:]).

Python also introduces an additional shorthand, \g, for referencing named groups, which will be discussed later. Shorthand classes are often preferred over POSIX forms because they are more concise, portable across tools, and easier to read.

Example – anchors

The ^ and \$ 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)
```

Renaming archive.1 to archive.2

- 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



101

QA

Anchors are special tokens in regex that match positions in the text rather than actual characters. The \$ anchor, for example, matches the end of a line or the end of the string. If you write 00\$, it will match two zeroes at the end of the text, or two zeroes followed by a newline at the end of a line.

The ^ anchor matches the start of a string. When used anywhere other than the beginning of a pattern, however, ^ loses its special meaning and simply represents a literal caret character.

By default, ^ and \$ apply to the start and end of the entire text. If the multiline flag (re.M) is used, they instead apply to the start and end of each individual line. To force a true start-of-text or end-of-text match in any case, you can use \A for the beginning and \Z for the end.

Regex also provides word boundary anchors. The token \b matches a word boundary (the position between a word character and a non-word character), without consuming any characters. Its opposite, \B, matches positions where no word boundary occurs, allowing you to find text embedded inside words. Be aware that the behaviour of \b with punctuation like apostrophes can sometimes

be unintuitive. One important caveat: inside a character class (...), \b does not mean a word boundary - it represents a backspace character instead.

Example – matching alternatives

The | character separates alternative words or

```
drink = 'A glass of Glenfiddich'  
if re.search(r'Glenfiddich|Lagavulin|Talisker', drink):  
    print("It's a Scottish Whisky!")
```

Use parentheses to group alternatives

- Required with text before or following alternatives

```
pattern = r'A (glass|bottle|barrel) of (Irn Bru|Buckfast|Glenfiddich)'  
  
if re.search(pattern, drink):  
    print("This drink is suitable for the Scots!")
```

QA



-102

The first example on the slide matches three possible patterns in the text; in this case, it successfully matches *Glenfiddich* (a good choice) and causes the if statement to evaluate to True. The second example expands this idea, allowing for nine different possible matches.

An important side effect of using parentheses for grouping is that the matched text is automatically captured in groups, which can then be extracted and reused - a feature we will explore later.

Example – repeat quantifiers

Quantifier characters repeat the preceding pattern

Repetition	Char	Example	Comment
0 or once	?	[+-]?	Optional sign
0 or many	*	[0-9]*	0 or more digits
1 or many	\+	[A-Z]+	1 or more uppercase chars

```
m = re.search(r"[+-]?\d\d\s*\d+", line) Optional sign, 2 digits, 0 or more whitespace, 1 or more digits  
m = re.search(r"boink+", sound) boink, boinkkk, boinkkkk  
m = re.search(r"(boink)+", sound) boink, boinkboink, boinkboinkboink
```

Greedy Vs Lazy

```
line = "root:x:0:0:superuser:/root:/bin/bash"  
"^.*:" root:x:0:0:superuser:/root:  
"^.*?:" root:
```

QA

103

The repeat quantifiers described here are greedy, meaning they consume as much of the text as possible without breaking the overall match.

For example, given the text:

```
line = "root:x:0:0:superuser:/root:/bin/bash"
```

The pattern “`^.*:`” will match “`root:x:0:0:superuser:/root:`” – the start of line all the way to the last colon.

Python also supports minimal (or lazy) quantifiers, which match the smallest amount of text necessary to satisfy the pattern. This is done by appending a question mark (?) after the repeat quantifier. For instance, with the same text, the pattern “`^.*?:"` will match only “`root:`”.

Example – interval quantifiers

Quantifier characters repeat the preceding pattern

Repetition	Char	Example	Comment
exact	{3}	[0-9]{3}	3 digits
min, max	{3,5}	\d{3,5}	Between 3 and 5 digits
at least	{5,}	[A-Z]{5,}	At least 5 uppercase chars
at most	{0,5}	[A-Z]{0,5}	Up to 5 uppercase chars

Match a U.S telephone number

- Start of text or a non-digit character
- Followed by 3 digits followed by a hyphen
- Followed by between 2 and 4 digits
- Followed by an optional whitespace
- Followed by between 4 and 8 digits
- Followed by a non digit character or end of text

QA

```
m = re.search(r"^(?|\D)\d{3}-\d{2,4}\s?\d{4,8}(\D|$)", phone)
```

104



In Python regular expressions, quantifiers control how many times the preceding character or group must appear to make a match. Interval quantifiers are written in curly braces and allow you to specify exact counts, ranges, or limits. For example, {3} matches exactly three occurrences, while {3,5} matches between three and five. Using {5,} enforces at least five repetitions, whereas {0,5} matches up to five. These quantifiers provide precise control compared to the simpler *, +, and ? operators.

The example on the slide demonstrates how these quantifiers can be combined to validate a U.S. telephone number format. The pattern begins by asserting either the start of the text or a non-digit character, followed by three digits and a hyphen. It then requires between two and four digits, an optional space, and between four and eight digits. Finally, the expression checks for either a non-digit character or the end of the text. This ensures that the match is properly bounded and does not overrun into unrelated characters.

By using interval quantifiers, Python's re module allows fine-grained pattern control. In this case, they help define flexible but strict rules for different segments of a phone number, accommodating variations in digit groupings while

still enforcing a valid overall structure.

Capturing Groups and back references

Python also allows substitution strings to reference groups within the pattern

- This makes it possible to create **self-referencing** regular expressions.
- Groups can be referenced by \n or \g<n>, where n is the group number
- Always use a **raw string** to avoid conflicts with Python string escapes
- Back references can also be used in the replacement string for sub() and subn()

```
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 2005-2025

QA

105

Python allows the use of **parentheses** to group characters or sub-patterns into a single unit, making it possible to apply quantifiers to the entire group. This is particularly useful when working with recurring blocks of text or repeated words. Such groups are also known as **capturing groups**.

The contents of a capturing group can be reused later in the same regular expression through a **back-reference**. This is done using \1, \2, etc., which refer to the text captured by the first, second, and subsequent groups. When groups are nested, numbering is assigned according to the order in which the opening parentheses appear. It is important to use **raw strings** (e.g., r'\1') for back-references, as \1 by itself has a special meaning in Python strings.

Python also provides an alternative back-reference syntax: \g<1>, \g<2>, and so on. This style is associated with **named capturing groups** (covered later) and has the additional benefit of supporting \g<0>, which refers to the entire matched string. Note that \0 is not supported in Python's re module. While back-references are powerful, they can make regular expressions slower to evaluate. Overuse of back-references may significantly increase processing time, so they should be applied thoughtfully and sparingly.

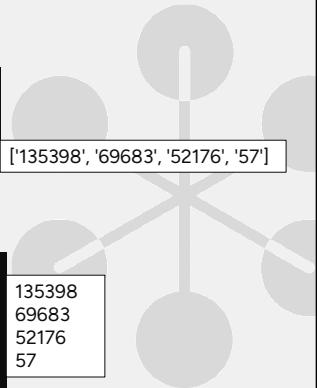
Finding multiple matches in Text

Finding all matches in a list using 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']



Iterating through all matches using 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.group())
```

135398
69683
52176
57

QA

106

By default, most re methods stop after finding the **first (leftmost) match**. In contrast, the `findall()` method scans through the entire text, collecting all matches into a list.

The `finditer()` method works in a similar way but returns an **iterator object**, which can be looped over to extract each match one by one. This is particularly useful when working with repeated patterns across multiple lines.

The `findall()` method was introduced in **Python 2.2**, and since **Python 3.7**, both `findall()` and `finditer()` allow non-empty matches to begin immediately after a preceding empty match.

RE Flags

Change the behaviour of the match

Long Name	Short	Embedded	Comment
re.IGNORECASE	re.I	(?i)	CaSe insensitive match
re.MULTILINE	re.M	(?m)	^ and \$ match start/end of line
re.DOTALL	re.S	(?s)	DOT also matches a newline
re.VERBOSE	re.X	(?x)	Whitespace is ignored for comments

- Can be embedded in the RE
- Can be applied to parts of the string (Modifier Spans, Py3.6)
- Can be specified as an optional argument to search, match, split, sub etc.
- Can be modified 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)
```

QA



107

The optional **flags parameter** was introduced in Python 3.1 and can be used with re methods to change how a pattern is matched. For example, **re.IGNORECASE** (or its shorthand **re.I**) makes matches case-insensitive.

It might seem that short names are simply the first letter of the long names, and that the embedded syntax is just those letters in lowercase, but this is not always true. For example, the S and X flags exist for compatibility with other regular expression engines rather than following this convention.

The examples on the slide illustrate different flag usages:

- The first two combine **IGNORECASE** and **MULTILINE**, so the pattern will find “john” in any case at the start of the text or after a newline.
- The third demonstrates a **modifier span**, where only the letter j is matched case-insensitively. This allows “John” or “john” but not “JOHN”.

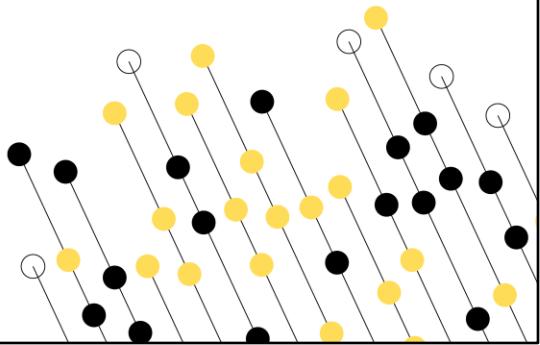
When flags are **embedded inside the regex**, the order of the single-character modifiers does not matter. When flags are passed to re methods, they are combined using a binary **OR (|)**, also in any order. Flags and embedded attributes

can be mixed, but doing so may reduce readability. The flags, `re.ASCII` and `re.LOCAL`, are deprecated since Python 3.5) or only for byte objects.

Review & Lab

Answer review questions

Your instructor will guide you which lab exercises to complete



Review Questions

1. Which `re` method returns all non-overlapping matches of a pattern as a list?

- a. `re.match()`
- b. `re.findall()`
- c. `re.search()`
- d. `re.finditer()`

2. What does the regular expression `r"\broot\b"` match?

- a. The substring `root` anywhere in the text
- b. `root` only at the beginning of the text
- c. The word `root` as a standalone word
- d. Any substring containing the letters `r`, `o`, `o`, and `t` in order

QA

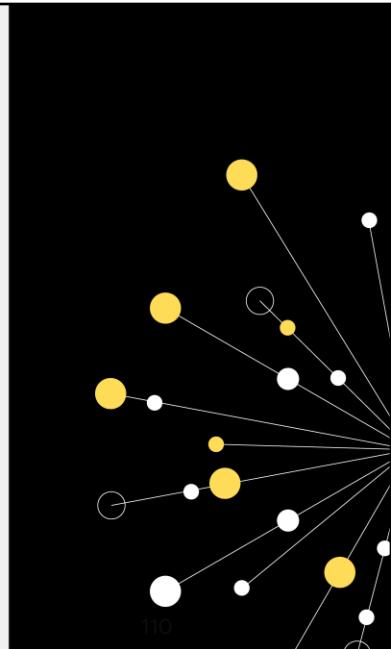
Answers on next page..



Review Questions

3. Which of the following correctly applies *both* the IGNORECASE and MULTILINE flags?
 - a. re.search(pattern, text, re.I + re.M)
 - b. re.search(pattern, text, re.IGNORECASE or re.MULTILINE)
 - c. re.search(pattern, text, re.I | re.M)
 - d. re.search(pattern, text, [re.I, re.M])
4. Which does the back-reference \1 represent in a regex pattern?
 - a. The first character of the input string
 - b. The contents of the first capturing group
 - c. A literal backslash followed by the digit 1
 - d. The number of matches found so far

QA



Correct answers:

1. Which re method returns all non-overlapping matches of a pattern as a list?
 - b. re.findall()

2. What does the regular expression r"\broot\b" match?

- c. The word *root* as a standalone word

Explanation: The escape chars \b match a word boundary character so pattern will match a standalone word

3. Which of the following correctly applies *both* the IGNORECASE and MULTILINE flags?

- c. re.search(pattern, text, re.I | re.M)

Explanation: Flags are separated using the | character.

4. Which of the following is true about Python's built-in functions

- b. The contents of the first capturing group

Explanation: The back-reference \1 tells the regex engine to match the same text that was previously captured by the first pair of parentheses in

the pattern.

Summary

Regular Expressions are widely used across many tool and languages

In Python, regex operations return a Match Object when a match is found

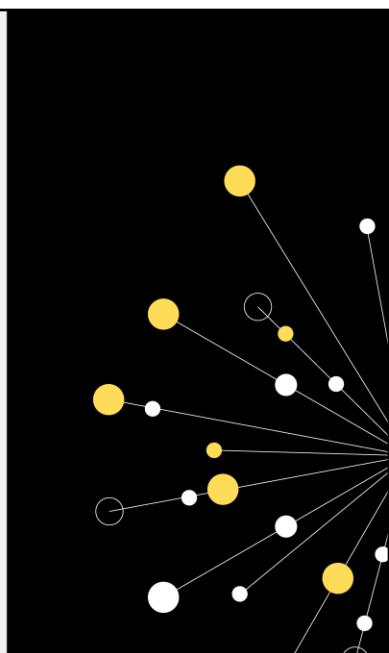
Several key functions are available in the re module, including:

- match() – check for a match at start of the string
- search() – finds the first occurrence of a pattern
- findall() – returns all matches as a list
- finditer() – returns an iterator over matches
- sub()/subn() – performs substitution
- split() – split text using regex patterns

Regex supports a rich set of character classes:

- Character classes, quantifiers, anchors, grouping, back-references and flags

QA

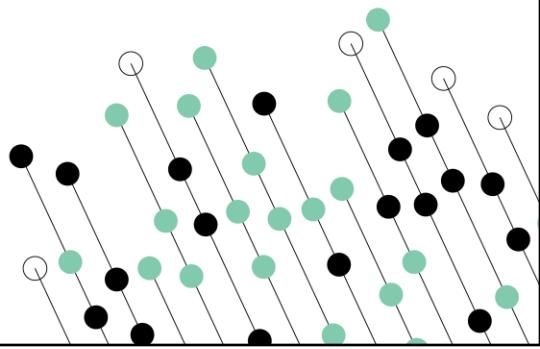


Module 7 - Data storage and file handling

7: Data Storage and File Handling

8: Functions

9: Advanced Collections

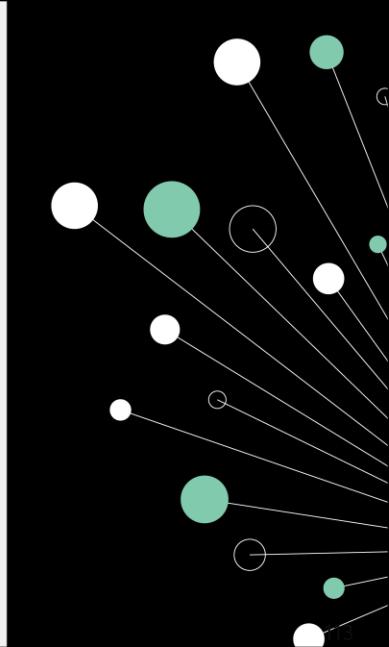


Learning objectives

By the end of this chapter, learners will be able to:

- Write code that reads and writes data sequentially and randomly to and from text and binary files
- Understand how to read and write to standard streams
- Preserve complex objects using Pickles and Shelves
- Use compression techniques to reduce file footprints

QA



Python has input/output features which are very easy to use, with syntax reminiscent of C and PHP.

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?

QA



114

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.

New file objects

File objects are created with the `open` function:

```
fh_object = open(filename, [mode='rt'], [buffering=-1], encoding=None, errors=None,  
                  newline=None, closefd=True, opener=None)
```

Valid open modes :

Mode	Meaning
'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

t=text, b=binary mode

buffering=1 (text)
buffering=4096|8192 (binary)

Info for other parameters:
`>>> help(open)`

QA

```
fh_object.close()
```

115

Files in Python are accessed through **file objects**, created using the built-in `open()` function. A 'b' should be appended to the mode for **binary files**, while 't' (for **text files**) is optional since it is the default.

Files are automatically **closed and flushed** when the program exits or when the file object is destroyed (for example, when it goes out of scope). However, it is **good programming practice** to close files explicitly by calling the `close()` method, or more commonly, by using a **with statement** (context manager) to ensure proper cleanup.

The mode 'x', introduced in **Python 3.3**, is used to **exclusively create and open** a new file. If the file already exists, an exception is raised: `FileExistsError: [Errno 17] File exists.`

The **buffering** argument controls how data is buffered:

- 0 Unbuffered (binary mode only)
- 1 Line-buffered (usually for text files)
- >1 Use a buffer of the given size
- 1 Use a system-appropriate buffer size (Default)

Other optional parameters include encoding, errors, newline, and closefd, though these are rarely needed. From **Python 3.3**, the **opener** parameter was added, allowing a custom file-opening function to be supplied.

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()
```

QA

116

Once a **file object** has been created, it can be read using one of several available **methods**.

The `read()` method reads data starting from the current file position. You can optionally specify the number of **characters** (in text mode) or **bytes** (in binary mode) to read.

Related methods include:

- `seek(offset)` Moves the current file position to the specified location.
- `tell()` Returns the current position within the file.

To read a single line of text (a record terminated by a newline character), use `readline()`. This is one of the most common ways to process text files in Python. Note that the newline character (\n) is included in the returned string; it can be removed using slicing (`[:-1]`) or the `rstrip('\n')` method.

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="")
```



In the first three examples:

- The first (lines) reads the entire file into **a single string**.
- The second (llines) also reads the whole file, then splits it into a **list of lines** using splitlines().
- The third (linelist) similarly produces a **list**, but the newline characters are **retained** in each element.

When reading a file line by line (a common requirement), it is generally more efficient to use the **file iterator** directly in a for loop. Using readlines() in a loop first loads the **entire file into memory** as a temporary list, then iterates through that list - which can be inefficient for large files.

Note the use of the **end** parameter in the print() statements. This prevents Python from printing an extra newline, since each record already ends with one.

Finally, avoid calling seek() (random access) inside a for loop that iterates over the file - it interferes with the file iterator and may cause an **infinite loop**.

Managing file handles using try/except/else/finally

Under very rare circumstances, a file could be left open

- An error could cause 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
- Else block only executes if try block succeeds
- Finally block always executes

First try block protects
the open() call

Nested try block protects the
write, and finally block ensures
file handle is always closed

```
try:  
    fh_out = open('spam.txt', 'wt')  
except FileNotFoundError as err:  
    print('Blooming Vikings!')  
  
else:  
    try:  
        fh_out.write('Spam, spam, spam!')  
    finally:  
        # Always close file handle after use  
        fh_out.close()
```

QA

8

When working with **external resources** such as files, it's important to **release them properly** once you're finished reading or writing. Failing to do so can cause your program to retain resources indefinitely, leading to **memory leaks** or **loss of unwritten data**.

Most programs include a **setup** and **teardown** phase - with the teardown phase handling clean-up tasks such as closing files, releasing locks, or shutting down network and database connections.

When writing to files, data is often **buffered in memory**, and if the program exits unexpectedly, that data may be lost unless the file is properly closed. Calling the file object's **close()** method ensures that any buffered data is flushed to disk and that the resource is released.

To guard against exceptions that occur before a resource is closed, Python offers two main strategies:

- Using a **try/finally** construct, or
- Using a **context manager** with the **with** statement.

In the first approach, the `open()` function is wrapped in a `try/except/else` block.

Any errors raised in the try block can be handled in except, and the finally block, which always runs safely closes the file.

Managing file handles using Context Resource Manager

Under very rare circumstances, a file could be left open.

- An error could cause an unhandled exception.

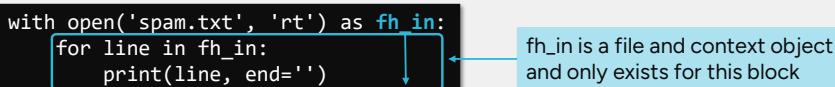
Some python classes are **context managers**.

- `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='')
```



A callout box points from the variable `fh_in` in the code block to a text box containing the explanatory text: "fh_in is a file and context object and only exists for this block".

QA

119

PEP 343 introduced a cleaner and safer way to manage external resources, such as file handles, using the **with statement** and the **Context Manager Protocol**. This approach replaces many uses of try/finally, producing more elegant, efficient, and reliable code.

The `with` statement creates a **runtime context** managed by an object that defines `__enter__()` and `__exit__()` methods, which automatically handle setup and cleanup.

A common example is the **file object**, used as follows:

`with expression [as variable]:`

 BLOCK

The expression returns a context object whose `__enter__()` method is called at the start of the block, and `__exit__()` when the block ends - even if an exception occurs. If the `as` keyword is used, the variable is bound to the object returned by `__enter__()`.

This mechanism provides **automatic resource management** and avoids problems with destructors or unclosed files. You don't need to retrofit existing

code, but it's strongly recommended for new development.

Writing to files from Python

Open a file handle with `open`:

- Specifying write or append.

```
output = open('myfile', mode='w')
append = open('logfile', mode='a')
```

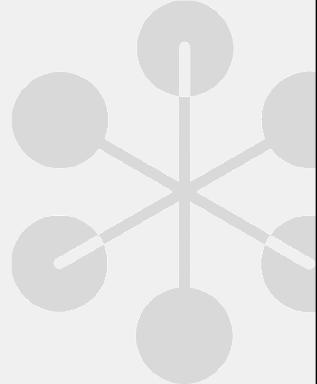
- Write a string.
- Append "\n" to make it a line.

- 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)
```



QA

120

Just as there are several ways to read from a file, there are also multiple ways to **write** to one.

The `write()` method writes characters (or bytes, in binary mode) to the current file position - but remember to include a **newline character (\n)** yourself when writing text files.

The `write()` method returns the **number of characters or bytes written**, depending on how the file was opened. This behaviour was added in later versions of Python; older versions did not return a value.

To write multiple records from a list, use the `writelines()` method.

Despite its name, it **does not** automatically add newline terminators - each line in the data must already include one.

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())
```



QA

121

Many programming languages, such as **C** and **Perl**, distinguish between **text** and **binary** file modes on Windows, because the Windows operating system handles these formats differently.

On **UNIX-like systems**, there is no such distinction - files are simply sequences of bytes.

In **Python 3** (as in Java and .NET), **strings are Unicode**, meaning they represent multi-byte characters by default. When a file is opened in **text mode** (the default), Python automatically handles the conversion between text (Unicode strings) and the underlying byte data.

Writing a Python string will appear as ordinary text in the file, provided the characters are within a single-byte character set such as ISO Latin-1.

Opening a file in **binary mode** ('b') forces single-byte operations using **bytes** objects.

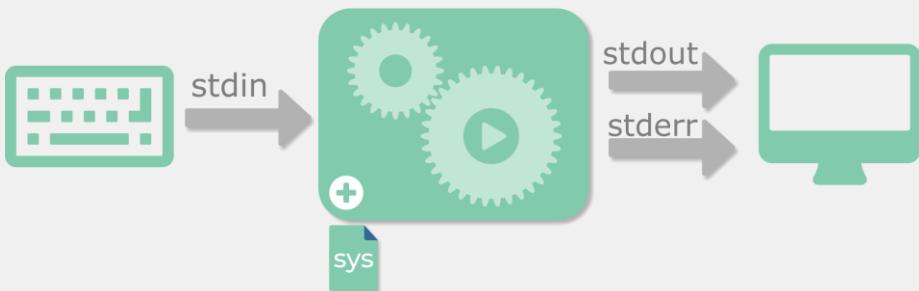
Fortunately, converting between strings and bytes is straightforward, and most string methods are also available for byte objects. Text mode also manages

platform-specific line endings automatically (\n on UNIX, \r\n on Windows). In **binary mode**, however, no such conversion takes place - if you need line endings, you must include them explicitly.

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`.



QA

122

Like Unix and Linux, **Windows** provides three standard I/O channels for handling keyboard input, program output, and error messages. These are special file-like interfaces that allow a process to interact with the user and display information on the screen. The three channels are traditionally numbered **0**, **1**, and **2**, and are named **stdin**, **stdout**, and **stderr** respectively.

Python supports these standard streams, allowing both built-in and library functions to read input and produce output or error messages without explicitly opening or closing files. To access them by name, you need to **import the `sys` module**.

By default, the built-in **`input()`** function reads from `stdin`, and the **`print()`** function writes to `stdout`.

Standard streams

The **sys** module exposes **stdin**, **stdout**, **stderr** as open file objects.

```
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.

- 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
```

QA

123

The three standard I/O streams, **stdin**, **stdout**, and **stderr**, can be accessed directly through the **sys** module. Each is a **file-like object** and can be used just like any other file handle.

The **input()** function reads from **stdin**, which is typically connected to the keyboard but can be redirected from another source. If the **readline** module is imported first, **input()** gains line-editing and command-history capabilities. (In Python 2, this function was called **raw_input()**.)

On Windows cmd.exe, line endings are represented by "**\r\n**". A bug in **Python 3.2.0** caused the newline character to be stripped while leaving "**\r**" behind. Although this was fixed in **Python 3.2.1**, using `str.rstrip()` remains a useful safeguard, especially if users might include trailing spaces.

By default, standard streams operate in **text mode**, but they can be used in **binary mode** by launching Python with the **-u** option, for example:

```
python -u myprog.py
```

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)
```

Text written to stderr will be displayed in red on python console

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

QA

124

In **Python 3**, the **print()** function can write directly to a file using the **file=** parameter. In earlier versions of Python, output redirection used a different syntax, for example:

```
print >> output, "Hello" # Python 2
```

Buffering can be disabled either by running Python with the **-u** command-line option or by setting the **PYTHONUNBUFFERED** environment variable to any non-empty value.

The final example involves the UNIX command **tail -f**, which continuously displays new lines as they are appended to a file. It works by reading to the end of the file, pausing briefly (typically one second), and then checking for new data. If new lines have been added, they are displayed; otherwise, the program waits and repeats. One unusual aspect of this example is the use of **seek()** to reset the current file position. When the end of the file is reached, the file position becomes invalid, so the line

```
fh_out.seek(fh_out.tell())
```

moves the pointer back to the same physical position, allowing the program to continue monitoring the file for new content.

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="")
```

QA

125

Most file I/O operations, especially with text files, are **sequential**. Accessing a specific record this way can be inefficient, particularly if repeated. A faster alternative is **random (positional) access**, where the program seeks directly to a known location in the file. To enable this, an **index** of record positions can be built, for example, using a **dictionary** that can later be saved with **pickle**.

Files should be opened in **binary mode**, particularly on Windows, where text mode automatically handles carriage returns ('\r') and can cause byte offsets to be misaligned. Binary mode ensures that data is read and written as **byte objects**, not strings.

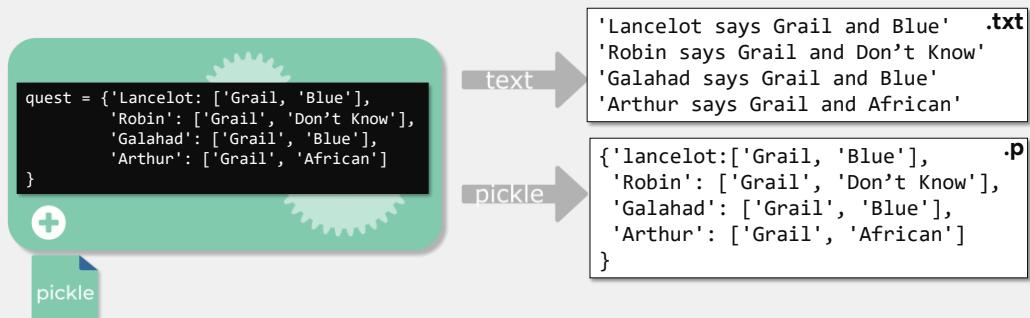
A **while loop** is preferred over a for loop when building an index, because the file iterator may read ahead into an internal buffer, making `tell()` unreliable. Once record positions are known, the `seek()` method can be used to move to a given offset and perform reads or writes.

Offsets are measured relative to: the **start of the file** (`whence=0`), the **current position** (`whence=1`), or the **end of the file** (`whence=2`), and most systems allow seeking beyond the end-of-file.

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.



QA

126

We have already seen how to read, write, and append text or binary **data** to a file. However, the **original Python object structure** in memory is lost when doing so, making it difficult to reconstruct later. Think of it like trying to restore an empty tin can after it has been crushed — the contents may be saved, but the shape is gone.

Pickling is the process of converting a complete Python object structure into a **byte stream**, a process known as **serialisation**, while preserving both its **data** and **structure**. **Unpickling** is the reverse operation, reconstructing the original object from the stored byte stream. This is similar in concept to **JSON** in Java or JavaScript, but the pickle module supports serialisation in both **text** and **binary** formats.

Today, the pickle protocol is often used to **transmit data and commands** between processes, either locally or across a network, rather than just for saving to disk. For very large datasets, such as **NumPy** or **Pandas** dataframes, you may want to use specialised third-party tools like **Dask**, **PyArrow**, or **IPyParallel**, which provide more efficient serialisation methods.

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
Preserve caps object to filehandle outp

- Without pickle, how would you preserve complex data structures?

```
import pickle  
  
inp = open('capitals.p', 'rb')  
caps = pickle.load(inp)  
inp.close()
```

Load object from filehandle

QA

127

Pickling is Python's way of storing its own object types in a file for later use.

Always open the file in **binary mode** when pickling or unpickling data.

Objects that can be pickled include:

- None, True, and False
- Integers, floating-point numbers, and complex numbers
- Strings, bytes, and bytearray
- Tuples, lists, sets, and dictionaries (provided they contain only pickleable objects)
- Classes defined at the **top level** of a module

Although the Python documentation notes that functions can also be pickled, this can be misleading - only the **function names** are stored, not their actual code. Code itself cannot be easily pickled; that's what **modules** are for. In fact, a compiled module (.pyc file) is conceptually similar to a pickle. Python's internal serialization mechanism, exposed through the **marshal** module, operates at a lower level than pickle and provides fewer features.

Pickle protocols

How would you like your data pickled?

- Currently 5 different protocols to choose from, pickle.protocol=n

Protocol	Meaning
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.DEFAULT_PROTOCOL]) + Preferred protocol
outp.close()
```

QA

- None of these protocols are secure.



Pickle files can be written in several formats. When in doubt, use the default format, which is automatically selected based on your Python version.

If you choose the ASCII format, you don't need to open the file in binary mode, but be consistent and use the same mode for both reading and writing. ASCII pickles are easier to debug, while binary format is preferred for compatibility with older Python programs. In most cases, sticking with the default is the safest and most efficient option.

Pickle Protocol Versions:

- Protocol 5 (Python 3.8+):** Introduced support for out-of-band data buffers, allowing applications to transmit metadata across separate communication channels (e.g., paired sockets); also improves performance for in-band data.
- Protocol 4 (Python 3.4+):** Added support for very large and complex objects (e.g., data frames) and improved efficiency. Note that it is not backward compatible with earlier protocols.
- Protocol 0–2 (Python 2):** Early protocols used in Python 2.x. Compatibility issues arose when creating pickles in Python 3.0 for use in Python 2, which were resolved in Python 3.1. However, protocols 0–2 cannot be exchanged

between Python 3.0 and 3.1.

For cross-version compatibility, use protocol 3 when sharing data between Python 3, and upgrade to at least Python 3.1 to exchange data with Python 2.

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') ← No mode or file suffix required
db['UK'] = 'London'
...
db.close() ← Same syntax as standard dict
```



```
db = shelve.open('capitals')
print(db['UK'])
db.close() ← close() does a sync() like a commit
```

QA

The shelve module is a convenient option when using pickling on a larger scale. It stores pickled Python objects in a **database file** (typically using dbm, depending on the platform) and allows access via **string keys**. However, you should not assume that the underlying database is portable between systems or compatible with non-shelve tools.

When a shelve database is opened, you can specify different pickle protocols if required. The opened database behaves just like a **standard dictionary**, supporting both reading and writing of objects.

Be aware that you should not allow multiple programs or threads to access the same shelve file simultaneously. Most operating systems prevent this through **file locking**, but doing so can still cause errors or data corruption.

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
fh_out = gzip.open('capitals.pgz', 'wb')
pickle.dump(caps, fh_out)
fh_out.close()
```

```
import pickle
import gzip
fh_in = gzip.open('capitals.pgz', 'rb')
caps = pickle.load(fh_in)
fh_in.close()
```



QA

130

For small datasets, compression may not reduce file size - in fact, the resulting file can sometimes be slightly larger.

When working with ZIP files, always open them in **binary mode** ('rb' or 'wb'), since they handle **byte streams** rather than text. It's also good practice to use a filename suffix such as **.gz** (or .tar, .bz2, etc.) to clearly indicate that the file is compressed.

Finally, don't confuse this with Python's built-in **zip()** function - it's unrelated. The zip() function combines multiple collections element by element, not file contents.

Other compression modules

Compression/decompression Modules

- All modules are in the Python Std Library
- Supports compression/decompression in memory or to/from files

Module	Comment
zlib	GNU zlib compression API
bz2	bzip2 compression
zipfile	Zip archive access (pkzip compatible)
tarfile	TapeARchive (tar) access
shutil	High level archiving operations

Last modification time of
file formatted to
day/month/year hh:mm:ss

QA

Example – Reading a Tar Archive

- How to **open/close** and inspect a .tar/tgz
- Uses **tarfile.is_tarfile()** to validate file
- Iterates through members using **getmembers()**

```
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)
        fm_time = time.strftime("%d/%m/%Y %X", tm)
        print(f"{mdata.name:<12} {fm_time}")

tfo.close()
```

131

The **zlib** module provides checksum functions such as **adler32** and **crc32**, which are useful for verifying **data integrity**. However, these checksums are not cryptographically secure.

The **bz2** module offers a file-like class, **bz2.BZ2File**, which can be used much like Python's built-in **open()** function for reading and writing compressed data.

The **tarfile** module supports standard archive operations. Its **getmembers()** method retrieves **metadata for all files** in an archive - not just their names. The example shown was run on Windows using a **.tar.gz** file originally created on Linux.

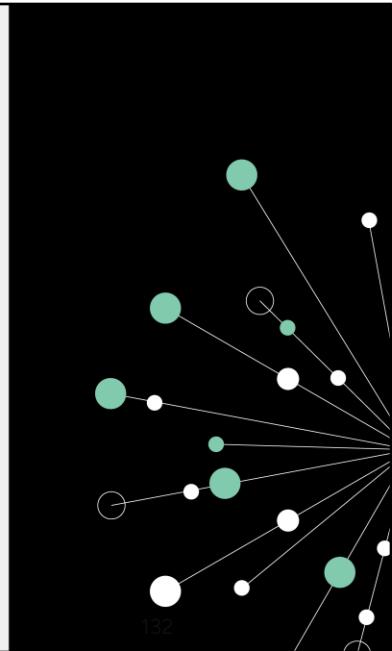
For a simpler, high-level interface, the **shutil** module provides generic file and archiving utilities, making it a convenient choice for common compression and extraction tasks.

Review Questions

1. When reading a file using the file object, what method is best for reading the entire file into a single string?
 - a. read()
 - b. readlines()
 - c. readline()
 - d. readall()

2. Which method is used to set the position of a file pointer?
 - a. tell()
 - b. seek()
 - c. ftell()
 - d. fseek()

QA



132

Answers on next page.

Review Questions

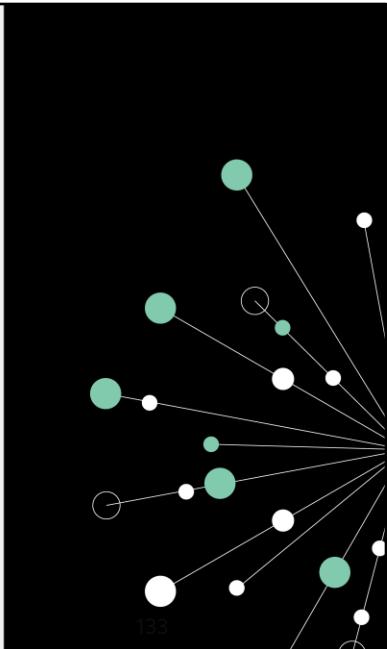
3. What is the main purpose of pickling in Python?

- a. To convert a Python object into a JSON string for web communication
- b. To convert a Python object into a byte stream so it can be saved to a file or sent over a network
- c. To compress a Python file to reduce its size
- d. To encrypt Python data for secure storage

4. What statement about the shelf module is true?

- a. It only supports plain text data
- b. It stores objects in memory, not on disk
- c. It allows objects to be stored in a file using a key, like a dictionary
- b. It replaces the pickle module entirely

QA



Correct answers:

1. When reading a file using the file object, what method is best for reading the entire file into a single string?

- a. read()

2. Which method is used to set the position of a file pointer?

- b. seek()

3. What is the main purpose of pickling in Python?

- b. To convert a Python object into a byte stream so it can be saved to a file or sent over a network

4. What statement about the shelf module is true?

- c. It allows objects to be stored in a file using a key, like a dictionary

Summary

Creating a file object:

- Use the open() function to create a file object

Reading from a file:

- Use read(), readline(), or readlines() methods
- Or iterate directly over the file object in a for loop

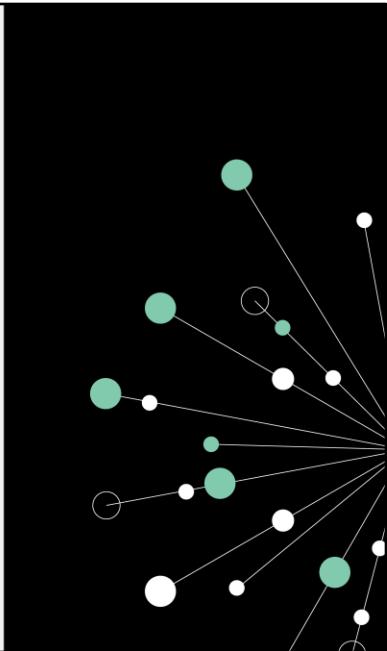
Writing to a file:

- Use write() or writelines() methods
- The print() function can also write to files
- Always close the file when finished or use a with statement

Additional notes:

- Many other file-handling methods are available.
- Pickling preserves Python objects to a file.
- Shelve can store multiple pickled objects using key access.
- Compression reduces file size and saves disk space.

QA



Python provides **database drivers** for most popular relational database systems, and many are bundled with standard or third-party distributions. These drivers adhere to a common interface defined by the Python Database API Specification v2.0 (DB-API 2.0).

To interact with a database:

1. Establish a connection using a **connection object**.
2. Use this connection to create a **cursor object**, which handles queries and transaction control (**commit** and **rollback**).

The **cursor** is the primary tool for executing SQL statements and getting results.

Some Common Cursor Methods

- `arraysize` Number of rows fetched by `fetchmany()`
- `close()` Close the cursor
- `description` Metadata about the result set or table
- `execute()` Execute a single SQL statement (supports placeholders)
- `fetchall()` Retrieve all remaining rows
- `fetchone()` Retrieve the next
- `rowcount` Number of rows affected by the last operation

Python is shipped with **SQLite**, a lightweight, single-file relational database that

requires no server setup and is ideal for small to medium-sized applications.

Example - SQLite from Python

Use the `sqlite3` module.

- Bundled with the Python release.
- Example using `cursor()`, `execute()`, `fetchall()` and `close()` methods:

```
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()
```

QA

136

This example demonstrates how to use Python's `sqlite3` module to connect to a relational database and execute a simple **SQL query**. The `connect()` function establishes a connection to a database file named **whisky** - if the file does not already exist, SQLite automatically creates it. From this connection, a **cursor object** is obtained using `db.cursor()`. The cursor is the main interface for sending SQL commands and retrieving results.

The `execute()` method runs an SQL SELECT statement that retrieves brand names and their corresponding regions from two tables, **BRANDS** and **REGION**. The WHERE clause ensures that the tables are joined using a common key (`REGION_ID`), while the ORDER BY clause sorts the results alphabetically by brand name. Once the query executes, the `fetchall()` method retrieves all matching rows as a list of tuples. The for loop iterates through each row, and the `print()` statement uses formatted string output to neatly display the brand and region names in two left-aligned columns, each 30 characters wide.

Finally, the `db.close()` call closes the database connection, ensuring that all resources are released properly. In modern Python code, it's often recommended to use a **context manager** (with `sqlite3.connect('whisky')` as `db`):

to handle connection closure automatically, even if an error occurs.

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!
```

QA

137

The first challenge when reading **binary data** is determining its **format**. Unless the data was originally written by Python (or via the Python API in another language such as C), it won't map directly to Python's built-in types - a common issue in all high-level languages.

The example shows a simple **C struct** written to a file. It contains **two integers (int)**, a **double-precision value (double)**, and an **array of 80 characters** (a C-style string ending with a null byte).

To interpret this in Python, the **struct.unpack()** function is used with a format string, here "iid80s", which specifies two integers, one double, and an 80-character string. Because C strings are single-byte, the resulting Python bytes object is converted to text using **.decode()**.

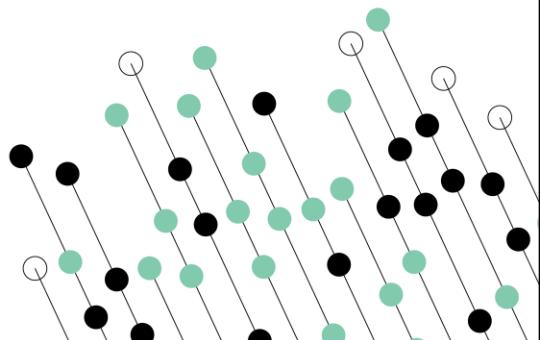
Note: Be careful of **padding bytes** added by C compilers for alignment, as these vary between 32- and 64-bit systems and can be changed with #pragma pack. Also watch for **endianness** differences, which are detailed in the Python struct module documentation.

Module 8 - Functions

8: Functions

9: Advanced Collections

10: Modules and Packages

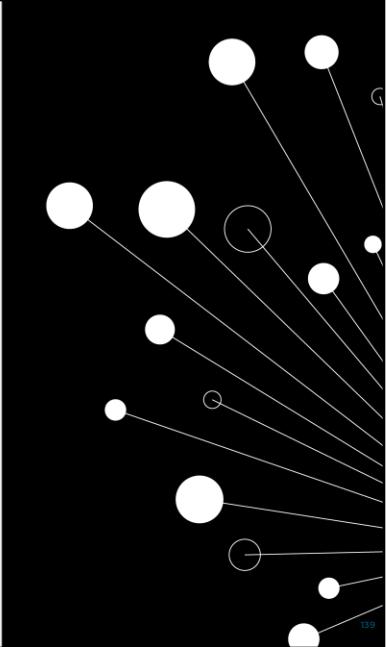


Learning objectives

By the end of this chapter, learners will be able to

- Create, call, return values and pass parameters to Python functions
- Understand how to use Variadic functions
- Add Annotations to function declarations
- Understand variable scope
- Nest functions
- Implement Python docstrings
- Define and invoke Lambda functions

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



QA

140

Python functions follow a syntax similar to conditional statements. A function is defined using the **def** keyword, followed by its name, an **argument list**, and a **colon**. The indented block beneath the definition contains the statements that make up the function body.

The argument list specifies the parameters passed to the function, with optional **default values**. This approach resembles that used in languages such as **awk**, **PHP**, and certain **mainframe scripting languages**, and shares some similarities with **C++**. The number of arguments provided by the caller must match those defined in the function (unless defaults are used). Function arguments in Python can be of **any data type**.

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 caller's variables.

```
def change_list(inlist, val, times):
    inlist += str(val) * times

mylist=[]
change_list(mylist, 'h', 8)
print(mylist)
```

QA

141

In Python, **arguments are passed by object reference** (sometimes described as *pass by assignment*). This means that when a function is called, the parameter name inside the function becomes a **local reference** to the same object that was passed in. The reference itself is passed **by value**, so reassigning it within the function has no effect outside the function.

However, if the object is **mutable** (for example, a list or dictionary), then changes made to its contents inside the function will affect the original object in the caller. To prevent this, you can pass a **copy** of the data structure instead, such as a slice of the list:

```
print_thing(mylist[:]) # Passes a shallow copy
```

Alternatively, you can convert the list to an **immutable type**, such as a tuple:

```
print_thing(tuple(mylist)) # Raises an error if modification is attempted
```

In general, it's considered better practice to return new values from functions rather than modify arguments in place. This avoids side effects and makes function behaviour clearer and easier to reason about.

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: {:.5.2f} Vat: {:.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
```

QA

142

In Python, a function can define **parameters with default values**.

For example, consider a function that calculates **VAT** on an item. In the UK, the VAT rate was historically **17.5%**, which makes a sensible default. However, since the rate may vary, it's better to define it as a **default parameter** rather than hard-coding it into the function.

When using default values, remember that only parameters on the right-hand side of the argument list can have defaults. Once a parameter is given a default value, all parameters that follow must also have defaults.

For instance:

```
def print_vat2(vatpc=17.5, gross):
```

will cause a **syntax error**:

```
File "functions.py", line 31
```

```
    def print_vat2(vatpc=17.5, gross):
```

```
SyntaxError: non-default argument follows default argument
```

To fix this, reverse the order or assign defaults to both parameters, for example:
def print_vat2(gross, vatpc=17.5):

Passing parameters - Review

Passing parameters

```
def my_func(file, dir, user='root'):
    print('file: {}, dir: {}, to: {}'.
          format(file, dir, user))
```

- By position:

```
my_func('one', 'two', 'three')
        file: one, dir: two, to: three
```

- By default:

```
my_func('one', 'two')
        file: one, dir: two, to: root
```

- Or by name:

```
my_func(file='one', user='three', dir='two')
        file: one, dir: two, to: three
```

QA

143

Before we look further at parameter passing, we should review the mechanisms we have seen so far.

Enforcing named parameters

Use * 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: {:.5.2f} Vat: {:.5.2f}'.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
```

*, can be placed anywhere
in the parameter list to
enforce named passing
for following parameters

Attempting to pass positional parameters will fail.

```
print_vat(15, 9.55)
```

```
TypeError: print_vat() takes exactly 0 positional arguments (2 given)
```

QA

144

The example demonstrates a technique that **forces the caller to specify parameters by name** rather than by position. This improves readability and helps prevent errors caused by incorrect argument order.

In the example, all parameters have default values, so calling the function with no arguments is still valid.

By convention, any parameters defined **after an asterisk (*)** must be passed **as keyword arguments**, while those **before the asterisk** may still be passed **positionally**. This syntax is specific to Python 3.

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')
```

QA

145

By default, Python functions accept a **fixed number of parameters**. However, if the arguments are stored in a **tuple or list**, they can be passed into a function by **unpacking** them with a leading asterisk (*) in the function call. The container must, however, contain the **exact number of elements** expected by the function.

You can pass **any iterable** this way, typically a tuple or list, though technically even a string would work (each character would be treated as a separate argument).

Conversely, a function can be defined to **collect multiple positional arguments** into a single tuple using the same * syntax. In C/C++ terms, this is known as a **variadic function**, meaning it can accept a **variable number of parameters**. The named parameter (for example, *files) becomes a tuple of all received arguments. Remember that tuples are **immutable**, but their elements may still reference **mutable objects** that can be modified.

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}
```

Capture parameters
in a dictionary

Named parameters
passing must be used

Use ****** to unpack caller's parameters from a dictionary.

```
argsdict = dict(vatpc=15, gross=9.55, message='Summary')  
print_vat(**argsdict)
```

QA

146

Keyword parameters (commonly referred to as **kwargs**) allow arguments to be passed as a **dictionary** of name–value pairs.

To **unpack** named parameters when calling a function, use **two asterisks (**)** as a prefix. Inside the function, these parameters are automatically collected into a dictionary, meaning their **order does not matter** - a useful feature when dealing with functions that have many parameters.

A parameter prefixed with ****** must appear **at the end of the parameter list** in a function definition. Placing it elsewhere will result in a **syntax error**.

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' ]
```

Function can be
called as a r-value or
part of an expression

QA

147

A function in Python can return a reference to **any type of object**, including a list, tuple, or dictionary.

If a function does not explicitly return a value, and the calling code attempts to use one, Python automatically returns **None**. This special built-in object represents the **absence of a value**. It does not mean zero, but in certain contexts it can behave similarly, for example, **None evaluates to False** in a Boolean expression.

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__)
```

QA



148

Function annotations were introduced in **Python 3** to allow arbitrary **metadata** to be attached to function parameters and return values. They are entirely **optional**, but can be useful for documenting how a parameter or return value is intended to be used. An annotation can be any valid **expression** (for example, str or "VAR, gross, message"). These expressions are evaluated at **compile time** and have **no effect at runtime**.

Although annotations may resemble **type declarations** in other languages, they do **not enforce type checking**, Python remains a dynamically typed language. Python 2 did not support annotations, but similar functionality could be achieved using decorators (see *PEP 318*) or by embedding type hints in a function's docstring.

Even in Python 3, annotations are largely **ignored by the interpreter**. Their value comes from **third-party tools** and libraries, for example, type checkers such as **mypy**, documentation generators, or frameworks that use annotations for **validation** or **help messages**. Annotations for a function can be accessed programmatically through the special attribute:

```
print(function_name.__annotations__)
```

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.
 - But are local to the current module, or namespace.

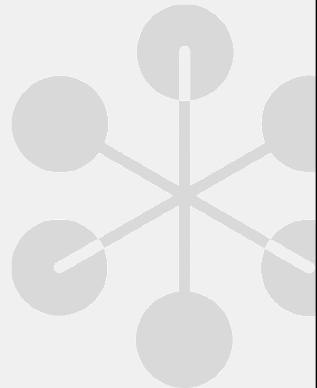
```
result = 3
def scope_test1():
    result = 42
scope_test1()
print(result)           3

def scope_test2():
    global result
    result = 42
scope_test2()
print(result)           42
```

Assignment means it is a local variable

Global variable is changed

QA



149

In Python, the **scope rules** determine whether a variable is treated as **local** or **global**. If a variable is **used** in a function without being assigned a value, Python assumes it refers to a **global variable**. However, if a variable is **assigned** within the function, it is considered **local** - unless explicitly declared as **global**.

Variables created inside a function exist only within that function's **local scope** and are discarded when the function exits. For example, two variables named `result`, one inside a function and one outside, are entirely independent. Assigning a new value to the local `result` does not affect the global one.

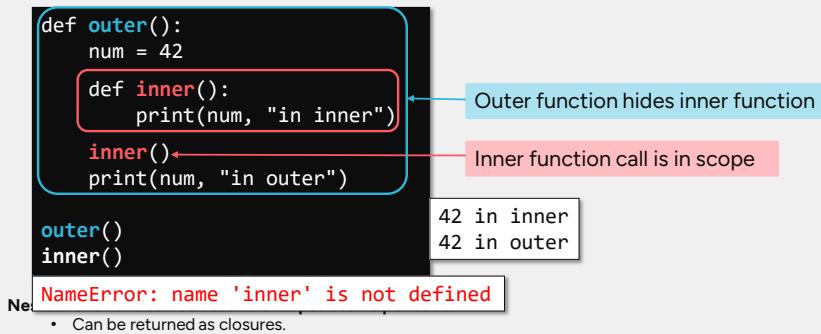
If you need to modify a global variable from within a function, you must declare it with the **global** keyword **before** using it. Although Python may allow the declaration later in the function, doing so can raise syntax warnings.

In practice, **global variables should be avoided** wherever possible. They can make code harder to maintain and are especially problematic in multi-threaded applications, where concurrent access can lead to unpredictable results.

Nested functions

A function can be defined within another function.

- This has the same scope as any other object.



QA

150

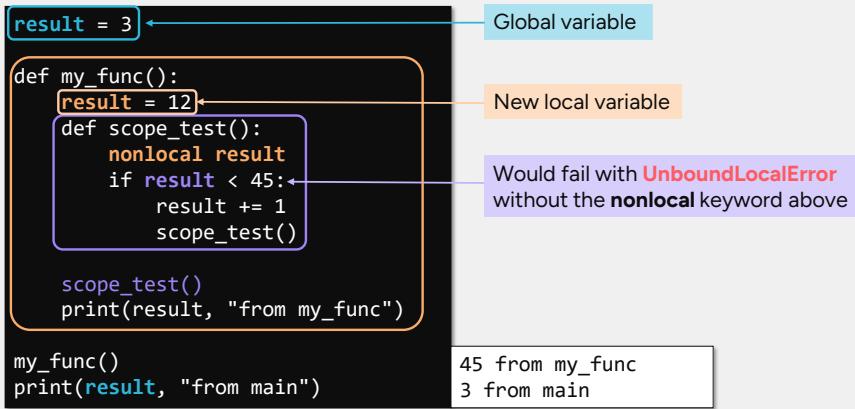
Defining a **function within another function** is a feature not supported by all programming languages, but in Python it can be very useful. It is commonly used for **simple helper functions**, for creating **closures**.

In the example, the inner function can access the variable `num` even though it is not declared as global - this works because `num` remains **in scope** within the enclosing function. However, if the inner function were to **assign** a new value to `num`, Python would treat it as a **new local variable**, visible only within that inner function.

Note: A **closure** in Python is a function that **remembers the values** from its **enclosing scope**, even after that scope has finished executing. In other words, the inner function “closes over” the variables it needs from the outer function, allowing it to access and use them later. Closures are often used to **preserve state**, create function factories, or implement encapsulation without using classes.

Variables in nested functions

Since scopes may be nested, we need to indicate that:



QA

151

Python's clean syntax and dynamic typing mean there is no explicit way to declare new variables. This can sometimes cause confusion over when a new variable is created and when an existing one is reused. We have already seen how the `global` statement identifies a variable as belonging to the global scope. But what about *enclosing* scopes? Python is unusual in allowing local or *nested* functions. To indicate that a variable belongs to an enclosing scope rather than the local one, we use the `nonlocal` keyword.

In the example on the slide, if `nonlocal` were omitted, the `if` statement would test a variable before it was defined, since `result` is not declared as `global` (nor should it be). Declaring it as `nonlocal` reuses the `result` variable from the outer function, allowing the recursion to work correctly.

If the inner function (`scope_test`) is returned and later called externally, it still retains access to the `nonlocal` variable:

```
def my_func():
    return scope_test
```

```
rfunc = my_func()
rfunc()
```

This behaviour is known as a **closure**, which in Python 2 could only be achieved

using globals.

Function and user documentation



Python supports Comments for developers:

- Useful for maintainers, but not designed for users.

Python supports Docstrings for users:

- 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")
```

Use triple-quoted
multi-line strings

```
>>> help(my_func)
Help on function my_func in module __main__:
my_func()
my_func has no parameters and prints 'Hello'.
```

QA

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__`.

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

Expression

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))
```

QA

153

The term *lambda* originates from Lisp and can sound a little intimidating. Its Greek and mathematical flavour gives it an aura of complexity, but in Python, lambda functions are actually simple, think of *lambda* as meaning ***inline function***.

Lambda functions don't do anything that a normal def-defined function can't; they're just shorter and quicker to write. They're ideal for small, one-line functions that don't need a name. A lambda expression returns a reference to a function, which can be stored in a variable and used just like a regular function name.

Many Python built-ins accept a *callable object* as an argument. This can be a function name, a variable referencing a function, or a lambda expression written directly inline. In the example with map(), the lambda adds 1 to each item in the list.

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,-,...
```

QA

154

Lambda functions are often used as **arguments to built-in functions**. In this example, we could have defined a separate function for the comparison, but instead we use a lambda that takes one parameter (c) and returns the key field in the appropriate format for comparison.

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)
```

Example of a dict comprehension



Note: This code uses *dictionary comprehensions*, which has not been covered yet!

155

The regular expression substitution functions `re.sub()` and `re.subn()` allow the second argument (normally a replacement string) to be a function instead. This function can be a pre-defined named function or a lambda expression.

In this example, we construct a dictionary where the keys come from a list of character ranges and the values from the NATO phonetic alphabet. It's important that both lists are in the correct order.

The substitution pattern matches any alphanumeric character and captures it using parentheses `(\w)`. Each matched character is then used as a key to look up its corresponding value in the dictionary.

The output in this case will be:

whisky golf zero seven oscar kilo delta

Review Questions

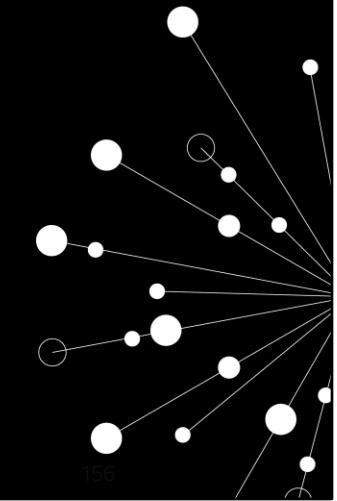
1. What will the following program print?

```
def calculator(num1, num2):
    first_value = num1 + 7
    second_value = num2 - 6
    return first_value

calculator(13, 10)
```

- a. 4
- b. 20
- c. 24
- d. None of the above

QA



156

Answer on next page.

Review Questions

2. What will the following program print?

```
def calculator(num1, num2):
    fval = num1 + 7
    sval = num2 - 6
    tval = trebler(sval)
    return tval

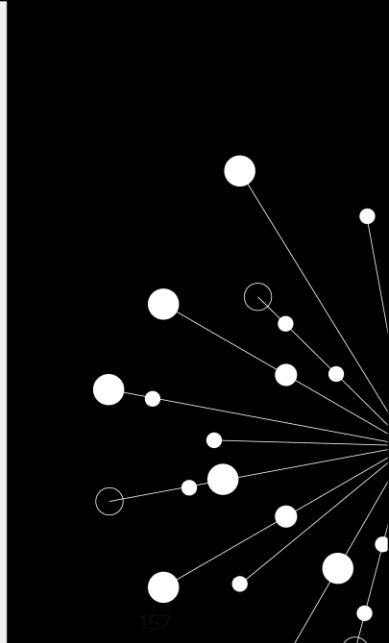
def trebler(aval):
    return (aval * 3)

calculator(13, 10)
print(sval)
```

- a. 12
- b. NameError: name 'sval' is not defined
- c. 4
- d. None of the above

QA

157



Answer to question 2 on next page.

Correct answer to previous page question:

1. What will the following program print?

- d. None of the above. The code contains no print statement!

Review Questions

3. What will the following program print?

```
num = 3
def num_fun(a=5):
    num = a
    return num

num_fun(7)
print(num)
```

- a. 3
- b. 7
- c. 5
- d. An error will occur

QA

158

Answer to question 3 on next page.

Correct answer to previous page question:

2. What will the following program print?

- b. NameError: name 'sval' is not defined.

Explanation: There is no variable within scope that has the name sval.

Review Questions

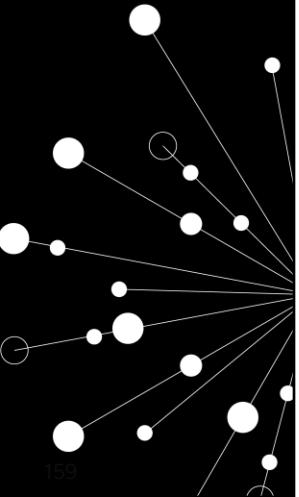
4. What will the following program print?

```
def my_function(pos=0, *pets):
    print("The chosen pet is " + pets[pos])

my_function(2,"Gnasher", "Flipper", "Lassie")
```

- a. An error will occur
- b. The chosen pet is Lassie
- c. The chosen pet is ("Gnasher", "Flipper", "Lassie")
- d. The chosen pet is Gnasher

QA



159

Correct answer to previous page question:

3. What will the following program print?

- a. 3.

Explanation: There are two variables called num but the one in the print statement's scope is initialised to 3 and is never changed.

4. What will the following program print?

- b. The chosen pet is Lassie.

Explanation: My_function is variadic and the pets parameter will soak up all the values that trail the first one (pos which will be given the 2). Consequently, it will contain a tuple of 3 strings (dog names) and since pos contains a 2 the third element in the collection ("Lassie") will be concatenated onto the end of "The chosen pet is " and the resultant string printed.

Summary

A function is a defined object

- Variables have local scope unless declared global.
- Functions can be nested within other functions.

Parameters are local variables

- Defaults may be assigned (from the right).
- *args collects arguments into a tuple.
- **kwargs collects keyword arguments into a dictionary.
- A lone * forces the caller to use named parameters.

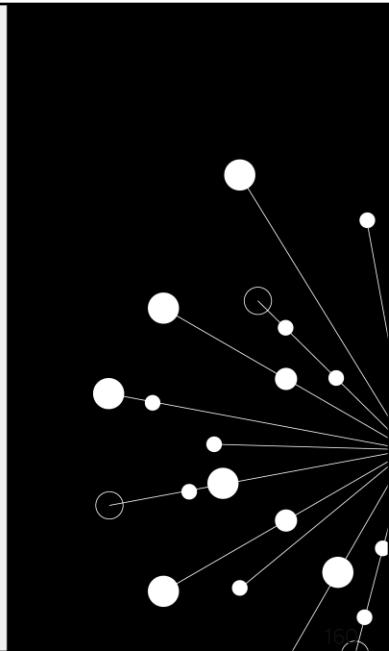
Functions can return any object

- Including lists, dictionaries, or even other functions.

Additional features:

- A docstring documents the function.
- Short, inline, anon functions can be created using lambda.

QA



160

Example using 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')
```

QA

161

This example demonstrates how to handle **arbitrary keyword arguments** using `**kwargs`.

1. The function `my_func(**user_args)` accepts any number of keyword arguments, passed as a dictionary.
2. A default dictionary `args` defines expected keys and their default values.
3. The line `diff = set(user_args.keys()) - set(args.keys())` checks for any unexpected argument names.
4. If invalid keys are found, they are printed to the error stream (`sys.stderr`) and the function returns.
5. Otherwise, `args.update(user_args)` merges the user-supplied values into the defaults.
6. Finally, the updated dictionary is printed.

The first call (`my_func(**my_dict)`) works correctly, updating town and country. The second call includes invalid keys (`twn`, `county`), so an error message is displayed instead.

Function attributes

Attribute	Meaning
<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 docstring 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

QA

162

Use function attributes like this:

```
def myfunc():
    print(myfunc.__name__)
```

At first glance, it may seem pointless to retrieve a function's name when you already know it. However, this becomes useful when working with **function references**. For example:

```
oref = myfunc
...
print(oref.__name__)
```

Here, `oref` is a reference to `myfunc`, and accessing `__name__` still returns the original function's name.

The attribute `__qualname__` (introduced at 3.3) will give the original name and where it was defined.

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')
```

QA

163

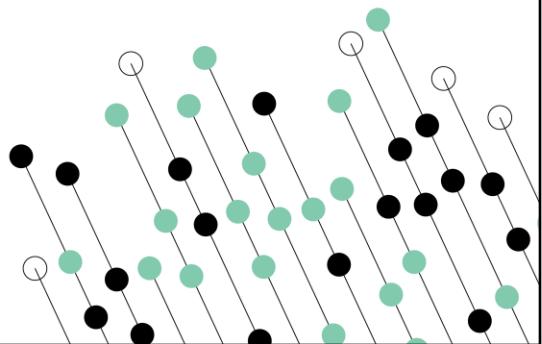
Annotations allow meta-data to be attached to parameters and return values. Python itself ignores them, as it would comments, but they are available for use by third-party libraries. Aside from the basic syntax, there are no standard data types or formats - these are left flexible for third-party libraries to exploit.

Module 9 - Advanced collections

9: Advanced Collections

10: Modules and Packages

11: OOPs

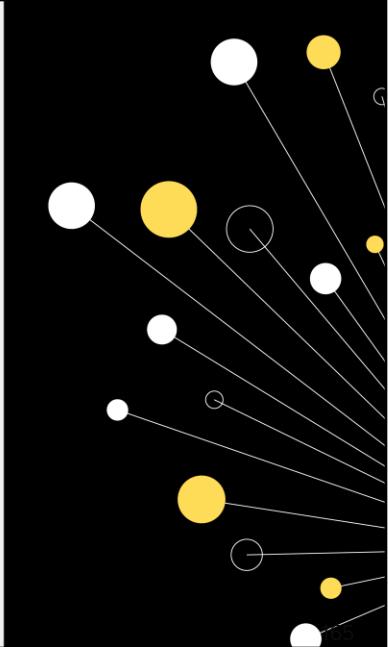


Learning objectives

By the end of this chapter, learners will be able to:

- Use the filter function
- Create and use List, set and dictionary comprehensions
- Understand what lazy lists are
- Use generators to create lazy lists
- Understand the difference between reference, shallow and deep copying when applied to collections (and other complex types)

QA



Advanced functions - filter

Syntax: `filter(os.path.isdir, glob.iglob(pattern))`

- 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

QA

166

The function passed to filter() can be either a named function or a lambda expression. In this example, we use `os.path.isdir`, which returns True for items that are directories. The first example prints each directory name, while the second uses `glob.iglob()` to expand a wildcard pattern and return an iterator of matching paths, from which a list of directory names is built.

The sequence supplied to filter() can be any iterable - such as a list, tuple, string, or even a file object. As a special case, if the function is None, filter() returns all items that evaluate to True (similar to the behaviour of grep).

Note that in Python 2, filter() returned a list, whereas in Python 3 it returns an iterator.

List comprehensions

A list comprehension returns a list

Pythonic replacement of the built-in filter() function

- It consists of 3 components:

- An **expression** which identifies a list item.
- A **loop** - typically a **for** loop.
- An optional **condition** to filter items.

```
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)]
```

```
dirs = []  
for fname in glob.iglob(pattern):  
    if os.path.isdir(fname):  
        dirs.append(fname)
```

Equivalent code using
standard syntax

QA

167

A **list comprehension** is often a more natural and concise way to express list operations. The concept originates from *set theory* and became popular through the *Haskell* programming language.

A list comprehension consists of three parts:

- **An expression** — evaluated for each item.
- **A loop** — typically a **for** statement.
- **An optional condition** — the item is included only if the condition evaluates to True.

List comprehensions fit naturally into Python's functional style, where many operations now return *iterators* rather than lists. They are considered more *Pythonic* than using the equivalent functional constructs `map()` and `filter()`.

Python 3 behaviour:

The syntax requires a single iterable expression after `in`.

```
a = [i for i in (1, 2, 3)]
```

The loop variable has its own local scope and does not affect variables of the same name outside the comprehension.

Set and dictionary comprehensions

Python 3 allows comprehensions on sets...

```
myset = {'booboo', 'yogi', 'care', 'foozie'}  
results = {do_ftp(m) for m in myset}
```

ftp to 'foozie'
ftp to 'yogi'
ftp to 'booboo'
ftp to 'care'

..and lists (of tuples):

```
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)]

... and dictionaries:

```
dsizes = {fname:size for fname, size in tsizes if size > 0}
```

{'C:/first.py': 90, 'C:/try.py': 21}

QA

168

In addition to list comprehensions, Python also supports **set** and **dictionary comprehensions**.

A **set comprehension** iterates over an existing set (or any iterable) to produce a new set. In the example, a user-defined function do_ftp() (likely using the ftplib module) performs an operation on four machines in an unspecified order. The resulting set stores whatever value the function returns for each machine.

Dictionary comprehensions are slightly more complex, as they require both a key and a value for each item. In the example, a list of two-item tuples is first created using a list comprehension, and this is then used to build a dictionary, with an additional condition (if size > 0).

This entire operation could also be written in a single statement:

```
sizes = {fn: os.path.getsize(fn) for fn in glob.iglob(pn) if os.path.getsize(fn) > 0}
```

Here, the variable names fname and pattern have been shortened to fn and pn for clarity and space.

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.

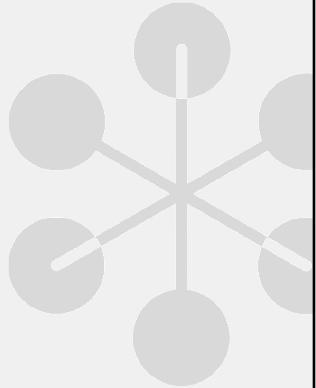
- One item at a time, as and when required.

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.



QA

169

Historically, many functions created or consumed **in-memory lists** for iteration. This worked well for small datasets, but became inefficient when lists grew large due to excessive memory overhead.

A **lazy list** solves this problem by generating items *only when needed*. If iteration stops early, the remaining items are never created, saving both time and resources. Lazy lists are implemented by providing an **iterator**, a mechanism that yields items one at a time - rather than building the entire list in memory.

Generators – Example 1

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.

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_handle):
    for row in file_handle:
        yield row ← Yield one line at a time

with open("large_file.csv", "r") as fh_in:
    count = 0
    for _ in csv_reader(fh_in): ← Iterate through Lazy List. We use
        count += 1 ← the underscore variable as not
                      interested in each line

print(f"Row count is {count}") ← Row count is 64186394
```

QA

170

This slide introduces **generator functions**, which are a special kind of function that return a *lazy iterator*. Unlike lists, generators don't store all their contents in memory at once - they produce one item at a time, only when needed. This makes them ideal for working with very large datasets or files that would otherwise exceed available memory.

In the example shown, the function `csv_reader()` takes a file handle and uses the `yield` statement to return each line one at a time. The `yield` keyword pauses the function after each line, resuming where it left off when the next value is requested. In the main code block, a large file is opened and passed to the generator. The loop iterates through each line, incrementing a counter to determine how many rows were read. The underscore variable (`_`) is used here because the loop doesn't need the actual content of the line - only to count the iterations.

Finally, when the loop completes, the total number of rows is printed. Even though the file contains over sixty-four million rows, the program handles it efficiently because only a single line is ever stored in memory at any one time. This demonstrates the key advantage of generators: they enable scalable,

memory-efficient data processing by generating values on demand rather than all at once.

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

171

This slide demonstrates how a generator can be used to produce an **infinite sequence** of numbers. The function `infinite_sequence()` starts with the variable `num` set to zero, and inside an endless `while True` loop, it uses the `yield` statement to return the current value before incrementing it by one. Because `yield` pauses the function after each value, it doesn't try to store all numbers in memory, making the generator extremely efficient.

If this generator is used inside a `for` loop, it will keep producing values forever until manually stopped, typically with a keyboard interrupt. However, you can also call `next()` on the generator to get one value at a time, which gives full control over how many results are retrieved. This example highlights how generators can create **data lazily** - producing items only when requested. This behaviour is very useful in data analysis and streaming applications where data may be continuous or too large to hold entirely in memory.

Generators – Example 3

Reading file names:

- 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.

```
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

QA

172

A **generator** allows for *lazy evaluation*, often serving as a more efficient alternative to list comprehensions. In each example shown on the slide, the function is entered only once. The `yield` statement pauses execution of the function's loop and returns an intermediate value, effectively supplying the next item in the sequence whenever it's requested.

Attempting to use a `return` statement within a loop that contains a `yield` expression will raise a `SyntaxError`; you should use `break` to exit the loop instead. Unlike list comprehensions or filters, which build a temporary list in memory, generators and lazy lists hold only the current item being processed. This makes them far more memory-efficient, especially when working with large datasets.

Note: the following imports have been omitted for clarity:

```
import os.path
import glob
```

List comprehensions as generators

A list comprehension may be used instead of `yield`.

- Sometimes - this does not support sending values.

```
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.
- Enclose the comprehension in () instead of []

```
def get_dir(path):
    pattern = os.path.join(path, '*')
    return (file
            for file in glob.iglob(pattern)
            if os.path.isdir(file))
```

QA

173

In Python 3*, we have an alternative to using `yield` - the **generator expression**, which uses a syntax similar to a list comprehension. The key difference is the use of **parentheses ()** instead of **square brackets []**.

The slide revisits an earlier `yield` example, showing its equivalent written as a generator expression. Both code fragments are functionally identical, as each returns a generator object. However, a limitation of generator expressions is that the caller cannot send values back into the generator, which is possible with `yield`.

* Introduced in Python 2.6

Copying collections - problem

Any problems with assignments?

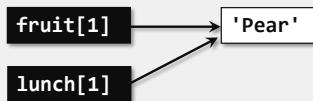
- Remember that Python objects are references:

```
fruit = ['Apple', 'Pear', 'Orange']
```



- Copy by reference:

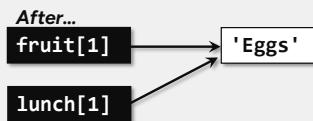
```
lunch = fruit
```



- Oops...*

```
lunch[1] = 'Eggs'  
print(fruit)
```

```
['Apple', 'Eggs', 'Orange']
```



QA

174

Since Python works with **references to objects**, copying one container (or collection) to another can sometimes lead to **unexpected behaviour**. An assignment copies only the **references**, not the underlying data. This is harmless for **immutable objects** such as strings, integers, or tuples. Still, mutable **objects** (by definition) can be changed, and those changes may affect multiple references, as the slide illustrates.

When one collection is assigned to another, only a **shallow copy** is created (the example uses lists, but this applies to all collection types).

Don't worry - there's a simple solution...

Copying collections - slice solution?

For a sequence, take a slice:

```
fruit = ['Apple', 'Pear', 'Orange']
lunch = fruit[:]
lunch[1] = 'Eggs'
print('fruit:', fruit, '\nlunch:', lunch)
```

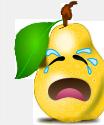
fruit: ['Apple', 'Pear', 'Orange']
lunch: ['Apple', 'Eggs', 'Orange']

We need a better solution for more complex structures.

- A slice is still a shallow copy and nested levels are still shared

```
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']]



QA

175

For a simple sequence, as shown in the previous example, the solution is straightforward - use a **slice**. The slightly unusual syntax of an **empty slice** (`[]`) means “take all the elements.” A slice always creates an **independent copy** of the original sequence.

However, this approach breaks down with **more complex data structures** — and it doesn’t take much complexity for slicing to fail. In the second example, assigning `fruit[:]` to `lunch` creates an independent copy of the outer list, but only the **reference** to the **nested list** is copied, not the nested list itself.

When an item within the **nested list** is changed, both `fruit` and `lunch` reflect that change.

This happens because slicing only performs a **shallow copy**, the outer list is duplicated, but both copies still **share the same reference** to the inner list. In this example, modifying `lunch[2][1]` to 'Eggs' also alters the nested list inside `fruit`, demonstrating how a shallow copy can lead to **unintended side effects** when mutable objects are involved.

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)
```

Beware! "copy" usually means a shallow copy

Collections are completely separate in memory

QA

176

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**.

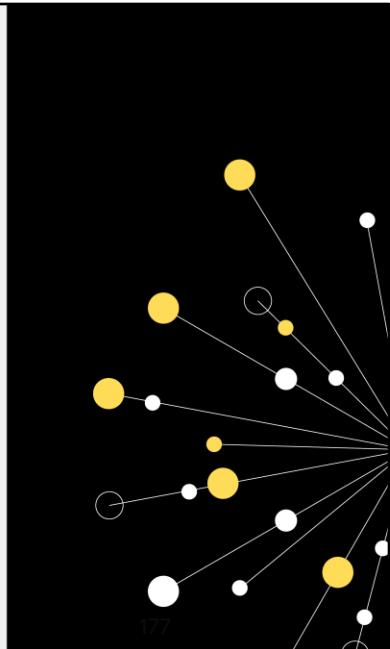
Not all objects can be copied using the **copy** module, those include other modules, class objects, functions and methods, files, and so on.

When we use the term "copy" we usually mean a shallow-copy, so be careful!

Review Questions

1. Which of the following correctly uses the built-in filter() function?
 - a. filter(lambda x: x * 2, [1, 2, 3, 4])
 - b. filter(x for x in [1, 2, 3, 4] if x % 2 == 0)
 - c. filter(lambda x: x % 2 == 0, [1, 2, 3, 4])
 - d. filter([1, 2, 3, 4], lambda x: x % 2 == 0)
2. Which comprehension creates a list of squares for numbers 1 through 5?
 - a. (x ** 2 for x in range(1, 6))
 - b. [x ** 2 for x in range(1, 6)]
 - c. {x ** 2 for x in range(1, 6)}
 - d. x ** 2 for x in range(1, 6)

QA



Answers on next page..

Review Questions

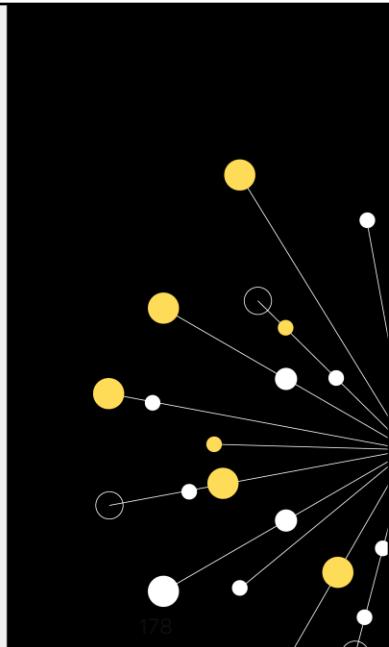
3. What is the main difference between a generator function and a normal function?

- a. A generator function must return a list
- b. A generator function can only be called once
- c. A generator function uses yield to return values one at a time
- d. A generator function automatically stores all results in memory

4. Given `b = a[:]`, what kind of copy is created?

- a. A deep copy of all nested elements
- b. A shallow copy of the outer container only
- c. A reference to the same object
- d. A reversed version of a

QA



Correct answers:

1. Which of the following correctly uses the built-in filter() function?

- c. `filter(lambda x: x % 2 == 0, [1, 2, 3, 4])`

2. Which comprehension creates a list of squares for numbers 1 through 5?

- b. `[x ** 2 for x in range(1, 6)]`

3. What is the main difference between a generator function and a normal function?

- c. A generator function uses yield to return values one at a time

4. Which statement about f-strings is true?

- b. A shallow copy of the outer container only

Summary

filter() returns items that are True

- Often used with a lambda

List comprehensions can replace filter() and map()

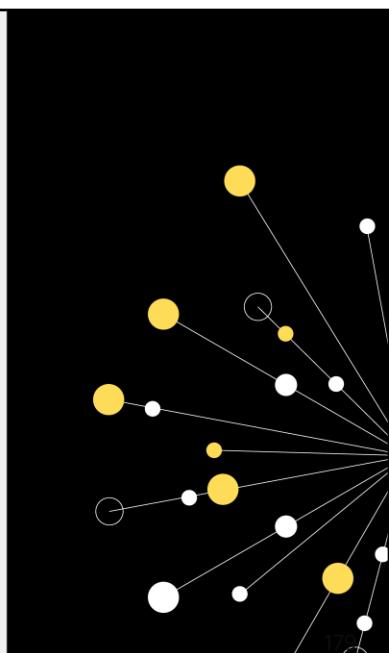
Generators yield values as needed and can replace comprehensions

- And are more memory efficient

Copying collections isn't always simple

- Sometimes a deep copy is 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')
```

Returns a Generator object. Using the
get_dir function from an earlier slide

- 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)
```

QA

180

Generator objects are created when a **generator function** is called. Each time the function is invoked, its **iteration starts from the beginning**.

If used outside a loop, the built-in **next()** function can be called to retrieve the next value produced by **yield**. Alternatively, you can call the **__next__()** method directly on the generator object, for example, **gen.__next__()** (this is what the for loop does automatically).

An optional second argument to **next()** specifies the **default value** to return when the generator sequence is exhausted.

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\Appendices
C:/MinGW\bin
C:/MinGW\dist
```

QA

181

The **send()** method can be used instead of **next()** to resume a generator. The key difference is that **send()** allows a **value to be passed back** into the generator function, where it is received as the **result of the yield expression**.

When **send()** is not used, the **yield** expression simply returns **None**.

These enhanced generator expressions were introduced in **Python 2.5** and are known as **coroutines**, as described in **PEP 342 – “Coroutines via Enhanced Generators.”**

In the example on the slide, the generator function **get_dir()** searches for subdirectories within a given path. After reporting the first two, it uses **send()** to receive a **new directory name** (passed back as the value from **yield**) and continues the search from there.

Generator delegation (Python 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.

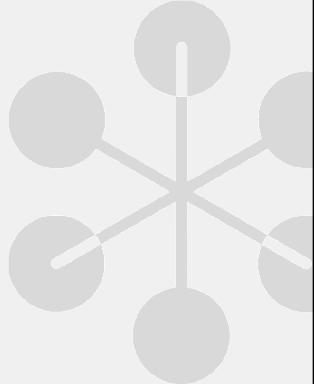
Simplistically:

- yield from iterable:

```
for file in glob.iglob(pattern):  
    yield file
```

- Can be written as:

```
yield from glob.iglob(pattern)
```



QA

182

Like any other code, **generators** can become complex and difficult to manage. The **yield from** syntax, introduced in **Python 3.3**, allows a generator to **delegate part of its operations to a sub-generator**, making it easier to **split (or decompose)** complex logic into smaller, more readable parts.

This feature is useful in **closures** and **nested generators** (generators that call other generators).

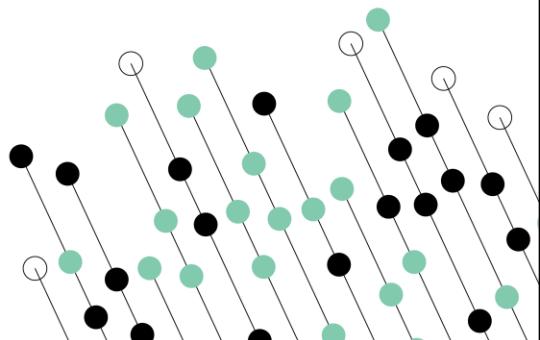
For more details and examples, see **PEP 380 – “Syntax for Delegating to a Subgenerator.”**

Module 10 – Modules and Packages

10: Regular Expressions

11: Object Oriented Programming (OOP)

12: Error Handling and Exceptions

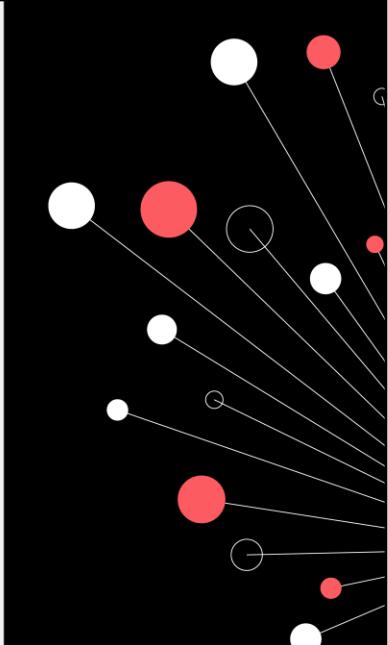


Learning objectives

By the end of this chapter, learners will be able to:

- Understand what modules and packages are
- Know why it's good to use multiple source files
- Import modules and names
- Utilise directories and namespaces as packages
- Implement code that utilises the 'main' trick
- Create module documentation by using docstrings
- Use the Python profiler to understand how code behaves

QA



We have already met modules that are bundled with Python, now we shall discuss writing our own

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 (and have a .pyc extension)

- 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.

QA

"Modules should have short, all-lowercase names." — PEP8



185

The modules we've used so far come **bundled with Python** and behave like built-in parts of the language.

While built-ins aren't stored as separate files, most other modules are — each represented by its own **independent Python file**.

When a module is loaded, Python compiles it once and saves the **bytecode** to a file in the `__pycache__` directory. From Python 3.2 onward, the compiled filename also includes the Python version, for example, `abc.cpython-32.pyc`, to prevent conflicts when multiple versions are installed.

Bytecode files (.pyc) aren't always portable across systems or Python releases.

Each contains a "**magic number**" identifying the version it was built for, so incompatible files are automatically recompiled. For this reason, packaged modules should always include the original source (.py) files.

Modules written as **C extensions** are stored as platform-specific binaries: `.pyd` on Windows or `.so` on Linux/UNIX.

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.



QA

186

A **Python module** is similar to a **source file** or **DLL** in C or C++, but it offers more flexibility and structure.

- Variables can be **local** to the module.
- Packages maintain an **independent namespace**, avoiding naming conflicts.
- In object-oriented programming, a module can define a **single class** or a group of related classes.

The term **package** refers to a **collection of modules**, typically stored together in a directory on disk (or on a network).

Dividing an application into modules follows good **structured programming principles**, but the main motivation is **code reuse** — there's no need to reinvent code that already exists and works well.

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:
 - Colon ":" for UNIX/Linux
 - Semi-colon ";" for Windows.

QA

187

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.

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.

QA

188

We've already covered the basics of **importing modules**, and it's hard to write any meaningful Python program without using import.

Here are a few additional points:

The **module name is case-sensitive** and must exactly match the file name. Even on **Windows**, this applies unless the environment variable **PYTHONCASEOK** is set.

You can assign an **alias** when importing a module, a common and convenient practice.

If a module has already been loaded, you can **reload** it using **importlib.reload()** from the **importlib** module.

This is particularly useful during development when modules are being updated programmatically.

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?

QA

189

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__.

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__)
```



QA

190

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.

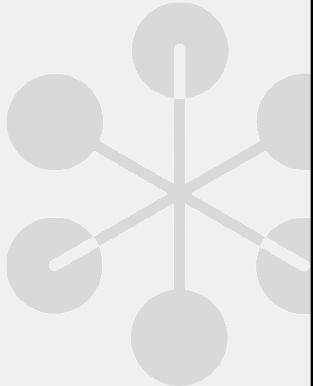
What are packages?

A package is a group of related modules organised into a directory structure.

- Use to organise code hierarchically and avoid naming conflicts
- Can contain sub-packages

Two kinds of package:

- **Regular**
 - Contains a file called `__init__.py` and other py files
- **Namespace**
 - Contains py files



QA

191

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 which may or may not contain a file called `__init__.py`.

If the directory does contain a `__init__.py` file it is known as a *Regular package*. The file is often empty or may just have 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 optionally be set is `__all__`, which gives a list of the public elements of the package.

From Python 3.3, *Namespace packages* were introduced. These don't have `__init__.py` file.

Regular Packages

A directory which contains an `__init__.py` file is a Regular Package.

The `__init__.py` file:

- Can be empty or contain initialization code
- Runs when the package is first imported
- Can define what gets imported when someone does from `package import *`
- Can contain package-level variables, functions, or classes
- Makes the package's structure explicit

```
mypackage/
    __init__.py
    module1.py
    module2.py
    subpackage/
        __init__.py
        submodule.py
```

```
# mypackage/__init__.py
from .module1 import important_function
from .module2 import MyClass
__version__ = "1.0.0"
__all__ = ['important_function', 'MyClass']
```

QA

192

Namespace Packages

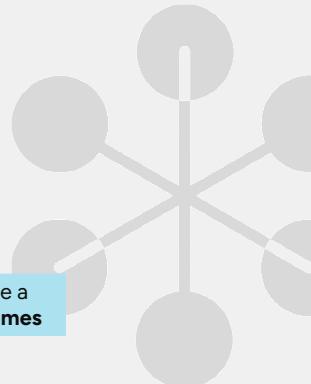
Directories without `__init__.py` are Namespace Packages.

Advantages:

- Allow multiple directories with the same package name to be combined
- Don't run any initialization code
- Are useful for distributing parts of a package separately
- Enable more flexible package structures across different locations

```
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 of code
- No `__name__` attribute for the namespace

QA

193

If your goal is simply to **group related modules logically**, traditional or “regular” packages can be somewhat **inflexible**. Although the initialization code in `__init__.py` can be powerful, it’s not always necessary or desirable.

Namespace packages, introduced in **Python 3.3**, solve this problem.

They allow a **shared namespace** to span multiple directories, meaning the same subdirectory name can appear in different parent locations and still belong to the **same package namespace**.

For full details, see **PEP 420**, which formally introduced this feature.

Regular vs Namespace Packages

Use Regular Packages (with `__init__.py`) when:

- You need **initialization code** to run when the package is imported
- You want to **control imports** (e.g. from package import *)
- You're building a **self-contained, traditional package**
- You need **compatibility with older Python versions**
- You require **package-level configuration** or constants

Use Namespace Packages (without `__init__.py`) when:

- You're creating a **plugin** or **extension** system
- Multiple teams or projects **share the same logical namespace**
- You want to **split a large package** across distributions
- You prefer a **simple structure** with no initialization code



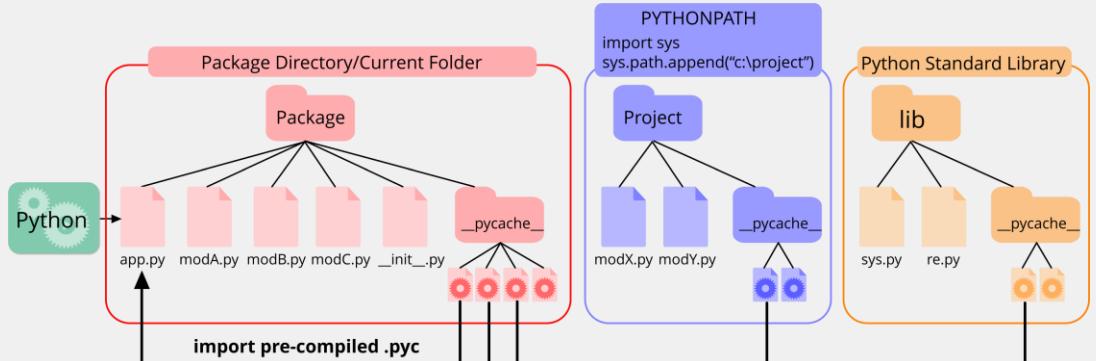
QA

194

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.

Namespace packages 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.

Finding Modules



QA

195

Python automatically stores compiled python modules in a `__pycache__` folder, which is found in the Package directory or Python install directory. When you are importing a module, including Python Standard Library modules, it is the pre-compiled modules with a suffix `.pyc` which are loaded, reducing the time to compile. If the `.py` module file has a newer timestamp, then it is re-compiled.

The Python interpreter finds the modules – as a built-in, in the current or package directory, in a list of directories defined in the `PYTHONPATH` environment variable, or an installation-dependent list of directories configured during Python install.

On Linux:

```
$ export PYTHONPATH="${PYTHONPATH}:${HOME}/python/lib:/projectX/lib"
```

On Windows: [Right Click]Computer > Properties >Advanced System Settings > Environment Variables >System Variables >New

Within a program:

```
>>> import sys  
>>> sys.path.append(r"C:\labs\projects\Proj_X")  
>>> print(sys.path)
```

Importing a package, and then a sub-package (from the current package):

```
>>> import my_package  
>>> from . import my_sub_package
```

The Namespace' trick

The `__name__` Trick:

- Every Python file has a built-in variable called `__name__`
- When a file is run directly, `__name__` is set to "`__main__`"

```
print(__name__) __main__
```

When a file is imported into another script, `__name__` is set to the module's filename instead.

```
from mainTest import importThis  
importThis()  
print(__name__) mainTest
```

The expression

- `if __name__ == "__main__":` being run as a standalone program or imported as a module.

QA

196

A script is what we call a python file that is run from a command line. The 3 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.

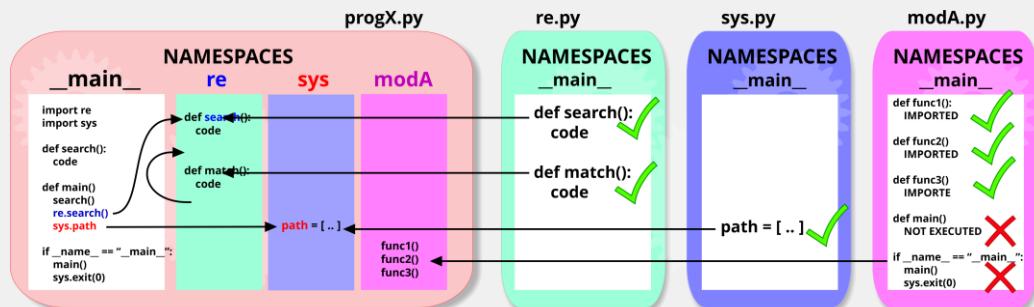
Have you noticed the `__name__ == "__main__"` test we have been using in conjunction with a `main()` function? It's called the namespace trick and allows your script to act as a module and your module to act as a script. In other words, it allows the user to execute the module directly as a script and also import the module into another script without executing the `main()` function!

It's also part of structuring our Python scripts correctly. Continue using it! Oh, and always end your script with an error code – 0 indicates zero errors and any other integer (1-255) indicates an error code! This exit code is passed back to the calling program and indicates success or failure of your script.

The 'Namespace' trick

Code Execution on Import

- Code outside functions runs immediately when the module is imported.
- This can be undesirable if the module is also meant to run as a standalone program.
- We can check the module's name — it will be "`__main__`" when run directly, and the module name when imported.



QA

197

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.

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.

Module docstring

Function docstring

```
>>> 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
```

QA

198

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...

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
```

- Analyse the output file using `pstats` shell.

```
C:\QA>python -m pstats start.prof
Welcome to the profile statistics browser.
% help
```

QA

199

A **profiler** helps you analyse how your code behaves at runtime.

It's most useful for **large or complex programs**, where performance bottlenecks are harder to spot.

The **cProfile** module comes with Python and is faster than the pure-Python **profile** module, which offers the same features.

Profiling shows:

- How many times each function was called
- How much time each took to execute
- The file and line number of each function

You can use **cProfile** within your program to profile specific sections — ideal for **unit testing** or performance tuning.

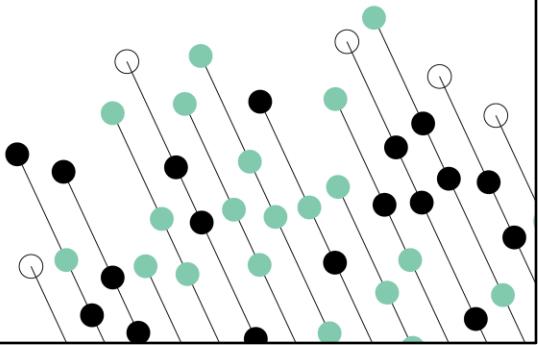
Results can be printed to the console or saved in a **binary format** for later analysis with **pstats**.

The **pstats** module can run interactively or programmatically to compare multiple profile reports. Also see the **trace** module for tracking execution paths.

Review & Lab

Answer review questions

Your instructor will guide you which lab exercises to complete

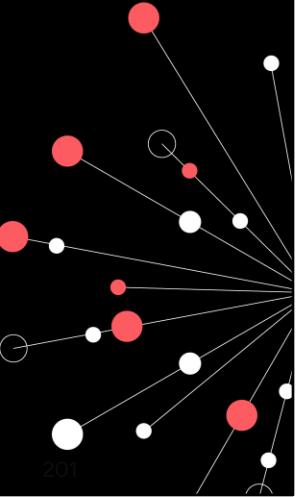


Review Questions

1. Suppose a function called `add()` is defined in a module called `arithmetic.py`. Which of the following code snippets correctly shows how to import and use the `add()` function?
Select all that apply.

- a. `import arithmetic
result = arithmetic.add(2, 3)`
- b. `from arithmetic import add
result = add(2, 3)`
- c. `from arithmetic import add
result = arithmetic.add(2, 3)`
- d. `import add from arithmetic
result = add(2, 3)`

QA



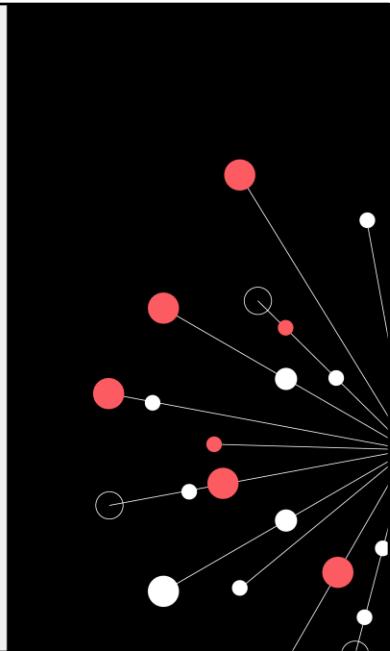
Answers on final review questions page..

Review Questions

2. What is the difference between a module and a package in Python?.

- a. A module is a folder, while a package is a single file
- b. A module is a single .py file, while a package is a collection of modules inside a folder with an `__init__.py` file
- c. Both terms mean the same
- d. A package can only contain built-in functions, while modules cannot

QA



Answers on next page

Review Questions

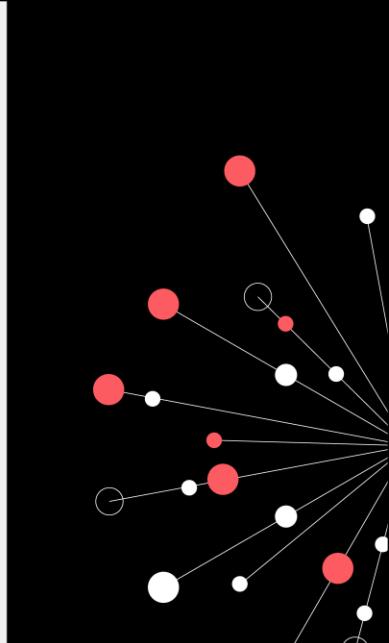
3. Where does Python look for modules when using import?

- a. Only in the current working directory
- b. In the PYTHONPATH, current directory, and standard library locations
- c. Only in the site-packages directory
- d. Nowhere, you must specify the absolute path

4. What is the purpose of the if __name__ == "__main__": statement in Python?

- a. It defines the main function of every Python program
- b. It specifies the entry point for Python packages on PyPI
- c. It ensures that a block of code only runs when the file is executed directly, not when imported as a module
- d. It prevents a module from being imported

QA



Correct answers:

1. Suppose a function called add() is defined in a module called arithmetic.py.

Which of the following code snippets correctly shows how to import and use the add() function?

a and b

```
import arithmetic
result = arithmetic.add(2, 3)

from arithmetic import add
result = add(2, 3)
```

2. What is the difference between a module and a package in Python?

- b. A module is a single .py file, while a package is a collection of modules inside a folder with an __init__.py file

This is the classic distinction.

__init__.py makes the folder behave like a package.

3. Where does Python look for modules when using import?

- b. In the PYTHONPATH, current directory, and standard library locations.

Python searches these locations in order, which you can check with sys.path.

4. What is the purpose of the if __name__ == "__main__": statement in Python?

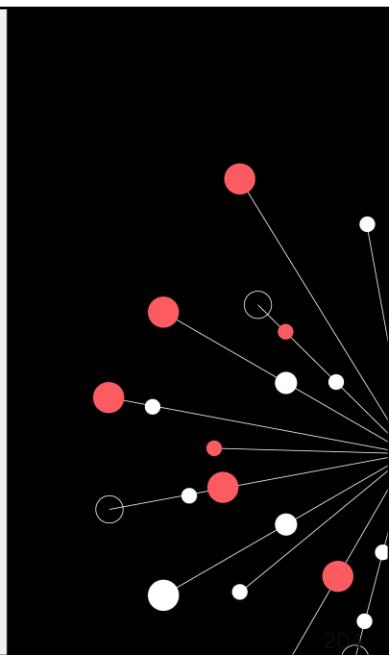
- c. It ensures that a block of code only runs when the file is executed directly, not when imported as a module

Every Python file has a built-in variable called __name__. When a file is run directly (e.g., python script.py), Python sets __name__ = "__main__". When the file is imported as a module in another script, __name__ will instead be the module's name (e.g., "script").

Summary

- **Writing a module is simple:** just save your Python code in a .py file.
- **Modules are loaded** from directories listed in `sys.path`.
- **Import modules** using the `import` statement — or import specific names into your namespace.
- **Directories become packages** when they include an `__init__.py` file.
- **Namespace packages** (no `__init__.py`) let you spread one package across multiple locations.

QA



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
```



QA

205

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.

Distributing libraries – distutils

There is a standard way of organising your files:

- Described in setup.py.

```
pydealer_pickcard/
 README.txt
 Documentation.txt
 libcard.py
 showcard/
 __init__.py
 showcard.py
 simple.py
 Bitmaps
 QA.ico
```

```
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']),
    ]
)
```

QA

206

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.

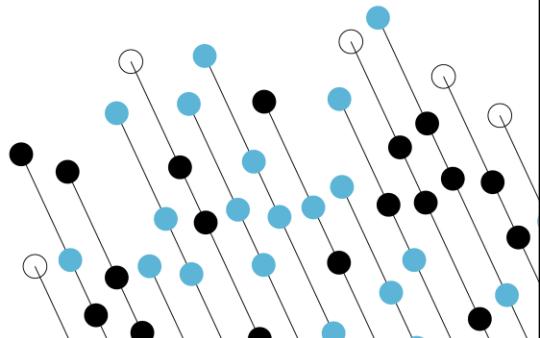
Module 11 – OO Programming

Object Oriented Programming, Classes and Objects

11: Object Oriented Programming

12: Error Handling and Exceptions

13: Testing

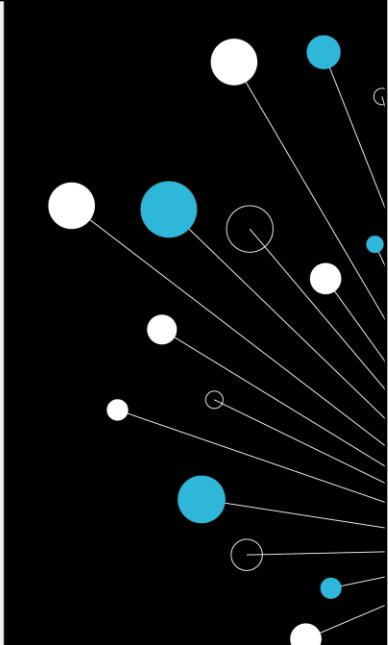


Learning objectives

By the end of this chapter, you will be able to:

- Understand key OOP principles: DRY and A PIE
- Describe and apply the four pillars of OOP:
 - Encapsulation, Inheritance, Abstraction and Polymorphism
- Define and instantiate Python classes, methods, and objects
- Use duck typing, special methods, and operator overloading
- Implement getter and setter methods using properties and decorators
- Apply inheritance to reuse and extend functionality

QA



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 key concepts in object modelling.

OOP, DRY and APIE

Object Oriented Programming

- Programming model that breaks down problems into reusable classes
- Origin in Simula (1967) and Smalltalk (1972-1980)
- Pure OO: Smalltalk, Ruby, Scala, Eiffel..
- Mostly OO: C++, C, Java, Python, VB.NET..
- Procedural + OO Extensions: JS, Perl, PHP, VB, Matlab..

Don't repeat yourself

- Why have sixteen when you can have one?

Four Pillars of OO - A PIE

- Abstraction, Polymorphism, Inheritance,
Encapsulation ..but often taught in order EIAP

QA



Attributes	Behaviour
• Colour – Black or White	Move 1 square forward
• Location – x, y position	Move 2 squares forward on 1 st move
• Value - 1	Capture one square diagonally

209

Repetition is everywhere—in life and business. People share similar traits with minor variations, and cars perform the same basic functions. Businesses operate similarly, with customers, suppliers, and employees sharing common attributes.

Board games, like chess, illustrate this concept well. Instead of creating separate variables for 16 pawns, we define a single **class** with shared attributes and generate multiple objects from it.

This follows the DRY principle—Don’t Repeat Yourself—ensuring efficiency by defining something once and reusing it.

Object-Oriented Programming (OOP) is a programming paradigm that simplifies problem-solving by breaking problems into reusable templates called *classes*. These classes enable the creation of multiple instances at runtime, promoting code reuse and efficiency. OOP is built on four foundational pillars: **Abstraction**, **Polymorphism**, **Inheritance**, and **Encapsulation**—often summarized with the acronym **A PIE**. Although, it is often easier to introduce them in the order of EIAP.

Four Pillars of OO - APIE



Encapsulation

- Hide data and methods
- Public interfaces
- Getter and Setter methods
- Data Security and Integrity

Reduce Code Smells

- Overly large classes
- Long and complicated methods
- Duplicated code
- Accessing private data and methods

QA

```
class Tank:  
    def __init__(self, country, model):  
        self.country = country  
        self.model = model  
        self._speed = 0  
        self._health = 100  
  
    def accel(self, increase):  
        self._speed += increase  
        return None  
  
    def get_health(self):  
        return self._health  
  
    def set_health(self, newhealth):  
        self._health = newhealth  
        return None
```

210

Encapsulation is a key concept in OOP, ensuring data and methods within a class are secure and controlled. For example, in a 3D tank game, each tank is a `Tank` object with attributes like health, location, and speed. To ensure fairness and security, a tank shouldn't access another tank's data directly. Instead, its data should be private and accessible only through its own methods, like `accelerate`, `decelerate`, or `get_health`.

Similarly, in a finance app, a bank account object must keep sensitive data private for security and confidentiality.

Encapsulation “encloses” data, allowing access only via controlled methods, preserving integrity. In Python, this is done by prefixing variables with a single `_` or a double underscore `__` (often referred to as a dunder). Avoid poor practices like directly accessing object data, as this weakens encapsulation and maintainability.

Finally, be cautious of **code smells**—not outright bugs, but poor coding practices like directly accessing object data, as this weakens encapsulation and maintainability.

Four Pillars of OO - APIE



Inheritance

- Write DRY code rather than WET
- Build Parent class for classes with:
 - Similar data
 - Similar functionality
- Leave differences to child classes
- Derived inherits from Base

The super() built-in function

- Better than ClassName
- Simplifies inheritance
- Promotes: Clean, readable code
- Follows MRO: Class.__mro__

QA

```
class Vehicle:  
    def __init__(self, country, model):  
        self.country = country  
        self.model = model  
        self._health = 100  
  
    def accel(self, increase):  
        self._speed += increase  
        return None
```



```
class Tank(Vehicle):  
    def __init__(self, country, model):  
        super().__init__(country, model)  
        self._health = 100  
  
    def get_health(self):  
        return self._health  
  
    def set_health(self, newhealth):  
        self._health = newhealth  
        return None
```



```
class Jeep(Vehicle):  
    def __init__(self, country, model):  
        Vehicle.__init__(self, country, model)  
        self._passengers = 0  
  
    def get_passengers(self):  
        return self._passengers  
  
    def set_passengers(self, people):  
        self._passengers = people  
        return None
```

211

In object-oriented programming, the DRY (Don't Repeat Yourself) principle prevents redundancy, while WET (Write Everything Twice) leads to inefficiency.

Inheritance helps by allowing a parent class to define shared attributes and methods, while child classes handle unique features. For example, instead of repeating code for Tank and Jeep classes, a parent class Vehicle can store common attributes like country, model, and speed, which Tank and Jeep then inherit.

In a subclass, the `__init__()` method initializes attributes and performs setup tasks but overrides the parent class's `__init__()` method. Calling `ClassName.__init__()` ensures the parent's attributes and behaviors are inherited. While effective in single inheritance, this approach can cause issues in complex multiple inheritance or when the parent class name changes.

A cleaner solution is to use the built-in `super()` function, calling `super().__init__()` to properly initialize inherited attributes and behaviours. Python's Method Resolution Order (MRO) determines the sequence in which classes are searched for methods and attributes, particularly in multiple inheritance. The MRO can be

viewed using the `__mro__` attribute or the `mro()` method.

Four Pillars of OO - APIE



Abstraction

- Focuses on Essential Features
- Hides implementation details

Abstract Class:

- Serves as blueprint for related objects

Enforce Rules:

- Ensures child class implements key methods

Promotes:

- Code consistency
- Reduces duplication
- Flexibility - unique features

QA

```
class Tank(ABC):
    def __init__(self, health, speed):
        self._speed = speed
        self._health = health

    def accel(self, increase):
        self._speed += increase
        return None

    @abstractmethod
    def fire_weapon(self):
        pass

    @abstractmethod
    def defend(self):
        pass
```

```
class HeavyTank(Tank):
    def fire_weapon(self):
        print("Firing heavy cannon")
        return None

    def defend(self):
        print("Activate shield armour")
        return None
```

```
class LightTank(Tank):
    def fire_weapon(self):
        print("Firing rapid machine gun")
        return None

    def defend(self):
        print("Using speed to dodge attack")
        return None
```

212

Abstraction is similar to inheritance but offers additional functionality. When you create an abstract class, you can define attributes and methods that child classes will inherit, much like in a typical parent class. However, abstraction also allows you to specify certain methods that all child classes are required to implement. These required methods act as a set of guidelines or rules for the child classes.

Using abstraction ensures that all child classes include essential methods, helping reduce errors and improving code consistency.

In a tank battle game, different tanks may share common traits (e.g., health, speed), but also have unique features like special weapons or defences. An abstract class ensures all tank types implement required core functionality.

Python's `abc` module enables abstract base classes using the `ABC` class and the `@abstractmethod` decorator, enforcing method implementation and design consistency.

Four Pillars of OO - APIE

Polymorphism – many forms

- Single method name can operate on different objects or classes
- Enables flexibility and reusability

Examples:

- Built in len() function on:
 - Str – returns number of characters
 - Tuple/Lists – returns number of elements
- Classes with same method:
 - Tank, Jeep and Helicopter all have fire_weapon()
 - Each implementation behaves differently (e.g., firing rate and damage)

QA



```
class Vehicle:  
    def fire_weapon(self):  
        print("Vehicle is firing a weapon!")
```



```
class Tank(Vehicle):  
    def fire_weapon(self):  
        print("Tank fires 3 high damage  
        rounds per minute")
```



```
class Jeep(Vehicle):  
    def fire_weapon(self):  
        print("Jeep fires 100 low-damage  
        rounds per minute")
```



```
class Helicopter(Vehicle):  
    def fire_weapon(self):  
        print("Helicopter fires guided  
        missiles")
```



213

Polymorphism means "many forms." In OOP, it refers to methods or functions that share the same name but can operate on different types of objects or classes.

A common example is Python's built-in len() function, which works on various collections such as strings, tuples, and lists. When used on a string, it returns the number of characters; on tuples and lists, it returns the number of elements.

In a class-based scenario, imagine having multiple classes like Tank, Jeep, and Helicopter, each with a fire_weapon method. While all these classes share the same method name, the method's behaviour differs for each object. For example, a Tank might fire three high-damage rounds per minute, while a Jeep fires 100 low-damage rounds per minute.

To streamline this, you could create a parent class called Vehicle, which defines a general fire_weapon method. The child classes—Tank, Jeep, and Helicopter—inherit this method but can override it to define their specific implementations.

Thanks to polymorphism, you can execute the fire_weapon method on any of

these classes seamlessly, ensuring flexible and reusable code. It promotes code consistency with uniform method calls and simplified program structure by using inheritance and allows for method overriding for customised behaviours in child classes.

A little Python OO

Declaring a Class in Python:

- A class is defined using the `class` keyword.
- Indentation determines class membership.
- Follow the CapWords convention for class names – per PEP 8 guidelines.

Methods in a Class

- Methods are defined as functions within the class.
- The first argument of any method is `self` (the object instance).
- The constructor is the `__init__` method, called when an object is created.
- The destructor is the `__del__` method but is rarely used and unreliable.

Class Files and Modules

- Classes are typically defined in a module with `.py` extension.



QA

214

Creating a class in Python is straightforward, and an example will be shown on the next slide. We'll explore constructors later, but note that special function names in Python begin and end with double underscores (`__`), such as the constructor (`__init__`) and destructor (`__del__`).

The **destructor** is rarely used because Python manages memory automatically using **reference counting**. Variables are references to objects, not the objects themselves. When a reference goes out of scope or is reassigned, the object isn't destroyed immediately. The destructor is only called when the **reference count** hits zero, but even then, its execution isn't guaranteed.

Because of this, avoid relying on the destructor for tasks such as:

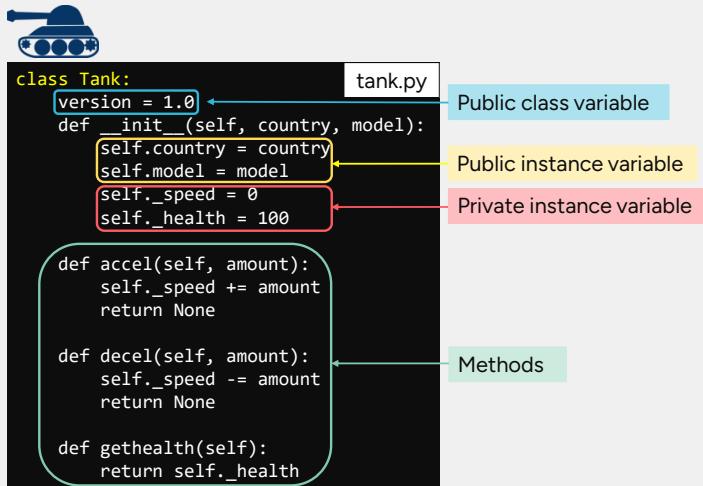
- Committing transactions
- Closing network connections
- Flushing buffers

Instead, use **exception handling** with a try-finally block (discussed in another chapter) to manage such operations safely.

Defining Classes

The class statement

- Is used to define a class object.
- Public attributes can be accessed using `ClassName.attribute` (e.g. `Tank.version`)
- Classes are typically defined in a module with same name as the class.



QA

215

Above is a simple example of a `Tank` class, defined in a module with the same name. (As per PEP 8, module names should be in lowercase.)

The class can be imported and used in two ways:

1. Using **import** and referencing the module explicitly:

```
import tank
heavy_tank = tank.Tank("german", "tiger")
```

2. Using **from ... import** for direct access to the class:

```
from tank import Tank
light_tank = Tank("american", "sherman")
```

Which is better?

Use Case 1 (`import tank`) is preferred when you want to make it clear where the class comes from, especially in larger projects with multiple modules.

Use Case 2 (`from tank import Tank`) is more concise but can clutter the namespace if multiple classes with the same name are imported from different modules.

The **class variable** version is exported by default because its name does not

start with an underscore (_). Variables with a leading underscore are treated as "private" by convention.

Using Objects

Calling a class creates a new instance object

- Invokes the constructor `__init__`.

```
from tank import Tank

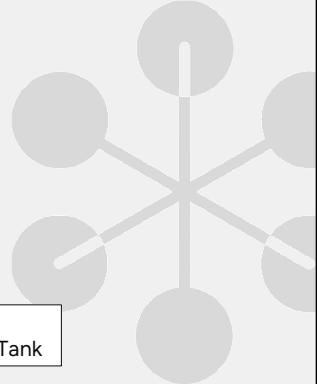
heavy_tank = Tank("German", "Tiger")
heavy_tank.accel(23)
heavy_tank.accel(9)
heavy_tank.decel(15)
print(heavy_tank.gethealth)

light_tank = Tank("British", "Scorpion")

print("Tank Version =", Tank.version)
print("Object light_tank is a class of",
      light_tank.__class__.__name__)
```

100

1.0
Object light_tank is a class of Tank



QA

216

Calling a class constructs (or instantiates) an object. This process is handled by a special function called the **constructor**, which can accept parameters just like any other function. In the Tank example, we pass “German” and “Tiger” (representing the country and model) as parameters to the constructor.

The constructor returns a reference to the created object. Once we have this reference, we can call **methods** (functions) on the object. These methods automatically receive the object reference as their first parameter (commonly referred to as `self`).

By default, object references aren't printable. However, we can define custom string representations using the `__str__` or `__repr__` methods. This is a form of special function overloading, related to operator overloading, a key feature of OOP.

To check an object's class, use the `__class__` attribute or the `type()` function, though `type()` may be less readable:

```
>>> print(type(heavy_tank))
<class 'tank.Tank'>
```

Directly checking an object's class (e.g., using `__class__`) goes against Python's **duck typing** principle. Instead of verifying the class, check if the object has the required attributes or methods. Use `hasattr()` to ensure the object can perform the needed action, following the "if it quacks like a duck" approach.

A Class is not a type!

Don't ask what type an object is:

- Only ask what the object can do - `hasattr(object, name)`
- Create a special method so the object can act like a str.

```
class Tank:  
    def __str__(self):  
        return f"Country: {self.country}, Model={self.model}"
```
- We don't care what class it belongs to, only that it can quack like a string! This is known as **Duck Typing**!

```
if hasattr(heavy_tank, "__str__"):  
    val = str(heavy_tank)
```

"If it walks like a duck,
swims like a duck,
and quacks like a duck ..."



Based on the original concepts of object orientation:

- Send/receive messages to/from an object.
- Signature-based polymorphism.

QA

217

A fundamental concept in object-oriented programming is that **messages** are sent to objects, requesting them to perform actions. Ideally, we shouldn't need to check the class of the object; we should only verify that it can carry out the requested action.

Unfortunately, this principle has been diluted in many OO languages, especially static ones. Even languages that fully support **polymorphism** often rely on checking the class rather than the object's behaviour.

Python, along with most dynamic languages, allows programmers to check if an object can perform the desired behaviour, rather than checking its class. This is known as **Duck Typing**: we don't care about the type of object, as long as it behaves as expected. If it quacks like a duck, that's good enough — even if it's a duck mimic!

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!

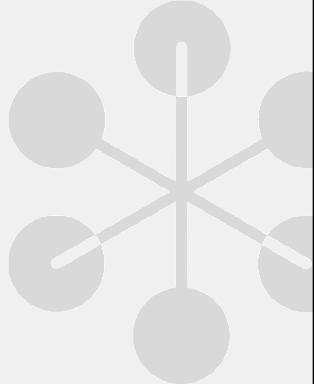
Object methods

- First argument passed to a method is the object.
 - Usually called '**self**' but can be anything.

Class methods and attributes

- Defined within the class with '**cls**' as the first argument.
- Can be called on a class or object.

QA



218

The conventions for exporting names from modules also apply to classes in Python. **Introspection**, a key concept in OOP refers to obtaining information about an object at runtime. Common introspection techniques include using the **dir()** function to display an object's attributes and methods, the **hasattr()** function to check if an object has a specific attribute, and the **help()** function to view an object's associated documentation strings (docstrings).

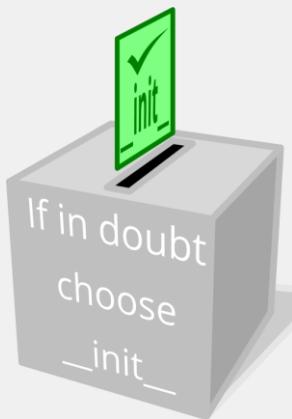
In Python, underscores in attribute and method names indicate protection or privacy levels.

- A single leading underscore (`_name`) signals internal use within a class but doesn't enforce privacy.
- A double leading underscore (`__name`) triggers name mangling, renaming it to `_ClassName__name` to prevent conflicts in subclasses.
- Double underscores around names (e.g., `__init__`, `__str__`) indicate special "dunder" methods used for operator overloading and internal functionality.

Class methods, defined with the **@classmethod** decorator, are bound to the class instead of an instance, receiving **cls** as the first argument. They are useful

for operations related to the class itself, such as factory methods, though module-level functions are often preferable. Calling a class method on an instance is generally discouraged, as it can be confusing.

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__`

QA

219

The constructor in Python can be a bit confusing because it has two possible functions: `__new__` and `__init__`. In most cases, we only need to implement `__init__`, which is automatically called after an object is created.

`__new__`, on the other hand, is a class method that is called before the object is created. It is primarily used in two scenarios: when creating immutable objects (such as strings or tuples) or when working with metaclasses. These use cases are relatively rare. If `__new__` is implemented, the `__init__` method will not be called automatically.

Special Methods

A mechanism for operator and special function overloading.

- Assists with duck-typing.

Method 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>	str.format support
<code>__hash__(self)</code>	Return a suitable key for a dictionary or set
<code>__init__(self, args)</code>	Initialise an object
<code>__len__(self)</code>	Implement the len() 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

```
class Tank:  
    def __bool__(self):  
        if self._health > 0:  
            return True  
        else:  
            return False
```

```
heavy_tank = Tank("German", "Tiger")  
  
if heavy_tank:  
    print("Tis but a scratch")
```



Tis but a scratch

QA

220

Python includes a collection of special methods that provide high-level behaviours for objects and enable their interactions with operators. Two special methods introduced earlier are `__init__` and `__str__`. The `__init__` method is implicitly called when a new object is created, while the `__str__` method is invoked when the `str()` or `print()` function is used on an object.

Other special methods allow you to customize the behaviour of standard functions when applied to objects. For example, the `len()` function can be overridden by implementing the `__len__` method. In the case of a `Tank` class, a `__len__` method could return the tank's length in meters, while a `__bool__` method could return True if the tank's health is non-zero. This enables intuitive usage in expressions such as:

```
if heavy_tank:  
    print("Tis but a scratch!")
```

It's worth noting that in Python 3, the special method for determining truth is `__bool__`, whereas in Python 2, this method is called `__nonzero__`. This distinction is important for maintaining compatibility across versions.

Operator overload special methods

All operators may be overloaded

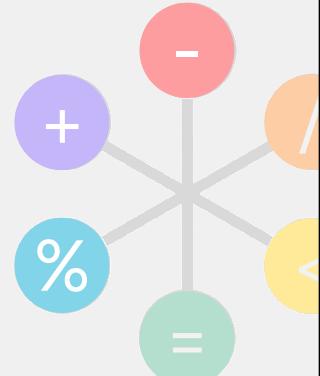
- See the online documentation for a complete list.

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>~ (logical NOT)</code>
<code>__getitem__(self, key)</code>	container element evaluation
<code>__setitem__(self, key, value)</code>	container element assignment



QA

221

This is not an exhaustive list of operator overloading special methods, as it is too extensive to cover here. For a complete reference, check the Python documentation (search for methods like `__add__`).

Compound (or augmented) assignments, such as `+=`, have their own special methods, typically prefixed with an "i". For example, the special method for `+=` is `__iadd__`. If `__iadd__` is not implemented, Python will fall back on the `__add__` method, so you don't need to duplicate code—unlike in C++.

However, methods for overloading binary operators such as `+`, `-`, `*`, `/`, `%`, `**`, `<<`, `>>`, `&`, `^`, and `|` are not automatically symmetrical. To handle cases like `1 + object`, you need to implement a corresponding method prefixed with `__r`, such as `__radd__`. For example:

- object + 1 calls `__add__`.
- 1 + object calls `__radd__`.

Many special methods for comparisons are similar and often call a shared underlying method. For example, methods like `__eq__`, `__ne__`, `__lt__`, `__gt__`, `__le__`, and `__ge__` can all be implemented as single-line calls to a generic

comparison method. This comparison method typically returns -1, 0, or +1, where 0 indicates equality.

Special methods - example

```
class Tank:
    def __init__(self, model, health, armour):
        self.model = model
        self._health = health
        self._armour = armour

    def __str__(self):
        """Overload __str__ method to return a user-friendly string."""
        return f"Tank Name: {self.model}, Health: {self._health}, Armor: {self._armour}"

    def __add__(self, other):
        """Overload __add__ method to combine two tanks into a stronger one."""
        if isinstance(other, Tank):
            new_name = f"{self.model[0:5]}-{other.model[0:4]}s"
            new_health = self._health + other._health
            new_armour = self._armour + other._armour
            return Tank(new_name, new_health, new_armour)
        raise TypeError("Can only add another Tank to a Tank!")

tank1 = Tank("Tiger", 100, 50)
tank2 = Tank("Panther", 120, 60)
print(tank1)
print(tank2)
print(tank1 + tank2)
```

QA

Tank Name: Tiger, Health: 100, Armor: 50
Tank Name: Panther, Health: 120, Armor: 60
Tank Name: **Tiger-Pants**, Health: 220, Armor: 110



222

The Tank class models a simple tank object with attributes for **name**, **health**, and **armour**. It demonstrates **operator overloading** for the `__str__` and `__add__` methods, and uses **encapsulation** by making `health` and `armour` private – managing these protected variables is covered on the next slide!

The `__str__` method provides a user-friendly string when printing a Tank object, making it easier to inspect tank details during debugging or user interaction:

Tank Name: Panther, Health: 100, Armor: 50

The `__add__` method defines how two tanks combine using the `+` operator:

- Names are merged using the first 5 characters of `tank1` and the first 4 of `tank2`.
- Health and armour values are summed to create a stronger tank.

Rather than modifying the original object, `__add__` returns a new one by calling the `Tank` constructor. If extra attributes are involved, `copy.deepcopy()` from the `copy` module can help – though it's not foolproof. A custom `__deepcopy__` method can be defined if needed.

Getter and Setter methods

Why use Getter and Setter methods

- Control access to private attributes
- Encapsulate logic for getting and setting values
- Omitting the setter method means attribute is read-only
- Maintain data integrity and enforce validation

```
class Tank:  
    def get_health(self):  
        return self._health  
  
    def set_health(self, new_health):  
        if new_health < 0:  
            self._health = 0  
        else:  
            self._health = new_health  
  
tank = Tank("German", "Panther")  
tank.set_health(90)  
print("Health of tank = {tank.get_health()}")
```

additional methods

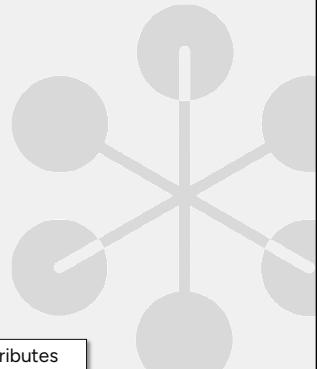
QA

Do NOT access private attributes

```
tank = Tank("German", "Panther")  
tank._health = 666  
print(f"Health of tank = {tank._health}")
```

Health of tank = 90

Health of tank = 666



223

Getter and setter methods are essential tools in object-oriented programming for encapsulating data and ensuring controlled access to private attributes. They are particularly useful when you want to enforce specific rules or validation logic before retrieving or modifying an attribute's value.

In Python, attributes prefixed with a single underscore (e.g., `_health`) or double underscore are treated as private and should not be accessed directly outside the class. Instead, getters and setters provide a structured way to manage these attributes while maintaining the integrity of the data.

In the `Tank` class example, getters (`get_health`) retrieve the values of private attributes, while setters (`set_health`) update those attributes with added validation. For instance, the `set_health` method ensures the health cannot go below zero. By using these methods, we not only protect the internal state of the object but also make the class more robust and easier to maintain.

While this example demonstrates a manual approach, Python also offers a more advanced and streamlined feature called the `@property` decorator, which allows you to define getter and setter behaviour directly with attribute-like syntax.

Properties

What is property()?

- A built-in function for defining getters, setters, deleters and docstrings.
- Provides control over attribute access with methods.

```
property(fget=None, fset=None, fdel=None, doc=None)
```

```
class Tank:  
    def get_health(self):  
        # same code as previous slide  
  
    def set_health(self, new_health):  
        # Same code as previous slide  
  
    def del_health(self):  
        self._health = None  
  
    tank_health = property(get_health, set_health, del_health, "Health of the Tank")  
  
tank = Tank("German", "Panther")  
tank.tank_health = 90  
print("Health of tank = {tank.tank_health}")
```

QA

224

The `property()` function in Python allows you to define getters, setters, and deleters for attributes in a clean and centralized way. This function links a private attribute to a set of methods for controlled access, ensuring data encapsulation. The `property()` function takes up to four arguments:

- **fget**: The getter method to retrieve the value.
- **fset**: The setter method to modify the value.
- **fdel**: The deleter method to delete the value.
- **doc**: An optional docstring describing the property.

In the `Tank` class example, `property()` is used to create a `health` property that provides controlled access to the `_health` attribute. Methods like `get_health`, `set_health`, and `del_health` handle retrieving, modifying, and deleting the `health` attribute, respectively. This approach ensures that invalid values (e.g., negative health) can be intercepted and handled properly.

While `property()` offers flexibility, it can be verbose for simple use cases. Python's `@property` decorator simplifies this process and will be discussed in the next slide.

Properties and Decorators

What is @property?

- A decorator is a function name prefixed with @
- Allows attribute-like access to methods
- Simplifies getter, setter and deleter methods

Decorators are syntactic sugar, but commonly used:

- Built-in property() is usually called using a decorator.

```
class Tank:  
    @property  
    def tank_health(self):  
        return self._health  
  
    @tank_health.setter  
    def tank_health(self, new_health):  
        if new_health < 0:  
            self._health = 0  
        else:  
            self._health = new_health
```

QA

print(tank_health)

tank_health = 100



225

The **@property** decorator in Python allows methods to be accessed like attributes, improving readability and usability. Instead of calling getters or setters explicitly, you can use attribute-style syntax:

- tank.health retrieves the value.
- tank.health = 100 updates the value.

In the Tank class example, **@property** defines a getter for health, and **@health.setter** adds a setter with validation to prevent invalid values (like negatives). This lets you control access while keeping syntax clean and intuitive.

The **@property** decorator simplifies the **property()** function. Instead of calling **property(get_health, setter=set_health)**, **@property** syntax defines the getter and setters and deleters can be added with:

- **@getter_method_name.setter**
- **@getter_method_name.deleter**

This preserves encapsulation and streamlines attribute management. Common decorators in Python include **@property**, **@classmethod**, and **@contextmanager**. Creating custom decorators is possible but beyond this course.

Why use Decorators?

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 get_health(self):
    return self._health

tank_health = property(get_health, set_health) ← Trailing property() call might
                                              be missed, or forgotten.
                                              When does it get executed?

@property
def tank_health(self): ← Method is bound to the attribute
    return self._health name and bound to @property.
```

QA



226

Decorators come from the concept of metaprogramming, where code can modify or extend itself at runtime. They aren't limited to properties—they can be applied to any function that takes another function as an argument. This makes decorators versatile and widely used in Python.

One common use case for decorators is simplifying the creation of class methods (next slide) or properties, as originally intended. They improve readability and reduce complexity, as described in PEP 318. Without decorators, using built-ins like `property()` requires manually naming new functions and tying them to attributes, which can be cumbersome and confusing. For small examples, this isn't a big issue, but in large, complex classes, managing properties without decorators can make the code harder to follow.

Decorators elegantly link functionality to attributes or methods, streamlining code organization and highlighting logical connections — almost like a Pythonic cherry on top of the cake for developers.

Class methods

What are class methods?

- Access and modify class variables shared by all instances
- Can add useful methods tied to the class
- Can provide alternative constructors to create objects

Defining class methods:

- Using `@classmethod` decorator (preferred):

```
class Tank:  
    _game_version = "1.0.0" # Class variable  
    @classmethod  
    def get_version(cls):  
        return f"Game Version: {cls._game_version}"  
  
print(Tank.get_version())
```

QA

Game Version: 1.0.0

The class name is passed implicitly

- Using built-in `classmethod()` function:

```
def get_version(cls):  
    return f"Game Version: {cls._game_version}"  
  
get_version = classmethod(get_version)  
  
print(Tank.get_version())
```

Game Version: 1.0.0

227

Class methods are rarely essential—often, a standalone function is clearer. Calling a class method on an instance is discouraged, as it can cause confusion. In Python, a function defined in a class without parameters like `self` or `cls` is unbound.

A class method is tied to the class and takes `cls` as its first parameter. It's defined using the `@classmethod` decorator. In contrast, a `@staticmethod` doesn't take `self` or `cls` and works independently—ideal for utility functions that don't need instance or class data. For example, in a `Tank` class, a static method could calculate damage based on external factors:

```
class Tank:  
    @staticmethod  
    def calculate_damage(damage, armour):  
        """Calculate effective damage based on armour."""  
        reduction = armour * 0.1 # Armour reduces damage by 10% per point  
        effective_damage = max(damage - reduction, 0) # Cannot < zero  
        return effective_damage  
  
damage = Tank.calculate_damage(damage=50, armour=5)
```

```
print(f"Effective Damage: {damage}") # Output: Effective Damage: 45.0
```

Inheritance

Understanding Inheritance

- Core OO concept.
- Build parent Base class for classes with:
 - Similar data
 - Similar functionality
- **Derived** inherits from **Base**
- Leave differences to derived child classes
- Python supports multiple inheritance - not often needed.
- 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)
```

The super() built-in function

- Simplifies inheritance and code
 - Follows MRO

```
class DerivedClassName(base_classes):  
  
    def __init__(self, arguments):  
        super().__init__(arguments)
```

QA

228

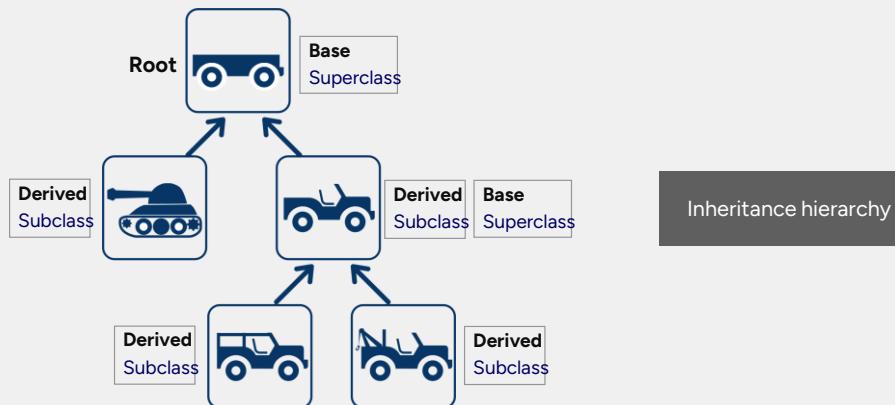
Inheritance in object-oriented programming creates an "is a" relationship between classes. A derived class inherits all attributes and methods from a base class and can add or **override** functions to provide specialised behaviour—this is known as **polymorphism**.

We use inheritance naturally—for example, when we talk about cars in general, we don't always specify sports cars or 4WDs. Likewise, in OOP, related classes can be grouped under a common base to simplify structure and reuse code.

Inheritance speeds up development by reusing tested code, improving reliability. In Python, it's common to extend existing classes. The **super()** function is a clean way to call parent methods, such as **super().__init__()** for proper setup.

Using **super()** is best practice in modern Python, especially with single or multiple inheritance. It follows the **Method Resolution Order** (MRO), ensuring correct method calls and avoiding issues like duplicate or skipped calls. It also improves maintainability by avoiding hardcoded class names.

Inheritance terminology



QA

229

Various ways exist to express OO designs, often using graphical notation. Here, we use simple arrows to show class hierarchies and base classes.

The **Chassis** class is the base, defining shared attributes like **wheels** and **engine_type**, and methods such as `start_engine()` and `stop_engine()`. It provides core functionality for all vehicle types.

The **Tank** class inherits from **Chassis**, adding attributes like armour and weapon, and may override methods like `start_engine()` for specialised behaviour.

The **Jeep** class also inherits from Chassis, focusing on features like `cargo_capacity` and `offroad_capability`, and adds methods such as `switch_to_4wd()`.

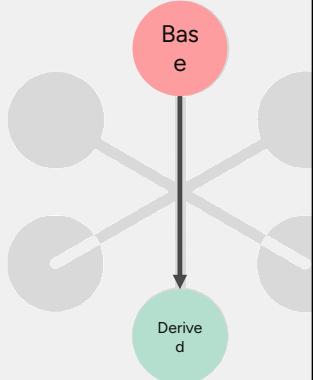
PickupJeep further specialises **Jeep**, introducing a `cargo_bed` for transporting goods and methods like `load_cargo()` and `unload_cargo()`.

This hierarchy shows how Python OOP encourages modular design and code reuse, enabling specialised classes to build on existing functionality.

Inheritance scope

Access Control and Attribute Visibility

- Attributes and methods can be public or private (no strict "protected" keyword).
- Python enforces privacy using a leading double underscore (`__attr`), which performs name mangling.
- A single underscore (`_attr`) is a convention indicating "protected" use, but not enforced.
- Base and derived classes can be in the same module.
- A derived class inherits public attributes and methods from its base class.
- Special methods (`__special__`) follow the same inheritance rules.



QA

230

When a derived-class object calls a method that does not exist in the derived class, the method from the base class is invoked (if it exists).

An interesting situation arises when a base class and its derived class have methods with the same name. If a base-class method calls one of these methods and '*self*' is a derived-class object, the derived-class method is executed instead of the base-class method. This behaviour occurs even if the method names have two leading underscores (i.e., are private). The same issue applies to public data attributes but not to private ones.

Inheritance example

The diagram illustrates the inheritance relationship between the `Vehicle` class (base/parent) and the `Tank` class (derived). The `Vehicle` class has an `__init__` method that initializes attributes `country`, `model`, and `_health`. It also has a `__str__` method that returns a string representing the vehicle's model and health. The `Tank` class inherits from `Vehicle` and adds its own attribute `_speed`. Its `__init__` method calls the parent's `__init__` method using `super().__init__(country, model)`. In the `game.p` file, a `Tank` object is created with "German" and "Tiger" as parameters, and its `__str__` method is called to print the tank's details.

```
class Vehicle:
    def __init__(self, country, model):
        self.country = country
        self.model = model
        self._health = 100

    def __str__(self):
        return f"Tank: {self.model}, Health: {self._health}"
```

```
from vehicle import Vehicle
```

```
class Tank(Vehicle):
    def __init__(self, country, model):
        super().__init__(country, model)
        self._speed = 0
```

```
from tank import Tank
```

```
mbt_tank = Tank("German", "Tiger")
```

```
print(mbt_tank)
```

QA

vehicle.p Base/parent class

tank.py

game.p

Calls the parent class constructor method

Calls parent class __str__ method

Tank: Tiger, Health: 100

231

In this example, we have a base class `Vehicle` and a derived class `Tank`. While the slide's `Tank` class only defines its own `__init__`, a real-world version would typically include methods like `start_engine`, `stop_engine`, `accelerate`, and `rotate_right`.

The `Tank` constructor uses `super()` to call the base class's `__init__`, ensuring proper initialisation of `Vehicle` attributes. Alternatively, you can call the base constructor directly (less preferred) with:

`Vehicle.__init__(self, country, model)`

The main criticism of `super()` is its complexity in multiple inheritance. Developers unfamiliar with MRO may find its behaviour unpredictable. Some prefer calling parent methods explicitly for clarity, but this bypasses MRO and can disrupt method chaining.

Using `super()` is recommended for its cleaner syntax, flexibility, and alignment with Python's design principles.

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 vehicle import Vehicle
from tank import Tank

mbt_tank = Tank("German", "Tiger")

if isinstance(mbt_tank, Tank):
    print(mbt_tank, "isa Tank")

if isinstance(mbt_tank, Vehicle):
    print(mbt_tank, "isa Vehicle")

if issubclass(Tank, Vehicle):
    print("Tank isa subclass of Vehicle")
```

All these conditions return True
(based on the inheritance example)

<__main__.Tank object at 0x00> isa Tank
<__main__.Tank object at 0x00> isa Vehicle
Tank isa subclass of Vehicle

QA

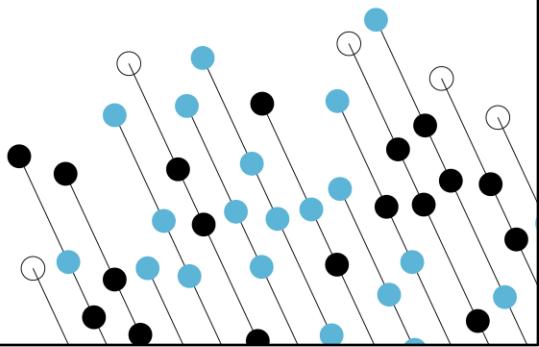
232

In both cases, *classinfo* can be a tuple of classes. These functions could be considered to discourage duck-typing.

Review & Lab

Answer review questions

Your instructor will guide you which lab exercises to complete



Review Questions

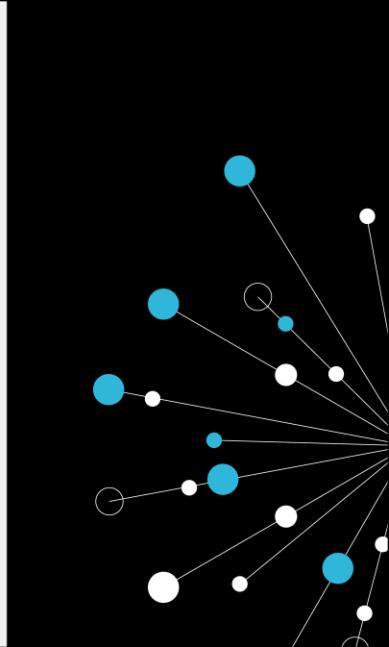
1. What are the four main principles of OOP?

- a. Inheritance, Encapsulation, Compilation, Optimization
- b. Inheritance, Encapsulation, Abstraction, Polymorphism
- c. Instantiation, Association, Delegation, Overloading
- d. Abstraction, Initialization, Protection, Serialization

2. What is the benefit of using super() in a derived class constructor?

- a. It creates a new class dynamically
- b. It bypasses the base class entirely
- c. It avoids hardcoding the parent class name and ensures proper method resolution order (MRO)
- d. It disables polymorphism

QA



Answers on next page..

Review Questions

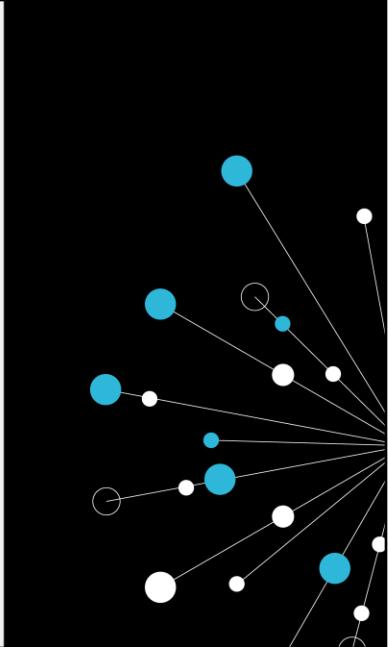
3. What is the difference between public, private, and protected attributes?

- a. Public attributes are encrypted, private ones are not, and protected attributes are temporary
- b. Public attributes can be accessed by anyone, private attributes are prefixed with __, and protected ones with _
- c. Only public attributes exist in Python
- d. Private attributes can be accessed directly, while protected and public cannot

4. What is the purpose of the __add__ method in a Python class?

- a. It adds two objects by modifying the current object directly
- b. It defines how objects should behave when added using the + operator
- c. It automatically combines all attributes of two classes
- d. It is used to overload subtraction between objects

QA



Correct answers:

1. What are the four main principles of OOP?

- b. Inheritance, Encapsulation, Abstraction, Polymorphism

2. What is the benefit of using super() in a derived class constructor?

- c. It avoids hardcoding the parent class name and ensures proper method resolution order (MRO)

3. What is the difference between public, private, and protected attributes?

- b. Public attributes can be accessed by anyone, private attributes are prefixed with __, and protected ones with _

4. What is the purpose of the __add__ method in a Python class?

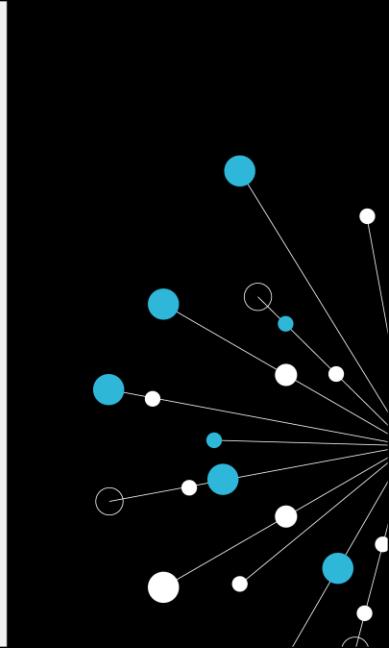
- b. It defines how objects should behave when added using the + operator

Summary

Easy as A PIE!

- OO promotes clean, reusable code
- Abstraction enforces rules and functionality
- Polymorphism enables functionality in many forms
- Inheritance reduces duplicated code
 - Classes are structured in a hierarchy for better organisation
- Encapsulation hides data and methods
 - Use Getter and Setter methods for data access
- Objects are instances of classes
- Classes can have special methods:
 - Duck Typing - `__str__`, `__int__`, `__list__`
 - For operator overloading - `__add__`, `__sub__`, `__bool__`
- Properties need decorating!

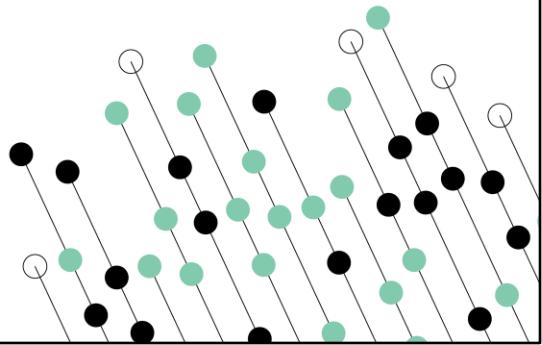
QA



Module 12 – Error Handling and Exceptions

12: Error Handling and Exceptions

13: Testing

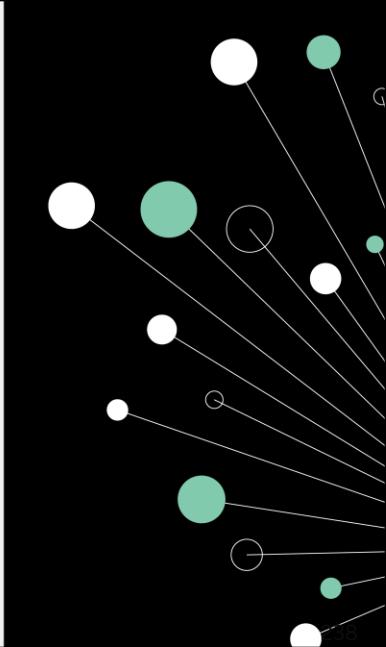


Learning objectives

By the end of this chapter, learners will be able to:

- Write messages to stderr.
- Handle errors using try / except / finally.
- Understand exception syntax and arguments.
- Describe the order of execution in exception blocks.
- Recognize the Python 3 exception hierarchy.
- Use assert for testing conditions.
- Use raise to trigger exceptions.
- Create and raise custom exceptions.
- Obtain and interpret tracebacks.

QA



Writing Errors 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 > output_file 2> error_log
```

```
import sys
if something_nasty:
    sys.stderr.write("Invalid types compared\n")
    sys.exit(1)
```

In Python 3, we can use the file parameter:

```
print("Invalid types compared", file=sys.stderr)
```

QA

See `errno` module, and `os.strerror()` for error number translation.



Most people know that `stderr` is intended for **error messages**, yet it's still common to see scripts writing errors to `stdout` instead. Python automatically routes its own error messages to `stderr`, but it cannot determine which of *your* messages are errors - that's up to you.

Using `stderr` correctly is important because many tools and systems rely on that stream to **detect, capture, or redirect errors**. For example, a common system administrator's trick is to redirect `stderr` (file descriptor 2) to a log file, then check the **file size** after the script has run. If the file is empty, no errors were produced, although a more reliable check would be to test the program's **return code**.

For more advanced error handling, use the `logging` module from the standard library. For debugging and web-based error reports, the `cgitb` module can provide detailed tracebacks.

To write to `stderr`, use: `sys.stderr.write(message)`

Standard error numbers are defined as constants in the `errno` module and can be converted to readable messages with `os.strerror()`. For example, error

number 2 corresponds to **errno.ENOENT**, meaning “*No such file or directory.*”

Managing and Customising Warnings in Python

Warnings can be generated by Python or by your own code

- Decide when a warning should be issued
- Decide where the warning should be sent
- By default, warnings are written to sys.stderr

The warnings standard module provides fine-grained control

- Create user-defined warnings with warnings.warn()
- Control how warnings are formatted and displayed (customisable functions)
- Filter warnings by type, message text, or category

Command-line control with options -Wd

- Enables warnings that are normally suppressed
- *DeprecationWarning* messages are hidden by default but can be shown with -Wd or a custom filter

QA

240

By default, Python sends **warnings** to **stderr**, but their behaviour can be controlled in several ways from within a program. In practice, you may not see many warnings, as the most common types are **ignored by default**. You can also **generate your own warnings** (using the **UserWarning** class) within a program.

Certain built-in warnings, such as **DeprecationWarning**, **PendingDeprecationWarning**, and **ImportWarning** - are hidden unless explicitly enabled using the **-Wd** command-line option or a **filter**. To activate this behaviour programmatically, use:

```
warnings.simplefilter('default')
```

Although warnings are technically part of Python's **exception system**, they are **handled differently**. Filters can be defined using **regular expressions** to match warning types or message text - examples of which are shown on the next slide.

We will also look later at how **warnings fit into the Python exception hierarchy**. Additional control is available using a **context manager**, though that is beyond the scope of this course.

Warnings – Examples

Raise a non-fatal UserWarning

```
import warnings  
  
warnings.warn('Oops')  
print('Ending...')
```

```
Warning (from warnings module):  
  File "warn.py", line 3  
    warnings.warn('Oops')  
UserWarning: Oops  
Ending...
```

Turn a warning into a fatal exception

```
import warnings  
warnings.simplefilter('default')  
warnings.filterwarnings('error', '.*')  
  
import locale  
locale.getdefaultlocale()  
print('Ending...')
```

Equivalent to -Wd options

```
File "warn2.py", line 6, in <module>  
  locale.getdefaultlocale()  
DeprecationWarning:  
'locale.getdefaultlocale' is deprecated  
and slated for removal in Python 3.15.
```

QA

Will not be executed

Raises a DeprecationWarning

241

The first example is straightforward, it demonstrates how to **issue a user-defined warning** using the `warnings` module. Notice that the program still **runs to completion**, even though a warning message is displayed.

The second example shows how to **enable and control system-generated warnings**, such as `DeprecationWarning`.

Using the following is equivalent to using the `-Wd` command-line options:

```
warnings.simplefilter('default')  
warnings.filterwarnings('error', '!*)
```

The filter pattern `'*'` matches **all warnings** and converts them into **errors**.

In the example, a call to a **deprecated function** (e.g. `locale.getdefaultlocale()`) triggers a `DeprecationWarning`, which is raised as an exception because of the filter.

Possible **filter actions** include:

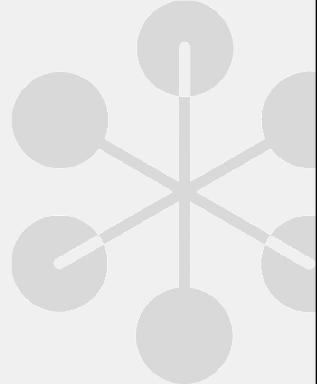
- **error** Turn matching warnings into exceptions.
- **ignore** Suppress matching warnings.
- **always** Display all matching warnings, every time they occur.
- **default** Display the first occurrence of a warning per location.
- **module** Display the first occurrence of a warning per module.

- **once** Display only the first occurrence of each warning globally.

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.

QA

242

For most programmers, **testing and handling errors** is among the least enjoyable tasks. Programs can fail in many ways, and the resulting **exception-handling code** often grows larger than the main logic itself.

Traditionally, errors were signalled by **returning a status value** from a function, but this approach has clear drawbacks:

- Leads to **nested if / else chains**, reducing readability.
- **Overloaded operators** lack spare return values for errors.
- **Return codes are easy to ignore**, allowing issues to go unnoticed.

Another technique is for an object to **track its own good/bad state**, but this often results in **clumsy, less cohesive code**. Simply **aborting the program** is a last resort, the component should detect the failure, while the caller decides how to respond.

Python's solution is to represent **exceptions as objects**. These are **self-describing**, providing detailed information about what went wrong. Exception handling in Python also uses a **separate control path**, so you only catch exceptions you care about - there's no need to manually detect and return every

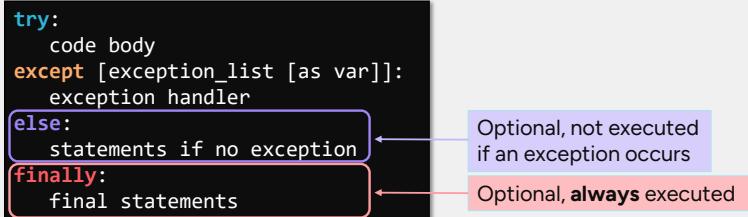
error.

Exception Handling - Syntax

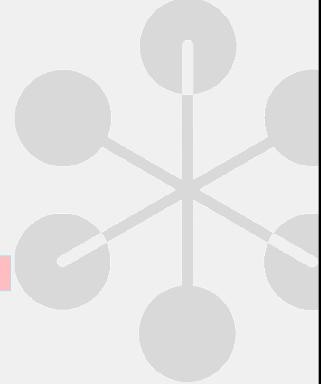
Unhandled exceptions terminate the program.

Trapping an exception:

```
try:  
    code body  
except [exception_list [as var]]:  
    exception handler  
else:  
    statements if no exception  
finally:  
    final statements
```



- Although we use terms like "try block", there is no real or implied scope within each code section.



QA

243

The **try block** contains the code to be tested. If any part of that code, often a function call, **raises an exception**, it can be **caught and handled** in the corresponding **except block**.

If a list of exceptions is specified, the handler will only execute if the raised exception **matches one in the list**; otherwise, the **stack is unwound** until an appropriate handler is found.

Within the **except block**, the raised exception can be accessed using the optional **as clause**. The **exception object** provides several useful attributes, which vary depending on its subclass.

The **else block** runs **only if no exceptions occur** in the try block, while the **finally block is always executed**, regardless of whether an exception was raised or handled (with a few specific exceptions discussed later).

Exceptions – Multiple except blocks

It is common 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 != "":  
    sys.exit(errmsg)
```

For example, TypeError would be raised if filename was not a string

Remember, exit() raises a SystemExit exception!

QA

244

Statements within a **try block**, especially function calls, may raise **different exceptions** under varying conditions. In such cases, you can handle multiple exception types using **separate except clauses**, similar to a **case-style structure**.

Alternatively, if several exceptions should trigger the **same handler**, they can be grouped together in a **tuple** within a single except statement.

Exceptions – Multiple except blocks

It is common 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 != "":  
    sys.exit(errmsg)
```

For example, TypeError would be raised if filename was not a string

Remember, exit() raises a SystemExit exception!

QA

245

Statements within a **try block**, especially function calls, may raise **different exceptions** under varying conditions. In such cases, you can handle multiple exception types using **separate except clauses**, similar to a **case-style structure**.

Alternatively, if several exceptions should trigger the **same handler**, they can be grouped together in a **tuple** within a single except statement.

Working with Exception Objects

Each exception object includes an `args` attribute:

- Stored as a tuple containing details about the error
- The number and meaning of the elements vary between exception types
- Some exceptions also provide additional named attributes, e.g., `filename` and `errno`

Accessing the Exception Object using 'as' clause:

```
import sys
try:
    f = open("foo")
except FileNotFoundError as err:
    print("Could not open", err.filename, err.args[1], file=sys.stderr) ← Errors to 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')

QA

246

When an exception is raised, Python creates an **exception object**, which like any other class instance, has attributes. New in Python 3 is the `__traceback__` attribute, which provides the full stack trace.

Different types of exceptions are actually **subclasses** of the **BaseException** class, and each subclass may define its own attributes. For example, as shown on the slide, the `FileNotFoundException` subclass includes a `filename` attribute. This was introduced in Python 3.3; in earlier versions, `IOError` was used instead. The full Python exception hierarchy is shown on a later slide.

One useful way to understand the syntax of the `except` statement is to think of it as a **special kind of function call**. It receives one parameter, the exception object and specifies which type of exception it will accept. The name following the `as` keyword becomes the local reference to that object, allowing its attributes to be accessed within the handler.

The Role of the `finally` block

The `finally` block is (almost) always executed:*

- Runs whether or not an exception occurs.
- Used for cleanup tasks, such as closing files or releasing resources.
 - The only exception: a call to `os._exit()` inside the try block bypasses finally.

The `finally` block executes before the stack is unwound.

```
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)
```

QA

Finally block
An OS error occurred



The purpose of the `finally` block is to hold **cleanup or teardown code** that must always be executed, regardless of whether an exception occurs. It is particularly useful as an alternative to a destructor, since, as we saw earlier the `__del__()` method may not always be called.

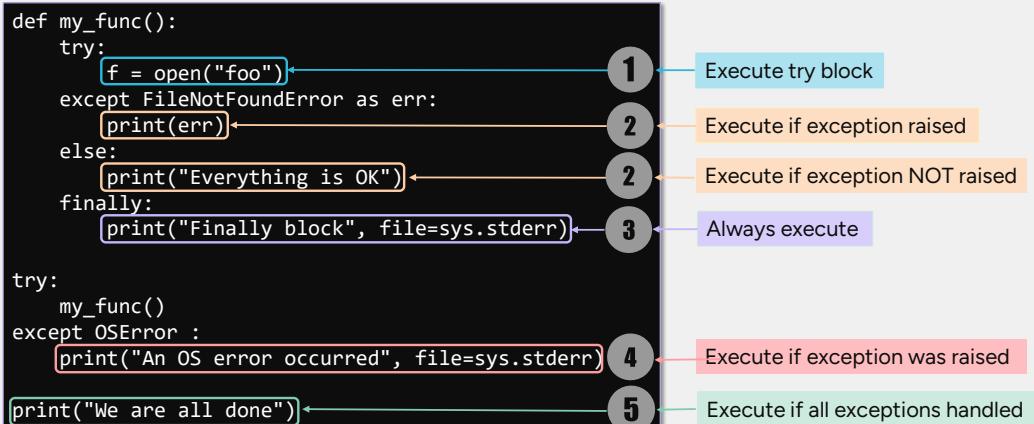
The `finally` blocks are also commonly used to **release locks** acquired in multithreaded programs. If a lock is obtained within the try block, placing the unlock operation in finally guarantees that it will always be released. However, caution is required: just because cleanup code runs does not mean that the resource is still in a consistent state. The try block may have failed partway through, leaving objects in an undefined condition. There is **no automatic rollback** - you must handle that logic yourself.

In the `finally` block, never assume that the try block succeeded; always **check handles and variables** (for example, test for `None`) before using or freeing them.

You cannot access the original exception object from within a `finally` block, and if the `finally` code itself raises an exception, the original one is **lost**.

Order of Execution

- Either the **except block** or the **else block** is executed before the **finally block**



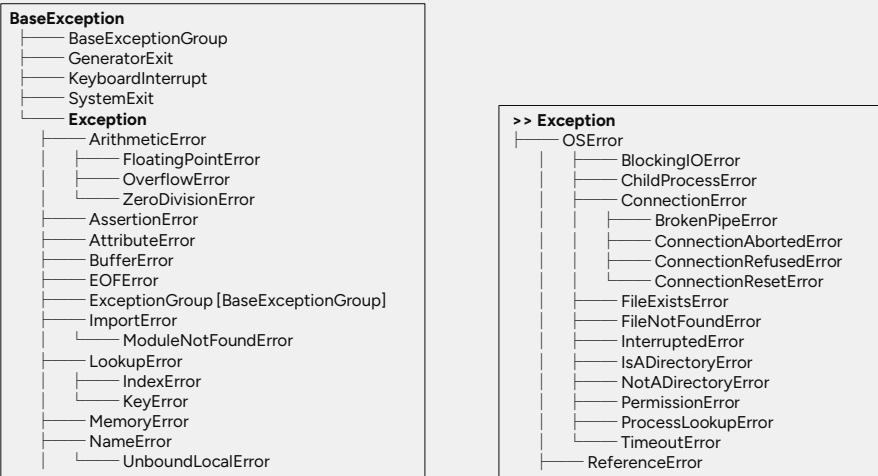
QA

248

Python first attempts to execute the code within the try block. If an exception occurs during execution, Python searches for a matching except block to handle it. If no suitable handler is found, the call stack is **unwound** until one is located, or, if none exists, the **default exception handler** is invoked and the program terminates. When a matching handler is found, it is executed, followed by the finally block (if present).

If no exception is raised, the else block is executed (if defined), and then the finally block runs.

Exception Hierarchy – part 1



QA

249

Although objects of any type can be raised as exceptions, not just those within the hierarchy shown earlier - it is best practice to use the **standard exception classes** already defined in Python's hierarchy. If a new type of exception is required, it can be created by deriving a class from an appropriate existing one. For example, database interfaces often define their own **specialised exception hierarchies** built on these standard classes. Python 3 introduced a complete reorganisation of the exception hierarchy, removing long-deprecated exceptions and unifying them under BaseException and Exception became the standard base for most user-defined exceptions.

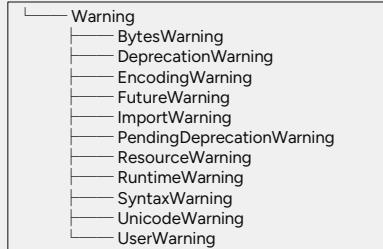
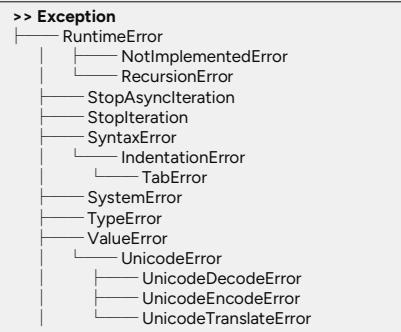
Some other changes include:

- 3.2: ResourceWarning, PendingDeprecationWarning, and ImportWarning.
- 3.3: OSError, FileExistsError, FileNotFoundError, PermissionError
- 3.5: RecursionError as a distinct subclass of RuntimeError.
- 3.6: ModuleNotFoundError.
- 3.11: ExceptionGroup and BaseExceptionGroup.

It can be tempting to catch *all* errors, but this is generally a bad idea. Some errors are **anticipated** and can be handled safely, while others are truly **unexpected**. Allowing unexpected errors to terminate the program is often safer—it stops

execution before further damage occurs. Trapping every exception can create a **false sense of security** and may **hide real bugs** that need to be fixed.

Exception Hierarchy – part 2



QA

250

Note that although warnings are technically part of the exception hierarchy, they **cannot be caught using the try...except mechanism** unless they are first **converted into exceptions** through a warning filter.

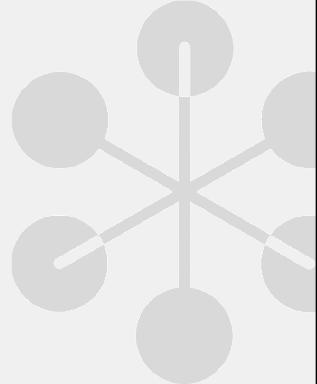
Using Assertions for Debugging

Assertions raise exceptions based on boolean tests:

- An AssertionError is raised if the expression evaluates to False
- An optional message or data value can be attached for clarity
- Syntax: `assert expression [, message]`

```
def my_func(*arguments):
    assert all(arguments), 'False argument in my_func'
    ...
my_func('Tom', '', 42)
```

AssertionError: False argument in myfunc



Assertions are best used for testing, not production:

- Remove or disable assert statements in production code:
 - Run Python with the -O (optimize) flag
 - Or set the PYTHONOPTIMIZE environment variable to 1 to disable them
 - Optimization mode sets __debug__ to False, causing all assert statements to be ignored

QA

251

The Python built-in assert statement was inspired by the C/C++ macro of the same name, though it behaves differently. It provides a simple way to perform **sanity checks** in code without writing full conditional tests. Assertions are intended mainly for **development use**, to detect conditions that should never occur. When one fails, the program stops, often the safest outcome during testing.

In production code, this behaviour is undesirable, so assertions are **normally disabled**. Running Python with the **-O (optimize)** flag ignores all assert statements:

```
python -O scriptname
```

This sets `__debug__` to False (default True), which cannot be changed directly in code.

The same effect is achieved by setting the environment variable **PYTHONOPTIMIZE** to any non-empty value (commonly 1). If unset, assertions still raise AssertionError; if set, they are ignored.

In the example, the built-in `all()` function checks that every argument evaluates to True, a typical assertion, but note that 0 counts as False, even though it may be

valid input.

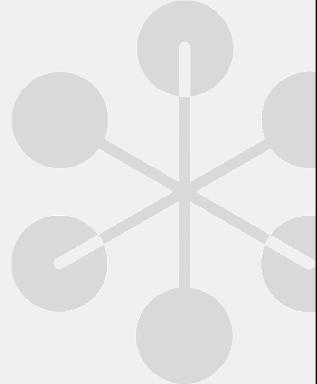
Raising Exceptions in Python

Raise a standard exception object, optionally with data:

- The raise statement creates and triggers an exception instance
 - You must **raise** an exception **object**, not a class or string
 - Include an **optional error message** or data for clarity

```
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)
```



If no exception is specified:

- Re-raises the current active exception (useful inside except blocks)
- If no active exception exists, a **TypeError** is raised

QA

252

The syntax of the `raise` statement changed in **Python 3**. In earlier versions, it was possible to raise strings or classes directly, but this was removed to make exception handling more consistent. In Python 3, you must raise a **standard exception object** (with parentheses), such as `ValueError`, optionally passing a message string to provide context.

Python 3 also introduced the ability to **chain exceptions** using the `from` keyword, which sets the `__cause__` attribute (normally unset). For example:

```
raise SomeException from AnotherException
```

This explicitly links one exception to another, preserving the original cause in the traceback.

You can also define your **own exception classes** by deriving from `Exception` or one of its subclasses - this is covered on the next slide.

Defining and Raising Custom Exceptions

Define a custom 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)
```

Oops: False argument in myfunc

QA

253

In earlier versions of Python, it was possible to **raise a string** and catch it using **except** (similar to early C++ behaviour). This was corrected in **Python 3**, where **only exception objects** can be raised - not strings.

The syntax remains straightforward, and it's common to see several **custom exception classes** defined in a single application. In many cases, these classes are empty, as shown in the example, serving only to identify a specific type of error. The full syntax of the **raise** statement (which is a **statement**, not a built-in function) includes an optional **traceback** argument:

```
raise exception with_traceback(traceback)
```

This allows an exception to be re-raised with a specific traceback, preserving information from a previous error. A more common pattern is to **chain exceptions** using the **from** keyword, for example:

```
except FileNotFoundError as err:
    raise MyError('Something went wrong') from err
```

You can also create your own traceback using the **with_traceback()** method on an exception object if you need to propagate custom error data.

Inspecting Exceptions and Tracebacks

sys.exc_info()

- Returns a tuple of (type, value, traceback) for the most recent exception
- The traceback object provides attributes such as the line number (tb_lineno) and stack trace

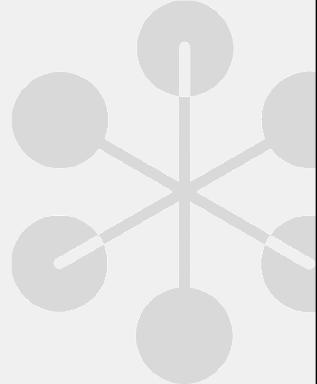
```
try:  
    open("some file name")  
except FileNotFoundError as err:  
    typ, val, tb = sys.exc_info()  
    print("Exception lineno:", tb.tb_lineno)
```

The traceback module

- Provides utilities for printing or formatting exception traces
- traceback.print_exc() is shorthand for:

```
traceback.print_exception(*sys.exc_info())
```

QA



254

Exceptions can occur deep within **nested function calls**, and one may trigger another. To inspect these errors, the **sys** and **traceback** modules provide useful tools.

`sys.exc_info()` returns a tuple containing the exception **type, value, and traceback**. While the traceback can be accessed directly, it's usually handled via the **traceback** module, which provides several utilities for examining exceptions.

The most common is `traceback.print_exc()`, which prints the stack trace in the same format as the interpreter:

```
class Exc1(Exception): pass  
  
def func1():  
    try: open("some file name")  
    except OSError as err: raise Exc1(err)  
  
    import sys, traceback  
  
    try:  
        func1()  
    except:  
        traceback.print_exc()
```

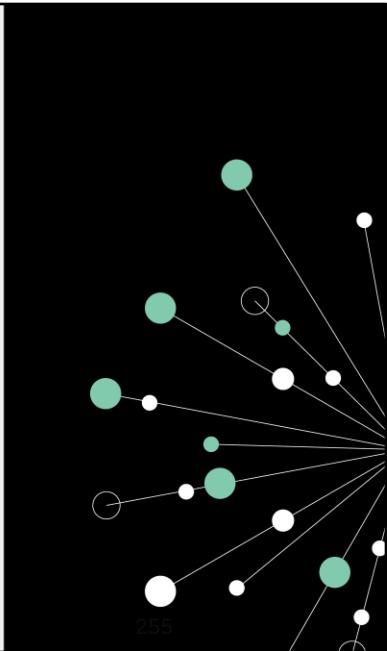
By default this prints to the console, but `traceback.format_exc()` returns the same information as a **string** for **logging or custom error handling**.

Review Questions

- 1. Which of the following statements about exceptions is true?**
 - a. All exceptions must inherit directly from BaseException
 - b. The finally block only runs if no exception occurs
 - c. Custom exceptions should normally inherit from Exception
 - d. assert statements cannot raise exceptions

- 2. What happens if an exception is raised but no matching except block is found?**
 - a. The program automatically ignores the exception
 - b. Python searches up the call stack for a handler, then stops the program if none is found
 - c. The interpreter restarts the function that caused the error
 - d. The else block is executed instead

QA



255

Answers on next page.

Review Questions

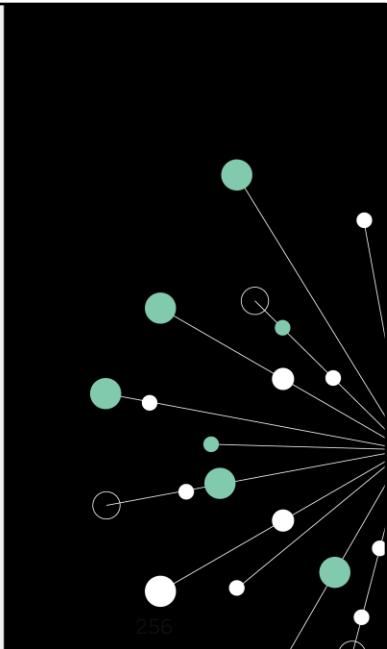
3. In Python 3, which of the following raise statements is valid?

- a. raise "Error occurred"
- b. raise ValueError, "Error occurred"
- c. raise ValueError("Error occurred")
- d. raise("Error occurred")

4. Why is the 'with' statement useful when working with files?

- a. It automatically creates a backup of the file
- b. It ensures the file is always closed, even if an exception occurs
- c. It suppresses all file-related exceptions
- d. It improves read/write performance

QA



Correct answers:

1. Which of the following statements about exceptions is true?

- c. Custom exceptions should normally inherit from Exception

2. What happens if an exception is raised but no matching except block is found?

- b. Python searches up the call stack for a handler, then stops the program if none is found

3. In Python 3, which of the following raise statements is valid?

- c. raise ValueError("Error occurred")

4. Why is the with statement useful when working with files?

- b. It ensures the file is always closed, even if an exception occurs

Summary

Write error messages to stderr

- Separates normal output from error reporting

Most modern languages support exception handling

- Enables cleaner, object-oriented error control

Exceptions are built into Python

- Many built-ins automatically raise exceptions
- Exceptions mark exceptional conditions, not always errors

Handle them properly!

- Wrap risky code with try: and handle with except:
- Optionally use else: and finally: for cleanup

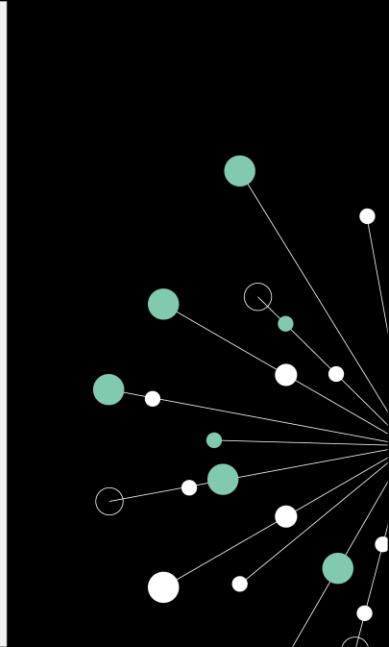
Define your own exceptions when needed

- Derive your own classes from Exception

Use assert for internal checks only

- Ideal for debugging, not for production code

QA



Context Managers

Context managers execute entry and exit code

- Implement two special methods:
 - `__enter__()`
 - `__exit__()`
- `__exit__()` handles:
 - Cleanup tasks
 - Or Exceptions automatically
- Used with the `with` statement to manage resources safely

```
with context_object as variable:  
    BLOCK
```

File objects are built-in context managers

- Automatically close files when the block exits
- No need for explicit finally blocks!

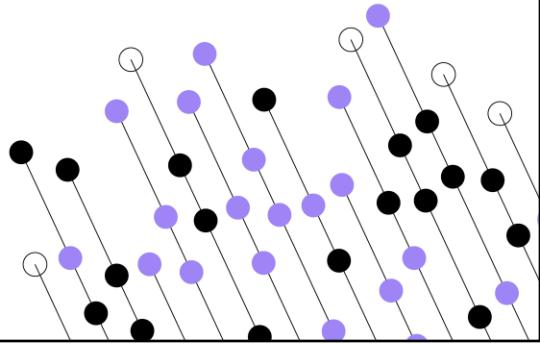
```
with open('spam.txt', 'rt') as fh_in:  
    for line in fh_in:  
        print(line, end='')  
  
print(fh_in)  
<_io.TextIOWrapper name='spam.txt'  
encoding='cp1252'>
```

QA

Context managers were introduced in **Python 2.5**. Their main advantage is that they allow objects to be **initialised and cleaned up automatically** within a defined context. Unlike destructors, which may not run immediately when an object goes out of scope, the `__exit__()` method of a context manager is **guaranteed to execute** when leaving the context - even if an **exception occurs** inside the block.

Module 13 – Testing

13: Testing



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.

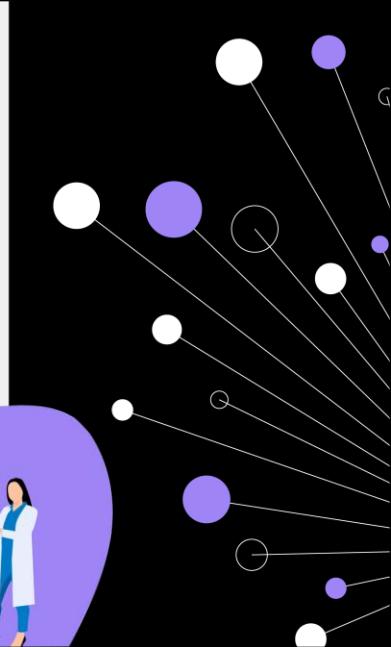
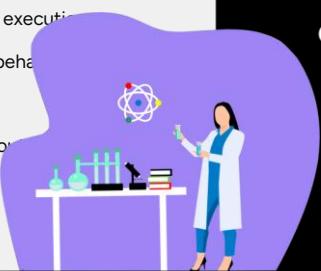
The solution is to automate your test plan using scripts. Luckily, Python already comes with a set of tools and libraries to automate your tests for your applications. We will discover how to use the **doctest** module to do DocString testing, and the **unittest** and **pytest** modules to perform automated unit testing.

Learning objectives

By the end of this chapter, learners will be able to:

- Explain the purpose of software testing and identify key testing types.
- Distinguish between manual and automated testing approaches.
- Develop a simple Python calculator app and apply basic testing principles.
- Write and test docstrings, including automated doctest execution.
- Use assertions and structured unit tests to verify code behavior.
- Run and manage test scripts with Python test runners.
- Create and execute tests using Pytest and interpret its output.

QA



What is Software Testing?

Checks whether the Application meets the expected requirements

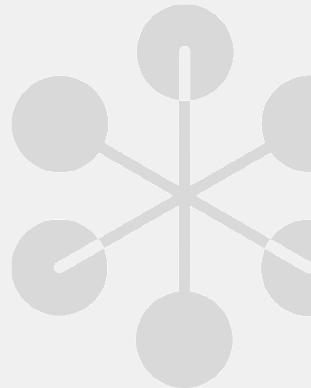
- Check for **Functional** requirements (When I click this button, an email is sent).
- Check for **Non-Functional** requirements (email is sent within 1 seconds and in a secure way).

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.



QA

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.

Why is Software Testing so Important?

SUBJECT TO
TECHNICAL
ISSUES



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>

https://www.appsierra.com/blog/software-failures-due-to-lack-of-testing?utm_source=chatgpt.com

QA

262

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 and 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.

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

263

Manual or Automated tests

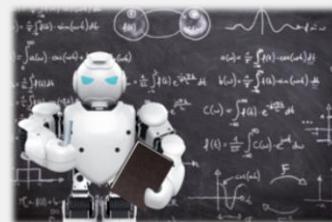
Manual testing

- 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 tests...
- **Unit testing** is often used to test code
- It is the duty of the coder to write these

QA



264

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.

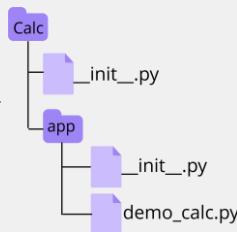
As the application scales, the list of features, and expected inputs and outputs increase. And every time the code changes you will have to manually re-apply all the tests. And possibly add more. Clearly not workable.

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.

Example – a Simple Calculator App

Demonstration

- Create a new Package called Calc
- Notice the `__init__.py` file
- Create a sub package called app
- Create a new script in the app folder called `demo_calc.py`
- Define add, multiply and divide functions
- Add and multiply functions should allow multiple parameters
- Define `main()` function to call functions



```
def add(*args):
    total = 0
    for num in args:
        total += num
    return total

def mul(*args):
    total = 1
    for num in args:
        total *= num
    return total

def div(x, z):
    return float(f"{x/z:.3f}")

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__":
    main()
```

QA

265

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.

Example – a Simple Calculator App

Manual testing on the REPL or command prompt.

```
>>> import sys  
>>> sys.path.append(r"c:\labs\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\Calculator> python demo_calc.py  
4 + 3 = 7  
4 * 3 = 12  
4/3 = 1.333  
C:\labs\Calculator>
```

QA

266

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?

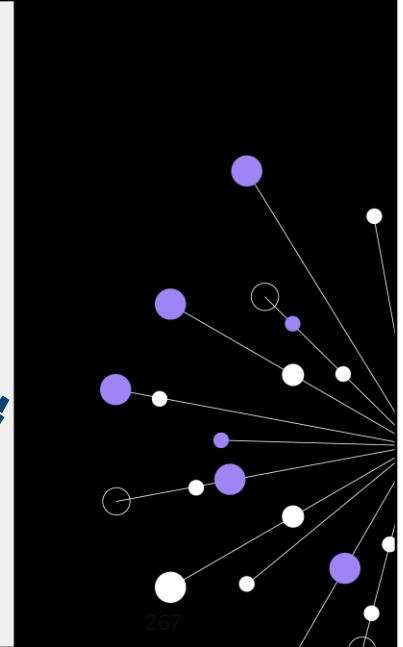
Document Strings and Testing

Document and test your code – Docstrings:

- Documents the script
- Documents the functions
- PEP 008 compliance
- Provides help()
- It can be tested!



QA



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!

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
```

```
demo_calc2.py

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}")
```

QA

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.

Docstrings – for User Documentation

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
```

```
>>> 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:\labs\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.
```

QA

269

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.

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}")
```

QA

70

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.

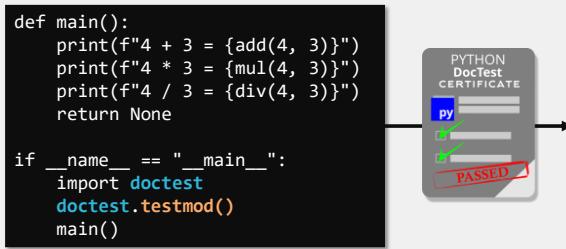
This helps the user understand how to call the function and what is expected out for given parameters, and how to manually test the function is working and matches the documentation.

But there is another benefit of USAGE examples in docstrings..

Docstrings – for Automated Testing

Docstring USAGE examples testing can be automated:

- Import the `doctest` standard library module.
- Execute `testmod()` function



```
def div(x, z):
    """ Return x divided by z to 3 decimal places
    """
    return float(f"{x/z:.3f}")

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

QA

71

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!

Docstrings – for Automated Testing

Docstring USAGE examples testing can be automated:

- Apply an error in your docstring and test again.

```
def add(*args):
    """
    >>> add(4, 3)
    8
    """
if __name__ == "__main__":
    import doctest
    doctest.testmod()
    main()
```

```
def div(x, z):
    """
    >>> div(4, 3)
    1.334
    """
if __name__ == "__main__":
    import doctest
    doctest.testmod()
    main()
```

PYTHON DocTest CERTIFICATE
PY
X
X
FAILED

```
def div(x, z):
    """Return x divided by z to 3 decimal places.
    >>> div(4, 3)
    1.334
    """
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
```

File "C:\lab\projects\14_NEU_TESTING\calc\demo_calc_plus_doc.py"
Failed example:
add(4, 3)
Expected:
8
Got:
7 Failed add test

File "C:\lab\projects\14_NEU_TESTING\calc\demo_calc_plus_doc.py"
Failed example:
div(4, 3)
Expected:
1.334
Got:
1.333 Failed div test

QA

272

In this example, we apply two errors to the DocStrings and re-run the automated testing.

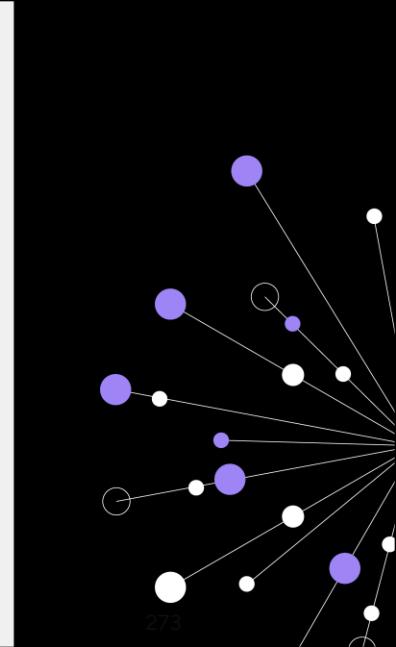
You will notice in the output that Expected and Actual values are highlighted and a summary of how many failures occurred. This can happen when code is changed and the DocStrings are not updated accordingly.

Testing Using Unittest

Unit testing:

- Unit vs integration testing
- Assertions
- Using the unittest module

QA



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**.

Unit Testing Vs Integration Testing

How to diagnose a complex application or project?

Unit testing

- First level of testing
- Tests individual units of code — functions, classes, and methods
- Often white box testing, performed by the developer



Integration testing

- Second level of testing
- Verifies that multiple components work together as a complete system
- Often black box testing, typically performed by a dedicated test team



Test step: call a function with parameters

Test Assertion: Verify that the output or behaviour is correct

QA

When we buy a new car, we trust that the manufacturer has thoroughly tested it before it leaves the showroom. Each part, such as the engine, battery, or fuel gauge, is tested individually to ensure it works correctly. This process is known as **unit testing**.

Once all the components are assembled, the manufacturer performs further tests to verify that they function properly **together** - this is **integration testing**, and can be more challenging. For example, if the car doesn't start, there could be many causes: a flat battery, a faulty starter motor, or a wiring issue. Each component may work on its own, but something in the combination has failed.

In software development, the **developer** typically performs unit tests on their code - testing individual classes, functions, and methods during the coding phase. **Integration testing** is often handled by a separate **test team**, using predefined test scripts and tools. In testing terms:

- **Test Step:** turning the key or pressing the start button.
- **Test Assertion:** observing that the engine turns over and dashboard lights come on - confirming it behaves as expected.

Modern cars often include built-in **unit tests** in their software, alerting the driver

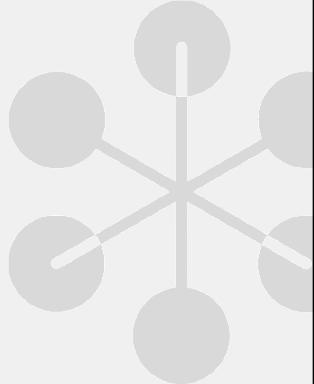
when something isn't working correctly - for instance, low fuel, low battery, or a system fault indicator.

Unit Testing – using assert

Python has a built-in function called assert() for performing unit testing.

The assert statement:

- 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.



Can be ignored when optimisation is requested at runtime by:

- PYTHONOPTIMIZE = 0
- Command line option, python -O

QA

275

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).

Unit Testing – Test Cases

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 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()
```

QA

276

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.

Unit Testing – Test Runners

tests\test_demo_calc.

The unittest module:

- Part of Python standard library
- Provides a testing framework and **test runner**
- Popular in commercial and open-source projects

unittest

- Create TestCalc class
- Inherit from `unittest.TestCase`
- Tests to be defined as **methods** in a class
- Requires some knowledge of OOP
- Uses special assertion methods

```
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()
```

QA

277

The **unittest** module has been part of the Python Standard Library since version **2.7**. It provides a structured **testing framework and test runner**, widely used across both commercial and open-source projects.

Because tests are defined as **methods within a class**, a little knowledge of object-oriented programming (OOP) is helpful. Each test class must **inherit from unittest.TestCase**, which provides a rich set of **assertion methods** for writing and validating test cases.

Converting Existing Test Cases

To convert simple tests into unittest style:

1. **Import** the unittest module
2. **Create** a new class, e.g. `TestCalc`, that inherits from `unittest.TestCase`
3. **Replace** built-in assert statements with unittest assertions
4. **Prefix** each assertion with `self` (since they're class methods)
5. **Run** the test suite using `unittest.main()` instead of a manual `main()` call

Unit Testing – Test Runners 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.

QA

Method	Equivalent
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertTrue(x)</code>	<code>bool(x) is True</code>
<code>assertFalse(x)</code>	<code>bool(x) is False</code>
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertIsInstance(a, b)</code>	
<code>assertRaises(a, *args)</code>	<code>a == Exception</code>

```
def div(x, z):
    """ 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):
    self.assertRaises(ZeroDivisionError, demo_calc.div,
                      4, 0)
    return None
```

78

The **unittest** module provides a wide range of assertions and capabilities beyond the built-in assert() statement. These include checks for whether values are **equal or not equal**, whether variables **exist or match a specific type**, whether expressions are **True or False**, and whether **exceptions are raised** as expected.

At the REPL prompt, you can explore all available assertions by importing the module and using:

```
import unittest
help(unittest)
```

This will display the full list of assertion methods and their usage.

The example on this slide demonstrates how to test whether a **ZeroDivisionError** is raised in your code.

When writing assertions, aim for tests that are:

- **Repeatable** – consistent every time they're run
- **Deterministic** – same input always produces the same result
- **Data-driven** – closely related to the input values being tested

Test Scripts – Location and Executing from Command Line

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:

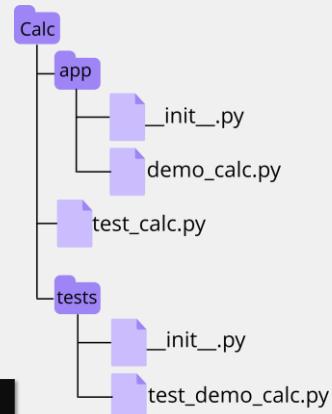
```
C:\Calc\> python -m unittest tests.test_demo_calc
```

Running a single test case:

```
C:\Calc\> python -m unittest tests.test_demo_calc.test_add
```

Running all tests:

QA C:\Calc\> python -m unittest discover



279

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 a 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.

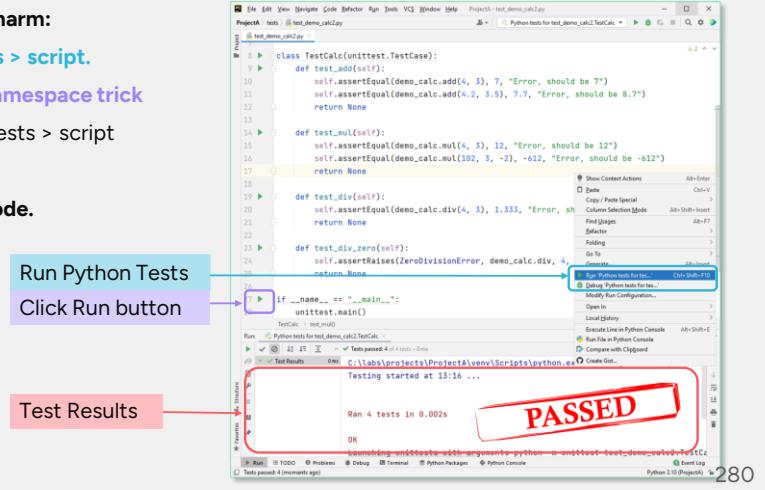
Test Scripts –Executing from Pycharm

Executing your test script in Pycharm:

- Right-click > Run Python Tests > script.
- Or Click Run Button next to Namespace trick
- Or Menu > Run > Run Python Tests > script

Test runner executes your test code.

- Test result in console.



QA

280

To execute your test script, you can either **right-click** the file and choose “**Run Python Tests**”, or use the **Run** → **Run Python Tests** menu option in your IDE. This triggers **unittest.main()**, the built-in Test Runner application, which discovers and runs all test cases within your script.

It interprets your test methods, executes them in sequence, and reports the results in the console, typically showing which tests **passed**, **failed**, or were **skipped**, along with summary statistics.

Each **test method** defined within a `unittest.TestCase` class is created and executed as a **separate test object**.

This means that before each test runs, the framework constructs a **new instance** of the test class, ensuring that tests are **independent** of one another.

Any variables, states, or resources set up during one test are not shared with others, making each test **self-contained** and **isolated** - a key principle of reliable automated testing.

280

Test Scripts –Executing from Pycharm

Edit your Test script to report failures:

- Execute the Test script again

Edit Test to Fail

Click Run button

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

Test Results

A screenshot of the PyCharm IDE interface. On the left, there's a project tree with files like test_demo_calc2.py and demo_calc2.py. The main editor window shows Python code. A red box highlights a line in the editor: `self.assertEqual(demo_calc.div(4, 3), 1.3333, "Error, should be 1.3333")`. A blue box highlights the 'Edit Test to Fail' button. Another blue box highlights the 'Click Run button' button. A red box highlights the 'Test Results' button. The bottom right shows the PyCharm run output window. It says 'Ran 4 Tests in 0.00s'. Under 'Failed', it lists 'test_demo_calc2' with '1 test(s) FAILED (failures=2)'. One failure is detailed: 'Error, should be 1.3333' vs 'Actual : 1.33333'. A red stamp 'FAILED' is overlaid on the output window. The status bar at the bottom right shows '2111 Python 3.0 (ProjectA)'.

QA

281

Edit your test script intentionally to make it fail.

In the example on the slide, the **expected result** of the division has been changed to use **four decimal places** instead of three, and the parameters have been modified so that a **ZeroDivisionError** is **not raised**.

Run the **test runner** again and observe the **console output**.

You'll see the total number of **failed tests**, along with detailed feedback for each one, including a **traceback** to the exact line that failed, and a comparison between the **expected** and **actual** results.

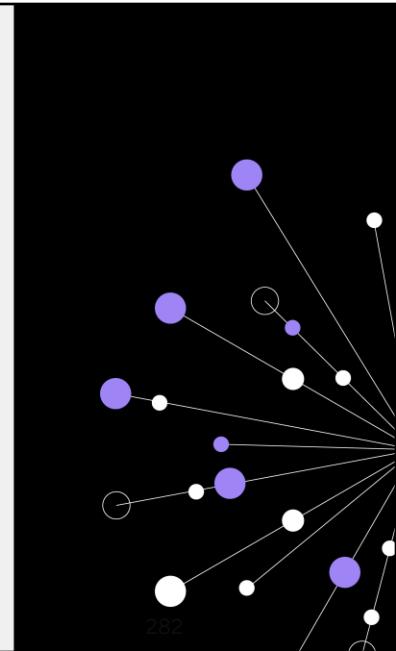
In the **test navigation window**, you can switch between viewing **all test results** or inspecting **individual test cases**, making it easy to locate and analyse specific failures.

Testing Using pytest

Unit Testing:

- Boilerplate code
- Using the pytest module
- Common data for testing using fixtures
- Varying data for testing using parametrization

QA



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.

Boilerplate Vs Elegance

Tests should be readable and consist of minimal information to understand the test case.

Unittest module:

- 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 module:

- 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

QA



In programming, **boilerplate code** refers to sections of code that are **verbose**, **repetitive**, and often reused with **minimal variation**.

You may have noticed this while using the **unittest** framework.

Before writing any actual assertions, you first need to **import unittest**, **create a new class**, **inherit from TestCase**, and then define individual **methods** for each test case using one of the many self.assert*() methods. And this process must be repeated for **every test script**! Some developers find this structure **wordy**, **less readable**, and even a bit **boilerplate-heavy**.

An alternative is to use the **pytest** module, which removes much of this overhead. With pytest, you can simply write **plain test functions** using the familiar built-in assert() statement or Boolean expressions.

If you already understand assert(), the **learning curve is very gentle** compared to mastering unittest. The only convention you need to follow is to prefix your test files and functions with the word test, for example, test_demo_calc.py.

And as a bonus, pytest provides cleaner, more readable output when you run your tests.

Installing pytest

Pytest is not part of the Python standard library – it can be installed from the:

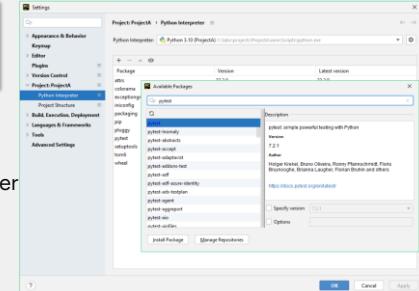
Command line:

- Can be installed in a virtual environment (venv) using pip

```
c:\ProjectA> python -m venv venv  
c:\ProjectA> .\venv\Scripts\activate  
(venv) c:\...\ProjectA> python -m pip install pytest
```

Pycharm:

- File > Settings > Project > Project Interpreter
- Select +
- Search **pytest**
- Select Install pytest



QA

284

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 up your IDE – for example, **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>

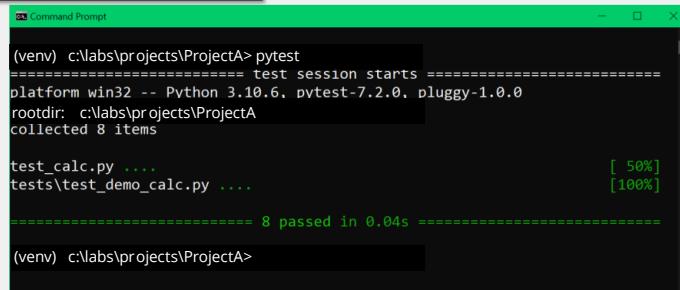
284

Pytest Output

Command line – reporting success

- Execute pytest in your project folder
- Discover test scripts
- Nicer output – **dot** (test passed), **F** (Failed), **E** (Exception)

(venv) C:\project\Calc> **pytest**



The screenshot shows a Windows Command Prompt window titled 'Command Prompt'. The command entered is '(venv) C:\labs\projects\ProjectA> pytest'. The output displays the following information:

```
(venv) c:\labs\projects\ProjectA> pytest
=====
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 =====
```

QA

285

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.

Pytest Output

Reporting failures:

```
Command Prompt
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 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"
AssertionError: Error should return 60
assert 60 == 50
+   where 60 = <function add at 0x0000012E9FEDE7A0>(10, 20, 30)
+     where <function add at 0x0000012E9FEDE7A0> = demo_calc.add

test_calc.py:13: Assertion
                    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")
AssertionError: 1.333 != 1.3333 : Error, should be 1.333

tests\test_demo_calc.py:25: Assertion
===== short test summary info =====
FAILED test_calc.py::test_add - AssertionError: Error should return 60
FAILED tests\test_demo_calc.py::testCalc::test_div - Assertion: 1.333 != 1.3333 : Error, should be 1.333
2 failed, 6 passed in 0.06s
```

First Failure

Second Failure

QA

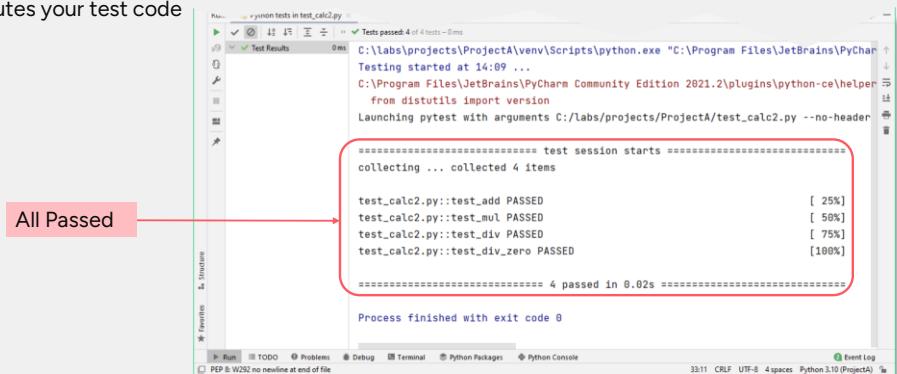
286

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.

Pytest – Reporting Success in Pycharm

Pycharm – reporting SUCCESS

- Right-click > Run Python Tests > script
- Or Menu > Run > Run Python Tests> script
- Pytest executes your test code



All Passed

```
+-----+ test session starts +-----+
collecting ... collected 4 items
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

287

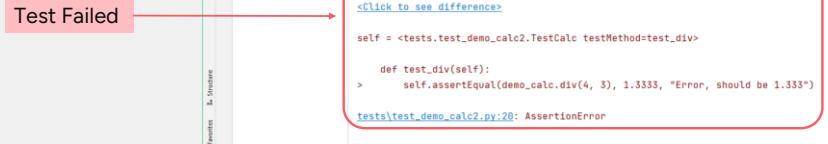
Fix your test scripts and rerun the tests in Pycharm. In this all tests passed.

Pytest – Reporting Failure in Pycharm

Pycharm – reporting FAILURE

- Edit and add an error to your tests
- Right-click > Run Python Tests > script
- Or Menu > Run > Run Python Tests > script
- Pytest executes your test code
- Detailed test report in console

Test Failed



QA

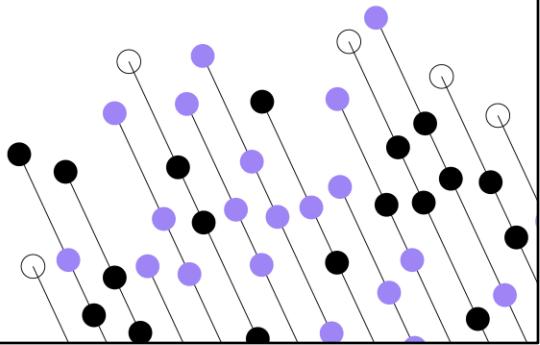
288

Edit your test scripts and add an error and rerun the tests in Pycharm. In this instance it displays that there were two failures and gives detailed breakdown of the failure, followed by an overall status report.

Review & Lab

Answer review questions

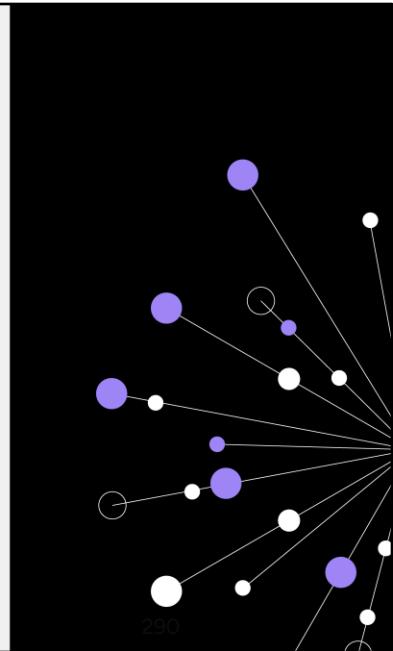
Your instructor will guide you which lab exercises to complete



Review Questions

- 1. Which of the following statements about software testing is true?**
 - a. Testing is only needed before deployment
 - b. Automated testing removes the need for human testers
 - c. Testing helps find and fix problems early in development
 - d. Testing guarantees code will never fail
- 2. Which built-in Python module provides a framework for writing and running structured test cases?**
 - a. pytest
 - b. doctest
 - c. unittest
 - d. testcase

QA

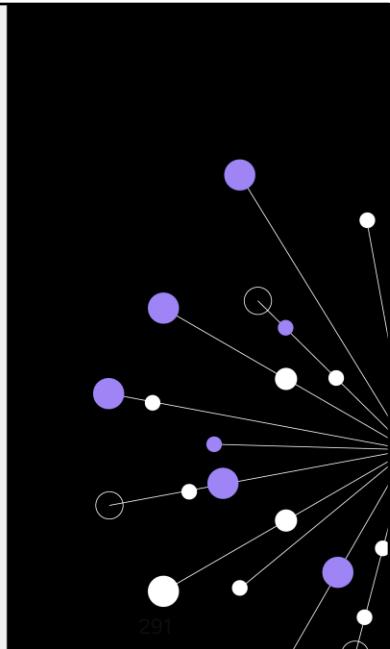


Answers on next page..

Review Questions

3. **What is the main advantage of automated testing over manual testing?**
 - a. It requires no setup or maintenance
 - b. It runs tests consistently and quickly
 - c. It eliminates all bugs
 - d. It only works for GUIs
4. **In Python, which statement is used for simple in-code checks or conditions?**
 - a. verify()
 - b. check()
 - c. confirm()
 - d. assert()

QA



Correct answers:

1. **Which of the following statements about software testing is true?**
 - c. Testing helps find and fix problems early in development
2. **Which built-in Python module provides a framework for writing and running structured test cases?**
 - c. unittest
3. **What is the main advantage of automated testing over manual testing?**
 - a. It runs tests consistently and quickly
4. **In Python, which statement is used for simple in-code checks or conditions?**
 - d. assert()

Summary

Why Software Testing Matters

- Bugs are costly to fix in production
- Good testing protects reputation, performance, and security
 - and can even save lives

Manual or Automated

- Tests can be manual or automated
- Automation is faster, consistent, and more reliable

Document Your Code

- Use docstrings and the Doctest module to verify examples.

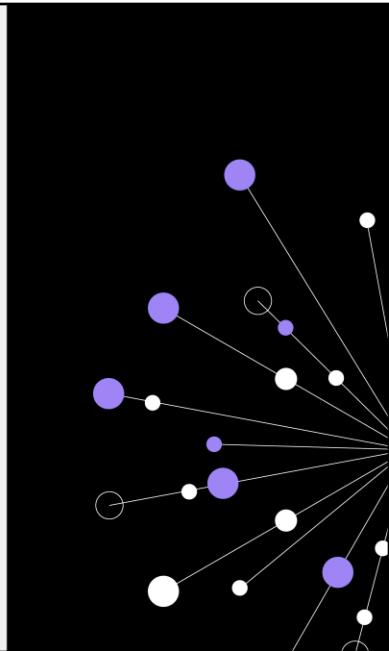
Testing Frameworks

- Write test cases and run them with tools like unittest or pytest.

Software testing isn't optional - it's essential for:

QA

- Quality, trust and maintainability



Managing test data - fixtures

Pytest fixture

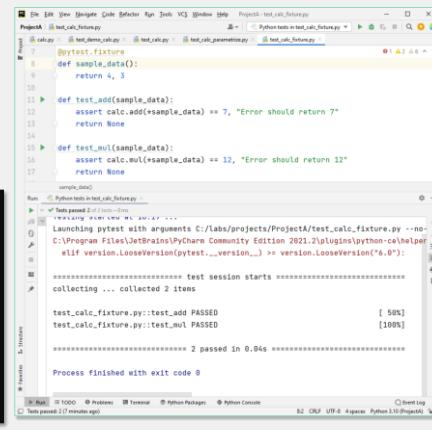
- Special function decorated with `@pytest.fixture`.
- Provides data or test double to the test function.
- Test function accepts fixture as a parameter.
- Can simplify multiple tests that use same data.
- Can be centralised in a module called `conftest.py`

```
@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
```

QA



The screenshot shows the PyCharm IDE interface with a project named 'ProjectA' containing files like `calc.py`, `test_calc.py`, `test_calc_params.py`, and `test_calc_fixture.py`. The `test_calc_fixture.py` file contains the provided code for fixtures and tests. The `Run` tool window shows the output of the pytest session, indicating that both `test_add` and `test_mul` passed. The status bar at the bottom right shows 'B2 CRU UTF-8 4 spaces Python 3.10 (ProjectA)'.

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.

Managing multiple test data - parametrized

Pytest – combining tests

- Solution to different inputs and outputs.
- Generalises the code.
- Scales better.
- Uses `@pytest.mark.parametrize()`

```
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
```

QA

The screenshot shows the PyCharm IDE interface. On the left, the project structure and file tree are visible. The main editor window contains the Python code for testing a calculator's addition function using parameterization. On the right, the 'Run' tool window displays the test results. It shows three test cases being collected and then executed. All three tests pass, with success rates of 33%, 66%, and 100% respectively. The total duration of the test session is 0.03s. The status bar at the bottom indicates that 3 items passed.

Fixtures help reduce code duplication by providing reusable setup or common parameters for multiple tests.

However, fixtures are best suited to **shared inputs or outputs**, not when your test data varies slightly between cases.

For tests with **different input–output combinations**, pytest allows you to **combine and generalise** test data using the `@pytest.mark.parametrize()` decorator.

The first argument is a **comma-separated string** of parameter names representing the inputs and expected results.

The second argument is a **list or tuple** containing the specific data sets to be tested.

Note: Avoid over-complicating your parameterization - tests should remain **clear, simple, and easy to read**.