



**UMBC**  
**TRAINING CENTERS**

Advanced Python

TCPRG0004-2021-09-08



# Table of Contents

1. Python Refresher . . . . .	1
Objectives . . . . .	1
Symbols . . . . .	2
Variables . . . . .	3
Basic Python data types . . . . .	4
Sequence Types . . . . .	5
Mapping Types . . . . .	6
Program Structure . . . . .	7
Files and Console I/O . . . . .	9
Conditionals . . . . .	10
Loops . . . . .	11
Builtins . . . . .	12
Exercises . . . . .	13
2. Packaging and Distributed Python Code . . . . .	15
Objectives . . . . .	15
Multiple Python versions with <code>pyenv</code> . . . . .	16
Virtual environments . . . . .	18
The virtual environment management swamp . . . . .	21
Importing modules . . . . .	22
Creating Packages . . . . .	24
Packaging for distribution . . . . .	26
Creating wheels . . . . .	34
Exercises . . . . .	38
3. Pythonic Programming . . . . .	39
Objectives . . . . .	39
The Zen of Python . . . . .	40
Tuples . . . . .	41
Iterable unpacking . . . . .	42
Unpacking function arguments . . . . .	43
The <code>sorted()</code> function . . . . .	45
Custom sort keys . . . . .	46
Lambda functions . . . . .	49
Comprehensions . . . . .	50
Iterables . . . . .	52
Generator Functions . . . . .	53
Generator Expressions . . . . .	54
String formatting . . . . .	55
f-strings . . . . .	57
<code>unittest</code> and Unit Testing . . . . .	59
Exercises . . . . .	72
4. Intermediate Classes . . . . .	73

Objectives .....	73
What is a class? .....	74
Defining Classes .....	75
Object Instances .....	76
Instance attributes .....	77
Instance Methods .....	79
Constructors .....	80
Getters and Setters .....	81
Properties .....	82
Class Data .....	85
Class Methods .....	86
Inheritance .....	87
Multiple Inheritance .....	91
Abstract base classes .....	93
Special Methods .....	95
Static Methods .....	100
Exercises .....	101
5. Idiomatic Data Handling .....	103
Objectives .....	103
Deep vs shallow copying .....	104
Using <code>OrderedDict</code> .....	106
Default dictionary values .....	106
Counting with Counter .....	107
The array module .....	108
Enumerated Types .....	110
Dataclasses .....	114
Named Tuples .....	116
Printing data structures .....	117
Zipped archives .....	118
Tar Archives .....	119
Archives the easy way .....	121
Serializing Data .....	122
Exercises .....	124
6. Generators and Coroutines .....	127
Objectives .....	127
Extended iterable unpacking .....	128
What exactly is an iterable? .....	129
Generators .....	130
Generator functions .....	131
Generator Expressions .....	134
Coroutines .....	136
Generator classes .....	140
Exercises .....	141

7. Type Hinting .....	143
Objectives .....	143
Type Hinting .....	144
Static Analysis Tools .....	145
Type Hinting Functions .....	146
<code>__annotations__</code> .....	148
Forward References .....	149
<code>typing</code> Module .....	151
Type Hinting of Parameters .....	152
Generator type hinting .....	154
Creating Types .....	155
Variance .....	157
Covariance .....	158
Invariance .....	159
Contravariant .....	162
Specifying variance .....	163
<code>Union</code> Types .....	164
<code>Optional</code> Types .....	165
<code>functools.singledispatch</code> .....	167
<code>multimethod</code> .....	168
Stub Type Hinting .....	169
Exercises .....	171
8. Functional Tools .....	173
Objectives .....	173
Higher-order functions .....	174
Lambda functions .....	175
The operator module .....	176
The <code>functools</code> module .....	177
<code>map()</code> .....	178
<code>reduce()</code> .....	179
Partial functions .....	180
Single dispatch .....	181
The <code>itertools</code> module .....	183
Infinite iterators .....	184
Extended iteration .....	186
Grouping .....	188
Combinatoric generators .....	190
Exercises .....	191
9. Metaprogramming .....	193
Objectives .....	193
Metaprogramming .....	194
<code>globals()</code> and <code>locals()</code> .....	195
The <code>inspect</code> module .....	196

Working with attributes .....	198
Adding instance methods.....	199
Decorators .....	201
Trivial Decorator .....	206
Decorator functions.....	207
Decorator Classes.....	209
Decorator parameters.....	211
Creating classes at runtime.....	213
Monkey Patching.....	215
Is a Metaclass needed?.....	217
About metaclasses .....	218
Mechanics of a metaclass .....	219
Singleton with a metaclass .....	222
Descriptors.....	224
Exercises .....	227
<b>10. Parallelism and Concurrency.....</b>	<b>231</b>
Objectives .....	231
Concurrency vs. Parallelism .....	232
Simple <code>fork/exec</code> Model .....	234
Multiprocessing with <code>multiprocessing</code> .....	236
Using Pipes .....	238
Using Pools .....	239
Multithreading with <code>threading</code> .....	243
Creating a thread class.....	248
The GIL .....	250
Debugging threaded Programs.....	258
Exercises.....	260
<b>11. Coroutines and <code>asyncio</code> .....</b>	<b>261</b>
Objectives .....	261
Asynchronous programming with <code>asyncio</code> .....	262
Key asynchronous vocabulary.....	263
Defining coroutines .....	264
The Event Loop .....	266
Futures .....	270
Tasks .....	273
Callbacks .....	280
How to run coroutines.....	281
Exercises .....	282
<b>12. Distributed Computing .....</b>	<b>283</b>
Objectives .....	283
Parallel and distributed computing.....	284
Asynchronous programming.....	284
Multiprogramming and parallelism .....	284

Distributed Computing .....	285
SCOOP .....	286
Message Passing Interface Using <code>mpi4py</code> .....	291
Exercises .....	296



# Chapter 1. Python Refresher

## Objectives

- Refresh the following, basic (intro-level), Python concepts:
- Understand symbols and variables
- Use the basic Python data types
- Differentiate and use effectively the various sequence types
- Differentiate and use effectively the various mapping types
- Structure scripts and programs in a standard format
- Use File and console I/O
- Understand and use the various Python conditional and looping constructs

# Symbols

Every **symbol** in Python is an alphanumeric string that does not start with a digit, and is not one of Python's keywords.

- Python has a number of **keywords**, special words that may not be used as variables, functions, or parameters, and may neither be assigned to nor overridden.

*Table 1. Python Keywords*

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

Symbols that start and end with two underscores are considered special to Python, often referred to as “dunders”.

- For example, both `__name__` and `__len__` are dunders.
  - ▶ Python may refer to dunder variables or invoke dunder methods in certain cases.

# Variables

**Variables** are declared by assigning to them.

- Python does not require explicit type specifiers, but sets the type implicitly by examining the value that was assigned.
  - ▶ Thus, assigning a literal integer to a variable creates a variable of type `int`, while assigning quoted text to a variable creates a variable of type `str`.
  - ▶ Once a variable is assigned a value, it will cause an error if the variable is used with an operator or function that is inappropriate for the type.

A variable cannot be used before it is assigned a value.

- Variables must be assigned **some** value.
- A value of `None` may be assigned if no particular value is needed.

The Python convention for variable names is often referred to as **snake case**, that is `all_lower_case_words_with_underscores`.

- This can be seen in the example below.

`snake_case.py`

```
1#!/usr/bin/env python
2
3 name = 'Fred Flintstone'
4 count = 0
5 name = "Fred Flintstone"
6 colors = ['red', 'purple', 'green']
7 name = '''Fred
8 Flintstone'''
```

# Basic Python data types

Python has many data types.

There are builtin constructors to convert from one type to another by passing the type to be converted.

- If the source type cannot be converted to the target type, a `TypeError` is raised.

## Numeric types

- `bool`
- `int`
- `float`
- `complex`

## Sequence types

- `str`
- `bytes`
- `list`
- `tuple`

## Mapping types

- `dict`
- `set`
- `frozenset`

# Sequence Types

Strings are text (arrays of Unicode characters)

```
s = 'text'
```

A 'bytes' object is an arrays of bytes

```
b = b'text'
```

A **list** is a sequence of values

```
my_list = []
```

A **tuple** is a *readonly* sequences of values (used as records)

```
my_tuple = 'Mary', 'Poppins', 'London'
```

Python supports four types of sequences—**str**, **bytes**, **list**, and **tuple**.

- All sequences share a common set of operations, methods, and builtin functions.
- Each type also has operations specific to that type.

All sequences support slicing, which returns a subsequence using the `sequence[start:stop:step]` syntax.

`sequences.py`

```
1#!/usr/bin/env python
2
3 colors = ['red', 'green', 'blue', 'purple', 'pink', 'yellow', 'black']
4 c1 = colors[0]      # 'red'
5 c2 = colors[1:4]    # 'green', 'blue', 'purple'
6 c3 = colors[-1]     # 'black'
7 c4 = colors[:3]     # 'red', 'green', 'blue'
8 c5 = colors[3:]     # 'purple', 'pink', 'yellow', 'black'
```

**NOTE**

The starting value of a slice is inclusive, while the ending value is exclusive.

# Mapping Types

Python also supports mapping types - dictionaries and sets.

A dictionary (`dict`) is a set of values indexed by an immutable keyword.

- Dictionaries are used for many tasks, including mapping one set of values to another, and counting occurrences of values.
- Prior to version 3.6, dictionaries were unordered, but beginning with 3.6, dictionaries preserve the order in which items are added.

Dictionary keys must be hashable which means it needs the following two special methods:

- A `__hash__` method whose return value never changes during its lifetime.
- An `__eq__` method to compare it with other objects with the added requirement that hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

- Most of Python's immutable built-in objects are hashable.
  - ▶ Immutable containers are only hashable if their elements are hashable.
- Objects which are instances of user-defined classes are hashable by default.
  - ▶ They all compare unequal (except with themselves), and their hash value is derived from their `id()`.

A set is an unique collection of values.

- There are two types
  - ▶ A `set` is dynamic (mutable)
  - ▶ A `frozenset` is fixed (immutable), like a `tuple`.

# Program Structure

In Python, modules must be imported before their contents may be accessed.

Variables, functions, and classes must be declared before they can be used. Thus most scripts are ordered in this way:

1. import statements
2. global variables
3. main function
4. functions
5. call to main function

You may want to make a template for your Python scripts such as the one shown on the following page.

- Most editors and IDEs support templates or code snippets.

*script\_template.py*

```
1 #!/usr/bin/env python
2 """
3 This is the doc string for the module/script.
4 """
5 # standard library imports
6 import sys
7
8 # third party module imports
9
10 # local module imports
11
12 # constants (AKA global variables -- keep these to a minimum)
13
14
15 def main(args):
16     """
17     This is the docstring for the main() function
18
19     :param args: Command line arguments.
20     :return: None
21     """
22     function1()
23
24
25 def function1():
26     """
27     This is the docstring for function1().
28
29     :return: None
30     """
31     pass
32
33
34 if __name__ == '__main__':
35     main(sys.argv[1:]) # Pass command line args (minus script name) to main()
```

**TIP** copy/paste this script to create new scripts

# Files and Console I/O

## Screen output

To output to the screen, use the `print()` function.

- `print()` normally outputs a newline after its arguments.
  - ▶ This can be controlled with the `end` parameter.
- `print()` puts spaces between its arguments by default.
  - ▶ To use a different separator, set the `sep` parameter to the desired separator, which might be an empty string.

## Reading files

To read a file, open it with the `open()` function as part of a `with` statement.

- To read it line by line, iterate through the file with a `for` loop.
- To read the entire file, call the `read()` method on the stream returned by `open()`.
- To read all the lines into a list, use `readlines()` of the stream.
- To read up to `n` bytes from the stream, use the `read(n)` method.

To navigate within a stream, use `seek(offset, whence)`.

To get the current location within the stream, use `tell()`.

## User input

To get input from the user, use the `input()` function.

- It provides a prompt to the user, and returns a string, with the newline already trimmed.

```
file_name = input('What file name? ')
```

# Conditionals

The conditional statement in Python, like most languages, is **if**.

- There are several variations on how **if** is used.
  - ▶ All depend on testing a value to see whether it is **True** or **False**.

The following values are **False**:

- **False**
- Empty collections (empty string, empty list, empty dictionary, empty set, etc.)
- Numeric zero (**0** or **0.0**)
- **None**

Just about everything else is **True**.

- (User-defined objects, and many builtin objects are **True**.)

If you create a class, you can control when it is **True**, and when it is **False**.)

Python has a shortcut if-else that is something like the A?B:C operator in C, Perl and other curly-brace languages.

```
result = value1 if condition else value2
```

*conditionals.py*

```
1 #!/usr/bin/env python
2 import sys
3 import getpass
4
5 name = getpass.getuser()
6
7 if name == 'root':
8     print('do not run this utility as root')
9 elif name == 'guest':
10    print('sorry - guests are not allowed to run this utility')
11 else:
12    print('starting processing')
13    limit = int(sys.argv[1]) if len(sys.argv) > 1 else 100
14    print(limit)
```

# Loops

Python has two kinds of looping constructs.

The `while` loop is often used for reading data, typically from a database or other data source, or when waiting for user input to end a loop.

The `for` loop is used to iterate through a sequence of data.

- Because Python uses iterators to simplify access to many kinds of data, the `for` loop is used in places that would use `while` in most languages.
  - The most frequent example of this is in reading lines from a file.

`while` and `for` loops can also have an `else` block, which is always executed unless a `break` or `return` statement is executed.

- Think of the `else` block as being “on normal loop exit”, i.e., once the condition is false or the collection exhausted.

*loops\_ex.py*

```

1 #!/usr/bin/env python
2
3 # creaet a list
4 colors = ['red', 'green', 'blue', 'purple', 'pink', 'yellow', 'black']
5
6 for color in colors: # loop over list
7     print(color)
8 print()
9
10 with open('../data/mary.txt') as MARY: # open text file for reading
11     for line in MARY: # loop over lines in file
12         print(line, end='')
13     print()
14
15 while True: # loop 'forever'
16     name = input("What is your name? ") # read input from keyboard
17     if name.lower() == 'q':
18         break # exit loop
19     print("Welcome, ", name)
```

# Builtins

Python has many *builtin* functions.

- As of Python 3.7, there are 73 builtin functions
  - ▶ These provide generic functionality that is not tied to a particular type or package.
  - ▶ They can be applied to many different data types, but not all functions can be applied to all data types.
- In addition to the builtin functions there are also data types and constructors defined
  - ▶ A complete list of the builtins can be obtained using the following code:

```
$ python
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> dir(__builtins__)
['ArithmetError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError',
'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError',
'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
'Exception', 'False', 'FileExistsError', 'FileNotFoundException', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError',
'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError',
'__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__',
['__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint',
'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex',
'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec',
'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len',
'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round',
'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple',
'type', 'vars', 'zip']
>>>exit()
$
```

# Exercises

## Exercise 1 (pres\_by\_state.py)

Using the file `presidents.txt` (in the `data` folder), count the number of Presidents born in each state.

- The output of the script should be a table with two columns.
  - ▶ The rows should be sorted by state name in the first column and the number of presidents that were born in that state in the second column.
- The following steps may be helpful in writing the application
  - ▶ Declare a dictionary to hold the data.
  - ▶ Read the file into your script, one line at a time.
  - ▶ Split each line into fields using a colon as the separator.
  - ▶ Add/update the element of the dictionary where the key is the state.
  - ▶ Update the value by 1 each time the state occurs.

### Expected output

```
Arkansas    1
California  1
Connecticut 1
Georgia     1
...
Vermont     2
Virginia    8
```

## Exercise 2 (pres\_dates.py, pres\_dates\_amb.py)

Write an interactive script that asks for a president's last name.

- For each president whose last name matches, print out their date of birth and date of death.
- For presidents who are still alive, print three asterisks for the date of death.

**NOTE**

Dates of death and term end date might be the string "NONE".

### For the ambitious

1. Make the name search case-insensitive and print out matches for partial names—so "jeff" would find "Jefferson", e.g.



# Chapter 2. Packaging and Distributed Python Code

## Objectives

- Use `pyenv` to manage different versions of Python
- Use virtual environments to manage dependencies
- Use `import` effectively
- Create a custom package
- Prepare a package for distribution

## Multiple Python versions with `pyenv`

`pyenv` is a Python version controller allowing for multiple versions of Python to be installed for each user, by storing the installations in the user's home directory.

- Those installations are added to the user's path, and managed through `pyenv`.
  - ▶ Installation requires a git clone and a one-time update to the user's shell.
  - ▶ Because of the shell update, the shell may need to be restarted afterward.

The following bash script is supplied in the `~/advpy/examples/packaging_distributing` directory:

*install\_pyenv.sh*

```
#!/bin/bash
git clone https://github.com/pyenv/pyenv.git ~/.pyenv

TARGET=~/ .bashrc

printf '\n# pyenv configuration begins here\n' >>$TARGET
echo 'export PYENV_ROOT=~/ .pyenv' >>$TARGET
echo 'export PATH=$PYENV_ROOT/bin:$PATH' >>$TARGET

printf 'if command -v pyenv 1>/dev/null 2>&1; then
    eval "$(pyenv init --path)"
fi' >>$TARGET

printf '\n# pyenv configuration ends here\n' >>$TARGET

echo 'Please close the terminal or restart the shell'
```

The following listing shows the install process from the command line using the above bash script:

```
$ cd ~/advpy/examples/packaging_distributing
$ ./install_pyenv.sh
Cloning into '/home/student/.pyenv'...
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 18278 (delta 0), reused 0 (delta 0), pack-reused 18277
Receiving objects: 100% (18278/18278), 3.64 MiB | 22.05 MiB/s, done.
Resolving deltas: 100% (12459/12459), done.
./install_pyenv.sh: 6: [: Illegal number: Linux
Please close the terminal or restart the shell
```

Typing `pyenv install --list` can be used to show all possible installable Python versions and distributions.

New versions of Python can be installed to the user's `~/.pyenv` directory via the `pyenv install` command.

- The following demonstrates the installing of Python version 3.8.5 using `pyenv`.

```
$ pyenv install 3.8.5
Downloading Python-3.8.5.tar.xz...
-> https://www.python.org/ftp/python/3.8.5/Python-3.8.5.tar.xz
Installing Python-3.8.5...
Installed Python-3.8.5 to /home/umbctc/.pyenv/versions/3.8.5
```

`pyenv` builds from source, and as such requires an adequate build environment.

- This build environment varies from system to system, and from python version to python version.
  - ▶ For instance, Python3.7 requires version 1.0.2+ of the SSL library.
- The `pyenv` page on GitHub has guidance on what libraries to be install for a given environment.

<https://github.com/pyenv/pyenv/wiki#suggested-build-environment>

The different installed Python versions can be switched between using the `local`, `shell`, and `global` subcommands to `pyenv`. From then on, invoking `python` will use the set version.

The example below shows using `versions` to list the available installed versions and setting the 3.8.5 version that was installed as the `global` version.

```
$ pyenv versions
* system (set by /home/umbctc/.pyenv/version)
  3.8.5
$
$ pyenv global 3.8.5
$ python --version
Python 3.8.5
$
```

Updating `pyenv` from GitHub to show newer python versions can be done as shown below:

```
$ cd ~/.pyenv
$ git pull
```

# Virtual environments

Azra creates an app using 3rd party **spamlib** and **hamlib** python modules.

- She builds a distribution package and gives it to Mikhail.
- Mikhail installs **spamlib** and **hamlib** on his computer, and then installs Azra's app.
  - ▶ It starts to run, but then crashes with errors.

What happened?

After Azra created her app, the author of **spamlib** module removed a function "that almost no-one used".

- When Mikhail installed **spamlib**, he installed the latest version, which does not have the function.

\*One fix might be for Mikhail to revert to an older version of **spamlib**

- But suppose he has another app that uses the newer version?
- This can get very messy very quickly.

A better solution to this problem is a **virtual environment**.

A virtual environment is a snapshot of a plain Python installation, before any other libraries are added.

- It is used to isolate a particular set of modules that will successfully run a given application.
  - ▶ Each application can have its own virtual environment, which ensures that it has the required versions of dependencies.

There are many tools for creating and using virtual environments, but the primary ones are the **pip** and **venv** modules.

## Preparing the virtual environment

Before creating a virtual environment, it is best to start with a "plain vanilla" Python installation of the desired release.

- The local version of Python 3.8.5 installed prior would act as a good candidate
  - ▶ This could act as more of a reference installation, where everything is always done from a virtual environment with it as its base.

## Creating the environment

The `venv` module provides the tools to create a new virtual environment.

**TIP**

It is common to create a folder named `.venvs` in the home folder to contain all virtual environments.

Its basic usage would be as follows:

```
$ mkdir ~/.venvs
$ cd ~/.venvs
$ python -m venv environment_name
```

- This creates a virtual environment named `environment_name` in the `~/.venvs` directory.
- It is a copy of the required parts of the original installation, not the whole python installation.

## Activating and Deactivating

To use a virtual environment, it must be **activated**.

- This means that it takes precedence over any other installed Python version.
- This is implemented by changing the `PATH` variable in your operating system's environment to point to the virtual copy.

## Activating on Windows

To **activate** the environment on a Windows system, run the `activate.bat` script in the `Scripts` folder of the environment.

```
$ .venvs\environment_name\Scripts\activate
```

## Activating on non-Windows

To activate the environment on a Linux, Mac, or other Unix-like system, source the `activate` script in the `bin` folder of the environment. This must be **sourced**—run the script with the `source` builtin command, or the `. shortcut`

```
$ source ~/.venvs/environment_name/bin/activate
or
$ . ~/.venvs/environment_name/bin/activate
```

**TIP**

To run an app with a particular environment, create a batch file or shell script that activates the environment, then runs the app with the environment's interpreter.

## Deactivating on Windows

When you are finished with an environment, you can deactivate it with the **deactivate** command.

Run the **deactivate.bat** script:

```
$ .venvs\environment_name\Scripts\deactivate
```

## Deactivating on non-Windows

Run the deactivate shell script:

```
$ deactivate
```

## Freezing the environment

When ready to share the app, specify the dependencies by running the **pip freeze** command.

- This will create a list of all the modules added to the virtual environments and their current versions.
  - ▶ Since the output normally goes to **stdout**, it is conventional to redirect the output to a file named **requirements.txt**.

```
$ pip freeze > requirements.txt
```

Now you can provide your **requirements.txt** file to anyone who plans to run your app, and they can create a Python environment with the same versions of all required modules.

## Duplicating an environment

When someone sends you an app with a **requirements.txt** file, it is easy to reproduce their environments.

- Install Python, then use **pip** to add the modules from **requirements.txt**.
  - ▶ The **pip install** command has a **-r** option, which installs the modules listed in the specified file.

```
$ pip install -r requirements.txt
```

# The virtual environment management swamp

There are more tools to help with virtual environment management, and having them all available can be confusing.

- You can always just use `pip` and `venv`, that has already been addressed.

Here are a few of additional tools, and what they do.

## `conda`

A tool provided by Anaconda that replaces both `pip` and `venv`. It assumes you have the Anaconda bundle installed.

## `pipenv`

Another tool that replaces both `pip` and `venv`. It is very convenient, but has some annoying issues.

## `virtualenv`

The original name of the `venv` module.

## `PyCharm`

A Python IDE that will create virtual environments for you. It does not come with Python itself.

## `virtualenvwrapper`

A workflow manager that makes it more convenient to switch from one environment to another.

## Importing modules

The `import` keyword is what brings symbols from one module or package into the current one.

- There are a number of ways to import symbols from a module.

```
import module_name
```

Add the `module_name` symbol to the current working environment.

- It functions as a namespace for all other symbols in `module`.

```
import module_name as alias_of_your_choice
```

Add the `alias_of_your_choice` symbol to the current working environment.

- It functions as a namespace for all other symbols in `module_name`.

```
from module_name import some_existing_symbol
```

Add the `some_existing_symbol` symbol to the current working environment.

- It functions as whatever `some_existing_symbol` is (function, class, etc.).

```
from module_name import some_existing_symbol as alias_of_your_choice
```

Add the `alias` symbol to the current working environment.

- It functions as whatever `symbol` is (function, class, etc.).

```
from module_name import *
```

Add all symbols in `module_name` to the current working environment.

- This should be used extremely rarely, and is usually bad practice.

A module is only loaded once.

- Attempting to import it a second time (assuming the first was successful) will not re-load any part of the module.
  - ▶ This re-loading of the module can be forced via the `importlib.reload()` function.

As projects grow into a collection of modules, organization of those modules often becomes an important part of the overall design of the project.

- Packages are a way of structuring Python's modules together in a single tree-like hierarchy.
  - ▶ This provides organization and helps avoid name collisions.

Making packages is as simple as organizing code as desired on the file-system and optionally adding a file named `__init__.py` to each folder.

In python, "package" is sometimes used casually to refer to both packages and modules.

- Officially, a package is a container for modules, and a module is a python script that ends with ".py".
  - ▶ The type of a package will be reported as a module as seen in the output below:

```
$ cd ~/advpy
$ python
Python 3.8.5 (default, Sep 19 2020, 10:28:38)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import examples
>>> type(examples)
<class 'module'>
>>> quit()
```

# Creating Packages

A **package** is a folder containing modules or subpackages.

- As of Python version 3.3, there is no special requirement to make a folder a package.

Because packages are used with the **import** statement, and become Python objects, they must follow the normal Python symbol naming rules, just like variables, functions, and classes.

## Regular Packages

To create a package, just create a new, empty folder.

- The top-level package is typically the name of an app or module.

You can then place any number of modules or subpackages in the package.

## What is `__init__.py` for?

The `__init__.py` module is used to initialize a package.

- When a package, or anything it contains, is loaded, if the package contains a module named `__init__.py`, it is executed at the time the package is loaded.

There are three main uses for the `__init__.py` module:

- documentation
  - ▶ A docstring at module level in `__init__.py` will be attached to the package, and is thus used for documenting the package itself.
- “courtesy” imports
  - ▶ It is sometimes convenient to make a symbol contained in module, possibly in nested subpackages, available through the top-level package.
  - ▶ This can be done by importing it in `__init__.py`.
- shared resources
  - ▶ Any top-level names in `__init__.py` are available through the package name.

The example on the next page shows the use of packages and an `__init__.py` module

*level\_a/\_\_init\_\_.py*

```
1 """ This package holds all of the modules for the examples in this chapter"""
2 from level_a.level_b.module_c import function_d
```

*level\_a/level\_b/module\_c.py*

```
1 def function_d():
2     print("function has been invoked successfully")
```

*deep\_access.py*

```
1 #!/usr/bin/env python
2 import level_a
3
4
5 level_a.function_d()
6 print()
7 print("docstring for the level_a package:")
8 print(level_a.__doc__)
```

The output of running the above example is shown below.

```
$ python deep_access.py
function has been invoked successfully

docstring for the level_a package:
This package holds all of the modules for the examples in this chapter
$
```

## Namespace Packages

Namespace packages were added in Python 3.3 to make it possible to split a single package across multiple folders.

- However, namespace packages come with several caveats and are not appropriate in all cases.
  - ▶ More info on namespace packages can be found at the following URL:

<https://packaging.python.org/guides/packaging-namespace-packages/>

# Packaging for distribution

## Package files

**pip** is the preferred package manager for modern python.

- Installing a package is as simple as executing **pip install package**.
  - ▶ **pip** is installed by default in modern Python.

Full documentation for pip can be found at the following URL:

```
https://pypi.org/project/pip/
```

The **wheel** module is a third party library used in the building of Python distributions. The following shows how to use **pip** to install the **wheel** module that will be needed for at least one of the examples in this chapter.

```
$ pip install wheel
Collecting wheel
  Downloading wheel-0.35.1-py2.py3-none-any.whl (33 kB)
Installing collected packages: wheel
Successfully installed wheel-0.35.1
WARNING: You are using pip version 20.1.1; however, version 20.2.3 is available.
You should consider upgrading via the '/home/umbctc/.pyenv/versions/3.8.5/bin/python3.8
-m pip install --upgrade pip' command.
$
```

If a warning appears as above, **pip** can be upgraded as shown below.

```
$ python -m pip install --upgrade pip
Collecting pip
  Downloading pip-20.2.3-py2.py3-none-any.whl (1.5 MB)
|██████████| 1.5 MB 3.1 MB/s
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 20.1.1
    Uninstalling pip-20.1.1:
      Successfully uninstalled pip-20.1.1
$
```

When packaging projects for distribution, there are several files to create in the package.

- **README.rst** Is a typical README file, that describes what the package does in brief.
  - ▶ It is not the documentation for the package that should tell how to install the package.
  - ▶ It is common (but not required) to use reStructuredText for this file, hence the .rst extension.
- **MANIFEST.in** is a list of all the non-Python files in the package, such as templates, CSS files, and images.
- **LICENSE** is a file describing the license under which you're releasing the software.
  - ▶ This is less important for in-house apps, but essential for apps that are distributed to the public.
- **setup.py** is a Python script that uses **setuptools** to control how your application is packaged for distribution, and how it is installed by users.
  - ▶ it contains a call to the **setup()** function where named options configure the details of your package.

The following link has some great suggestions for laying out the files in a package:

<https://blog.ionelmc.ro/2014/05/25/python-packaging/>

## Overview of setuptools

The key to using **setuptools** is **setup.py**, the configuration file.

- This tells **setuptools** what files should go in the module, the version of the application or package, and any other configuration data.

The steps for using setuptools are:

- Write a setup script (**setup.py** by convention)
- Write a setup configuration file (optional)
- Create a source, wheel, or specialized built distribution

The entire process and a glossary of packaging and distribution terms can be at the following two URLs:

<https://packaging.python.org/distributing/>

<https://packaging.python.org/glossary/#term-built-distribution>

## Preparing for distribution

The first step in preparing a project for distribution is to create `setup.py` in the package folder.

- `setup.py` is a Python script that calls the `setup()` function from `setuptools` with keyword arguments that describe your module.
- For modules, use the `py_modules` keyword; for packages, use the `packages` keyword.
  - ▶ There are many other options for fine-tuning the distribution.

Your distribution should also have a file named README or README.txt which should be a brief description of the distribution and how to install its module(s).

- You can include any other files desired.
- Many developers include a LICENSE.txt which stipulates how the distribution is licensed.

*Table 2. Keyword arguments for `setup()` function*

Keyword	Description
author	Package author's name
author_email	Email address of the package author
classifiers	A list of classifiers to describe level (alpha, beta, release) or supported versions of Python
data_files	Non-Python files needed by distribution from outside the package
description	Short, summary description of the package
download_url	Location where the package may be downloaded
entry_points	Plugins or scripts provided in the package. Use key <code>console_scripts</code> to provide standalone scripts.
ext_modules	Extension modules needing special handling
ext_package	Package containing extension modules
install_requires	Dependencies. Specify modules your package depends on.
keywords	Keywords that describe the project
license	License for the package
long_description	Longer description of the package
maintainer	Package maintainer's name
maintainer_email	Email address of the package maintainer
name	Name of the package
package_data	Additional non-Python files needed from within the package

Keyword	Description
package_dir	Dictionary mapping packages to folders
packages	List of packages in distribution
platforms	A list of platforms
py_modules	List of individual modules in distribution
scripts	Configuration for standalone scripts provided in the package (but entry_points is preferred)
url	Home page for the package
version	Version of this release

## Creating a source distribution

Run `setup.py` with your version of python, specifying the `sdist` option.

- This will create a platform-independent source distribution.

*temperature\_project/setup.py*

```

1 #!/usr/bin/env python
2
3 from setuptools import setup
4
5
6 setup(name='temperature',
7       version='2.0',
8       description='Temperature management',
9       url='https://umbctraining.com',    # Should be the homepage of project
10      author='UMBC Training Centers',
11      author_email='instructors@umbctraining.com',
12      packages=['temperature', 'temperature.foods', 'temperature.scales'])

```

The output below shows the results of running the `setup.py` script.

```

$ cd ~/advpy/examples/packaging_distributing/temperature_project/
$ python setup.py sdist
running sdist
running egg_info
creating temperature.egg-info
writing temperature.egg-info/PKG-INFO
writing dependency_links to temperature.egg-info/dependency_links.txt
writing top-level names to temperature.egg-info/top_level.txt
writing manifest file 'temperature.egg-info/SOURCES.txt'
.
.
.

copying temperature/foods/__init__.py -> temperature-2.0/temperature/foods
copying temperature/foods/meat.py -> temperature-2.0/temperature/foods
copying temperature/scales/__init__.py -> temperature-2.0/temperature/scales
copying temperature/scales/temperature.py -> temperature-2.0/temperature/scales
Writing temperature-2.0/setup.cfg
creating dist
Creating tar archive
removing 'temperature-2.0' (and everything under it)
$
```

A `dist` directory is created in the project folder and contains a `.tar.gz` file whose contents are shown below.

```
$ ls dist
temperature-2.0.tar.gz

$ tar tf dist/temperature-2.0.tar.gz
temperature-2.0/
temperature-2.0/PKG-INFO
temperature-2.0/README
temperature-2.0/setup.cfg
temperature-2.0/setup.py
temperature-2.0/temperature/
temperature-2.0/temperature/__init__.py
temperature-2.0/temperature/foods/
temperature-2.0/temperature/foods/__init__.py
temperature-2.0/temperature/foods/meat.py
temperature-2.0/temperature/scales/
temperature-2.0/temperature/scales/__init__.py
temperature-2.0/temperature/scales/temperature.py
temperature-2.0/temperature.egg-info/
temperature-2.0/temperature.egg-info/PKG-INFO
temperature-2.0/temperature.egg-info/SOURCES.txt
temperature-2.0/temperature.egg-info/dependency_links.txt
temperature-2.0/temperature.egg-info/top_level.txt
$
```

## Installing from a source distribution

To install a source distribution, extract the resulting `.tar.gz` into any directory and cd into the root (top) of the extracted file structure that contains the `setup.py` file.

- Then execute the following command:

```
$ python setup.py install
```

The following listings show the process of installing the `temperature-2.0.tar.gz` distribution just created.

The first step shown below will copy the `temperature-2.0.tar.gz` file into a new directory.

- This is to show that the whole process is independent of where the original source came from.

```
$ mkdir ~/transferred_project
$ cd ~/advpy/examples/packaging_distributing/temperature_project/dist
$ cp temperature-2.0.tar.gz ~/transferred_project
$
```

The next step will create a virtual environment named `temperature-venv` that the distribution will be installed to.

- The use of the virtual environment localizes the install to only that environment as opposed to the global Python environment declared using `pyenv` earlier.

```
$ cd ~/.venvs
$ python -m venv temperature-venv
$ source ~/.venvs/temperature-venv/bin/activate
(temperature-venv) $
```

- Notice the command prompt has changed `(temperature-venv) $` to indicate the virtual environment is activated.

The next step is to extract the distribution file and install it using `setup.py`

```
(temperature-venv) $ cd ~/transferred_project
(temperature-venv) $ tar -xzf temperature-2.0.tar.gz
(temperature-venv) $ cd temperature-2.0/
(temperature-venv) $ python setup.py install
running install
running bdist_egg
running egg_info
writing temperature.egg-info/PKG-INFO
writing dependency_links to temperature.egg-info/dependency_links.txt
writing top-level names to temperature.egg-info/top_level.txt
.
.
.

Adding temperature 2.0 to easy-install.pth file

Installed /home/student/.venvs/temperature-venv/lib/python3.8/site-packages/temperature-2.0-py3.8.egg
Processing dependencies for temperature==2.0
Finished processing dependencies for temperature==2.0
(temperature-venv) $
```

With the installation of the distribution successful, the following output of the Python interactive shell shows that the install modified the **PYTHONPATH** to include the new distribution.

```
(temperature-venv) $ python
Python 3.8.5 (default, Sep 19 2020, 20:30:52)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> import sys
>>> for path in sys.path:
...     print(path)
...
/home/umbctc/.pyenv/versions/3.8.5/lib/python38.zip
/home/umbctc/.pyenv/versions/3.8.5/lib/python3.8
/home/umbctc/.pyenv/versions/3.8.5/lib/python3.8/lib-dynload
/home/umbctc/.venvs/temperature-venv/lib/python3.8/site-packages
/home/umbctc/.venvs/temperature-venv/lib/python3.8/site-packages/temperature-2.0-
py3.8.egg
>>> exit()
(temperature-venv) $
```

The following interactive Python shell shows how the newly installed distribution can easily be used within a script.

```
(temperature-venv) $ python
Python 3.8.5 (default, Sep 19 2020, 20:30:52)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import temperature.scales.temperature as temper
>>> temper.c2f(100)
212.0
>>> exit()
(temperature-venv) $
```

## Creating wheels

A wheel is prebuilt distribution, that can be installed with `pip`.

- A wheel is a `setuptools` extension for building wheels that provides the `bdist_wheel setuptools` command.

A wheel can be one of 3 different kinds.

- A Universal wheel is a pure Python package (no extensions) that can be installed on either Python 2 or Python 3.
  - ▶ It has to have been carefully written that way.
- A Pure Python wheel is a pure Python package that is specific to one version of Python (either 2 or 3).
  - ▶ It can only be installed by a matching version of pip.
- A Platform wheel is a package that has extensions, and thus is platform-specific.

The example that follows builds a pure python wheel for the `temperature-project` within the `temperate-venv` virtual environment.

```
(temperature-venv) $ cd ~/advpy/examples/packaging_distributing/temperature_project/
(temperature-venv) $ python setup.py bdist_wheel
running bdist_wheel
running build
running build_py
creating build
creating build/lib
creating build/lib/temperature
.
.
.
creating 'dist/temperature-2.0-py3-none-any.whl' and adding 'build/bdist.linux-x86_64/wheel' to it
.
.
.
adding 'temperature-2.0.dist-info/RECORD'
removing build/bdist.linux-x86_64/wheel
(temperature-venv) $
```

The `dist` directory for the project now contains a wheel named `temperature-2.0-py3-none-any.whl` file.

```
$ ls dist
temperature-2.0.tar.gz
temperature-2.0-py3-none-any.whl
```

The next set of listings show how to use `pip` to install a local `.whl` file.

The first step shown below will copy the `temperature-2.0-py3-none-any.whl` file into a new directory. \*\* This is to show, once again, that the whole process is independent of where the original source came from.

```
$ mkdir ~/wheel_project
$ cd ~/advpy/examples/packaging_distributing/temperature_project/dist
$ cp temperature-2.0-py3-none-any.whl ~/wheel_project
$
```

The next step will create a virtual environment named `from-wheel-venv` that the distribution will be installed to.

- The script first deactivates any current virtual environment

```
$ deactivate
$ cd ~/.venvs
$ python -m venv temperature-venv
$ source ~/.venvs/temperature-venv/bin/activate
(from-wheel-venv) $
```

- Notice the command prompt has changed `(from-wheel-venv) $` to indicate the virtual environment is activated.

The next step is to install the wheel using `pip`.

```
(from-wheel-venv) $ cd ~/wheel_project
(from-wheel-venv) $ pip install temperature-2.0-py3-none-any.whl
Processing ./temperature-2.0-py3-none-any.whl
Installing collected packages: temperature
Successfully installed temperature-2.0
(from-wheel-venv) $
```

With the installation of the distribution successful, the following output of the Python interactive shell shows once again that the install modified the **PYTHONPATH** to include the new distribution.

```
(temperature-venv) $ python
Python 3.8.5 (default, Sep 19 2020, 20:30:52)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> import sys
>>> for path in sys.path:
...     print(path)
...
/home/umbetc/.pyenv/versions/3.8.5/lib/python38.zip
/home/umbetc/.pyenv/versions/3.8.5/lib/python3.8
/home/umbetc/.pyenv/versions/3.8.5/lib/python3.8/lib-dynload
/home/umbetc/.venvs/from-wheel-venv/lib/python3.8/site-packages
>>>
>>> import temperature.scales.temperature as temper
>>> temper.c2f(0)
32.0
>>> exit()
(temperature-venv) $ deactivate
$
```

## Code Portability

Before thinking about distributing packages, code portability should be considered.

If developing on the same platform where the code will be deployed, this is not so much an issue, but it's always best to expect the unexpected.

- Maybe next month the platform the code is run on will move from HP-UX to Red Hat linux.
  - ▶ Consider using `sys.platform` to detect the current platform, and isolate platform-specific code.

Many portability issues revolve around the file system and external utilities.

- For filenames, always use forward slashes to separate directories.
  - ▶ If you need it, `os.sep` contains the directory separator.
- Use `pathlib.Path` to concatenate parts of a filename.
  - ▶ It will do the right thing for the current OS.

Another useful module in the standard library is `shutil`.

- This module contains platform-neutral functions to copy, rename, and delete files.
- It also contains platform-neutral functions to work with folders, and many other tasks.

# Exercises

## Exercise 1

Create a virtual environment named `arithmetic` and use it for the remaining exercises below.

## Exercise 2

Create a project that deals with modules relating to arithmetic functions.

- There should be sub-packages `basic` and `advanced`.
- Addition, subtraction, and multiplication modules should be in the basic sub-package while the advanced one should have power and squareroot modules.
  - ▶ Each sub-module should have a function that carries out the stated operation.

## Exercise 3

Extend the `arithmetic` project so that an application may import all of the functions in all of the modules. The import should be able to look something like the following:

```
from arithmetic import add, subtract, multiply, divide
```

## Exercise 4

Create the necessary files to package up the `arithmetic` project.

## Exercise 5

Package up the `arithmetic` project as a wheel.

# Chapter 3. Pythonic Programming

## Objectives

- Learn what makes code "Pythonic"
- Understand some Python-specific idioms
- Create lambda functions
- Perform advanced slicing operations on sequences
- Distinguish between collections and generators
- Use `unittest` to verify functionality

## The Zen of Python

The Zen pf Python is a set of guidelines for writing code that is Pythonic

- It was written by Tim Peters, a longtime contributor to Python.
  - ▶ One such contribution to the codebase is the sorting routine, known as "timsort".

The Zen of Python forms the basis of PEP20 and can be found at the following URL:

<https://www.python.org/dev/peps/pep-0020/>

The text of PEP20 is also printed out if you `import this` in your code as shown below.

```
$ python
Python 3.8.5 (default, Sep 19 2020, 10:28:38)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 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!
>>> exit()
$
```

# Tuples

While a **tuple** and a **list** can be used to store a collection of any data:

- Use a list when you have a collection of similar objects.
  - ▶ Such as the days of the week or grades on an exam for example.
- Use a tuple when you have a collection of related objects, which may or may not be similar.
  - ▶ For example the first name, last name and age of a person.
  - ▶ They are related in that they all pertain to a person
  - ▶ But they are not similar in that some are strings and some are numeric

While on the surface it seems like just a read-only list, it is often used when you need to pass multiple values to or from a function, but the values are not all the same type.

- A variable number of arguments is passed to a function that has a parameter using the \* as a **tuple**

To create a **tuple**, use a comma-separated list of objects.

- Parentheses are not needed around a tuple unless the tuple is nested in a larger data structure.
- To specify a one-element tuple, use a trailing comma, otherwise it will be interpreted as a single object.

A tuple in Python might be represented by a struct or a "record" in other languages.

*creating\_tuples.py*

```

1 #!/usr/bin/env python
2
3 # Explicitly defining a tuple
4 hostinfo = ('gemini', 'linux', 'ubuntu', 'hardy', 'Bob Smith')
5 birthday = ('April', 5, 1978)
6
7 # Implicitly defining a tuple
8 person = "Nadeem", "Sharif", 37
9
10 # color is a one item tuple
11 color = ('red',)
12
13 # color is a string
14 color = ('red')
```

## Iterable unpacking

An iterable such as a tuple or list provides access to individual elements by index.

- However, `birthday[0]`, `birthday[1]`, and `birthday[2]` are not as readable compared to `month`, `day`, and `year`.
  - ▶ To unpack an iterable into individual variable names, just assign the iterable to a comma-separated list of names:

```
birthday = ('April', 5, 1978)
month, day, year = birthday
```

Why not just assign to the variables in the first place?.

- For a single tuple or list, this would be true.
- The power of unpacking comes in the following areas:
  - ▶ Looping over a nested iterable of iterables
  - ▶ Passing tuples (or other iterables) into a function

*unpacking\_people.py*

```
1 #!/usr/bin/env python
2
3
4 # A list of 3-element tuples
5 people = [
6     ('Melinda', 'Gates', 'Gates Foundation'),
7     ('Steve', 'Jobs', 'Apple'),
8     ('Larry', 'Wall', 'Perl'),
9     ('Paul', 'Allen', 'Microsoft'),
10    ('Larry', 'Ellison', 'Oracle'),
11    ('Bill', 'Gates', 'Microsoft'),
12    ('Mark', 'Zuckerberg', 'Facebook'),
13    ('Sergey', 'Brin', 'Google'),
14    ('Larry', 'Page', 'Google'),
15    ('Linus', 'Torvalds', 'Linux')
16 ]
17
18 # The for loop unpacks each tuple into the three variables.
19 # Even though only two of the three are needed
20 for first_name, last_name, org in people:
21     print(first_name, last_name)
```

# Unpacking function arguments

Sometimes you need the other end of iterable unpacking.

What do you do if you have a list of three values, and you want to pass them to a method that expects three positional arguments?

- One approach is to use the individual items by index, but this will not scale well for large lists
- A more Pythonic approach is to use \* to *unpack* the iterable into individual items:
  - ▶ Use a single asterisk to unpack a list or tuple (or similar iterable)
  - ▶ Use two asterisks to unpack a dictionary or similar.

*unpacking\_function\_args.py*

```

1 #!/usr/bin/env python
2
3 # A list of 4-element tuples
4 people = [
5     ('Joe', 'Schmoe', 'Burbank', 'CA'),
6     ('Mary', 'Rattburger', 'Madison', 'WI'),
7     ('Jose', 'Ramirez', 'Ames', 'IA'),
8 ]
9
10
11 # A function that takes 4 parameters
12 def person_record(first_name, last_name, city, state):
13     print("{} {} lives in {}, {}".format(first_name, last_name, city, state))
14
15
16 for person in people: # Each person is a 4-element tuple from people
17     # *person unpacks the tuple into four individual parameters
18     # This is also sometimes referred to as the "splat operator"
19     person_record(*person)

```

The next example unpacks a list as parameters to the string formatting `format()` method.

```
1 #!/usr/bin/env python
2 BARLEYCORN = 1 / 3.0
3 CM_TO_INCH = 2.55
4 MENS_START_SIZE = 12
5 WOMENS_START_SIZE = 10.5
6
7 FMT = '{:6.1f} {:8.2f} {:8.2f}'
8 HEADFMT = '{:>6s} {:>8s} {:>8s}'
9 HEADINGS = ['Size', 'Inches', 'CM']
10
11 SIZE_RANGE = []
12 for i in range(6, 14):
13     SIZE_RANGE.extend([i, i + .5])
14
15
16 def main():
17     for heading, flag in [("MEN'S", True), ("WOMEN'S", False)]:
18         print(heading)
19
20         # format() expects individual arguments for each placeholder;
21         # the asterisk unpacks HEADINGS into individual string arguments
22         print((HEADFMT.format(*HEADINGS)))
23
24         for size in SIZE_RANGE:
25             inches, cm = get_length(size, flag)
26             print(FMT.format(size, inches, cm))
27             # The above print could have been written as:
28             # print(FMT.format(size, *get_length(size, flag)))
29             # by unpacking the return value of get_length()
30             # but tends to make it too busy and hard to read.
31         print()
32
33
34 def get_length(size, mens=True):
35     start_size = MENS_START_SIZE
36     if not mens:
37         start_size = WOMENS_START_SIZE
38
39     inches = start_size - ((start_size - size) * BARLEYCORN)
40     cm = inches * CM_TO_INCH
41     return inches, cm
42
43
44 if __name__ == '__main__':
45     main()
```

## The `sorted()` function

The `sorted()` builtin function takes an iterable as a parameter and returns it as a sorted `list`.

*basic\_sorting.py*

```

1 #!/usr/bin/env python
2 """Basic sorting example"""
3
4 fruits = ["pomegranate", "cherry", "apricot", "date", "Apple", "lemon", "Kiwi",
5           "ORANGE", "lime", "Watermelon", "guava", "papaya", "FIG", "pear",
6           "banana", "Tamarind", "persimmon", "elderberry", "peach", "grape",
7           "lychee", "BLUEberry"]
8
9 sorted_fruit = sorted(fruits) # sorted() always returns a list
10 # determine length of longest string in list
11
12 fmt = "{:18}"
13 for counter, fruit in enumerate(sorted_fruit):
14     print(fmt.format(fruit), end=' ')
15     if counter % 4 == 3:
16         print()
17 print()

```

The results of running the above script are shown below:

```

$ python basic_sorting.py
Apple          BLUEberry      FIG        Kiwi
ORANGE         Tamarind       Watermelon apricot
banana         cherry        date       elderberry
grape          guava         lemon      lime
lychee          papaya       peach      pear
persimmon      pomegranate
$
```

## Custom sort keys

A reference to a function can be passed as a parameter to the `sorted()` function with a named parameter of `key`.

- This function will be invoked once for each element of the list being sorted, to provide the alternative comparison value.
  - ▶ Thus for example, a list of strings can be sorted case-insensitively, or sort a list of ZIP codes by the number of Starbucks within the ZIP code.
- The function must take exactly one parameter (which is one element of the sequence being sorted) and return either a single value of any type.
  - ▶ The returned values will be used by the `sorted()` method to modify the order based on its comparison.

Any builtin or custom Python function or method that meets these requirements can be used.

The following example demonstrates altering the normal sorting order of the `sorted()` method for a list of numbers.

*custom\_sort\_numbers.py*

```

1 #!/usr/bin/env python
2
3 nums = [800, 80, 1000, 32, 255, 400, 5, 5000]
4
5 n1 = sorted(nums) # Numbers sort from low to high by default
6 print("Numbers sorted numerically:")
7 for n in n1:
8     print(n, end=' ')
9 print("\n")
10
11 n2 = sorted(nums, key=str) # Sort numbers as strings instead of numerically
12 print("Numbers sorted as strings:")
13 for n in n2:
14     print(n, end=' ')
15 print()

```

The next example will demonstrate various functions that can be used to alter the normal sorting order of the `sorted()` method for a list of strings.

*custom\_sort\_strings.py*

```
1 #!/usr/bin/env python
2
3
4 def ignore_case(item): # Parameter is one element of iterable to be sorted
5     return item.lower() # Return value to sort on instead of item
6
7
8 def by_length_then_name(item):
9     # Key functions can return tuple of values to compare, in order
10    # Since lists and tuples automatically sort to the n-th element level
11    return (len(item), item.lower())
12
13
14 fruit = ["pomegranate", "cherry", "apricot", "date", "Apple", "lemon", "Kiwi",
15           "ORANGE", "lime", "Watermelon", "guava", "papaya", "FIG", "pear",
16           "banana", "Tamarind", "persimmon", "elderberry", "peach", "BLUEberry"]
17
18 fs1 = sorted(fruit, key=ignore_case) # register function to use to sort
19 print("Ignoring case:")
20 print(" ".join(fs1), end="\n\n")
21
22 fs2 = sorted(fruit, key=by_length_then_name)
23 print("By length, then name:")
24 print(" ".join(fs2))
25 print()
```

The `sort()` method of `list` provides the same type of custom sorting as shown in the next example.

- While the top level `sorted()` function returns a new `list`, the `list.sort()` method modifies the existing `list` in place.

`sort_holmes.py`

```
1 #!/usr/bin/env python
2 """Sort titles, ignoring leading articles"""
3 import re
4
5 books = [
6     "A Study in Scarlet", "The Sign of the Four", "The Valley of Fear",
7     "The Hound of the Baskervilles", "The Adventures of Sherlock Holmes",
8     "The Memoirs of Sherlock Holmes", "The Return of Sherlock Holmes",
9     "His Last Bow", "The Case-Book of Sherlock Holmes"]
10
11 # compile regex to match leading articles
12 rx_article = re.compile(r'^the|an)\s+', re.I)
13
14
15 # create function which takes element to compare and returns comparison key
16 def strip_articles(title):
17     # strip off article and convert title to lower case
18     return rx_article.sub('', title.lower())
19
20
21 for book in sorted(books, key=strip_articles): # sort using custom function
22     print(book)
```

# Lambda functions

A **lambda function** is a one line function created on the fly that has no name.

- This can be useful for passing functions into other functions, to be called later.
  - ▶ Functions passed in this way are referred to as "callbacks".
  - ▶ Normal functions can be callbacks as well.
    - ◆ One advantage of a lambda function is its convenience.
    - ◆ Another is the ability to register a function that when invoked can easily be passed local variables.

One common use of lambda functions is for providing functions as keys to other functions.

The basic syntax for creating a lambda function is `lambda parameter_list: expression`

- Where `parameter_list` is a list of function parameters and `expression` is an expression involving the parameters.
  - ▶ The value the expression evaluates to acts as the return value of the lambda.

While the equivalent of a lambda function could also be defined in the normal manner as shown below.

```
def anonymous(parameter_list):
    return expression
```

- It is not possible to use the normal syntax as a function parameter, or as an element in a list.
  - ▶ This is one of the advantages of using lambda functions.

*lambda\_examples.py*

```
1 #!/usr/bin/env python
2
3 fruits = ['Pomegranate', 'Apple', 'Mango', 'KIWI', 'apricot', 'Watermelon']
4
5 # The lambda function is passed one fruit and returns it in lower case
6 fruits.sort(key=lambda e: (len(e), e.lower()))
7 print(" ".join(fruits))
8
9 # the top level max() function also allows a function to be passed as a key.
10 print(max(fruits), max(fruits, key=lambda f: len(f)))
11 # Though it is much simpler in this case to simply rely on the built-in len
12 print(max(fruits, key=len))
```

# Comprehensions

## List comprehensions

A **list comprehension** consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses.

- The result is a new list resulting from evaluating the expression in the context of the for and if clauses which follow it.
  - ▶ Functional programmers refer to this as a mapping function.

*listcomp.py*

```

1 #!/usr/bin/env python
2
3 fruits = ['watermelon', 'apple', 'mango', 'kiwi', 'apricot', 'lemon', 'guava']
4
5 # Creating a list from an iterable without using a list comprehensions
6 results_a = []
7 for fruit in fruits:
8     results_a.append(fruit.upper())
9
10 # Creating the list using a list comprehension instead of a explicit for loop
11 results_b = [fruit.upper() for fruit in fruits]
12
13 # The following list comprehension uses an if to filter out
14 # fruits that do not start with the letter 'a'.
15 results_c = [fruit for fruit in fruits if fruit.startswith('a')]
16
17 print("results_a:", ".join(results_a))
18 print("results_b:", ".join(results_b))
19 print("results_c:", ".join(results_c))
20
21
22 values = [2, 42, 18, 92, "boom", ['a', 'b', 'c']]
23
24 # list comprehension doubles each value
25 doubles = [value * 2 for value in values]
26
27
28 print("doubles:", end=' ')
29 for doubled in doubles:
30     print(doubled, end=' ')
31 print()

```

## Set comprehensions

A **set comprehension** is useful for turning any sequence into a set based on a given function.

- The syntax is very similar to a list comprehension.
  - Where a list comprehension uses [ ], a set comprehension uses { }

*set\_comprehension.py*

```
1 #!/usr/bin/env python
2
3 with open("../data/mary.txt") as mary_in:
4     # Get unique words from file
5     # Only one line (ln) is in memory at any point
6     # Skip "empty" words
7     s = {w.lower() for ln in mary_in for w in ln.split() if w}
8
9 print(s)
```

## Dictionary comprehensions

A **dictionary comprehension** has syntax similar to a set comprehension.

- There are two expressions separated by a colon
  - The value of the first expression becomes the key added to the resulting dictionary
  - The value of the second expression is used as the value added to the resulting dictionary.

*dict\_comprehension.py*

```
1 #!/usr/bin/env python
2
3
4 animals = ['OWL', 'Badger', 'cat', 'Tiger', 'Wombat', 'GORILLA', 'AARDVARK']
5 # Create a dictionary with key/value pairs derived from an iterable
6 result = {a.lower(): len(a) for a in animals}
7 fmt = "Key:{:10} : Value:{}"
8 for animal, num_chars in result.items():
9     print(fmt.format(animal, num_chars))
```

# Iterables

Python has many builtin iterables.

- A file object, for instance, which allows iterating through the lines in a file.
- Collections and generators are also examples of iterables.

Collections keep all their values in memory.

- Collections can be further subdivided into sequences and mappings
  - ▶ The following are some examples of data types and functions that are sequences:
    - ◆ `str, bytes, list, tuple, collections.namedtuple, sorted`, and list comprehensions.
  - ▶ The following are some examples of data types and functions that are mappings:
    - ◆ `dict, set, frozenset, collections.defaultdict`, and `collections.Counter`.

Generators do not keep all its values in memory, it creates them one at a time as needed.

- This is a Good Thing, because it saves memory.
  - ▶ The following are some examples of data types and functions that are generators
    - ◆ `open, range, enumerate, zip, dict.items`, generator expressions, and generator functions.

## for Loop Syntactic Sugar

All iterable objects have a `__iter__` method defined on them.

- This method is invoked to retrieve an iterator object, which has a `__next__` method.

*for\_loops.py*

```

1#!/usr/bin/env python
2for number in [4, 8, 9, 3, 5, 1]:
3    print(number)
4
5# The above for loop is simply syntactic sugar for the following code
6iterator = iter([4, 8, 9, 3, 5, 1]) # Really invokes list.__iter__()
7try:
8    while True:
9        number = next(iterator) # Really invokes iterator.__next__()
10       print(number)
11except StopIteration:
12    pass

```

# Generator Functions

A generator is like a normal function, but instead of a return statement, it has a `yield` statement.

- Each time the `yield` statement is reached, it provides the next value in the sequence.
  - ▶ When there are no more values, the function call returns, and the loop stops.
- A generator function maintains state between yields, unlike a normal function.

*line\_trimmer.py*

```

1 #!/usr/bin/env python
2
3
4 def trimmed(file_name):
5     with open(file_name) as file_in:
6         for line in file_in:
7             # existence of a 'yield' causes function to return a generator
8             yield line.rstrip()
9
10
11 # looping over the a generator object returned by trimmed()
12 for trimmed_line in trimmed('../data/mary.txt'):
13     print(trimmed_line)

```

*sieve\_generator.py*

```

1 #!/usr/bin/env python
2
3
4 def all_primes(limit):
5     flags = set() # initialize empty set (to be used for "is-prime" flags
6
7     for i in range(2, limit):
8         if i in flags:
9             continue
10        for j in range(2 * i, limit + 1, i):
11            flags.add(j) # add non-prime elements to set
12        yield i # execution stops here until next value is requested
13
14
15 np = all_primes(50) # next_prime() returns a generator object
16 for prime in np:    # iterate over *yielded* primes
17     print(prime, end=' ')
18 print()

```

## Generator Expressions

A **generator expression** is similar to a comprehension, but it provides a generator instead of a **list**, **set**, or **dict**.

- The main difference in syntax is that the generator expression uses parentheses rather than brackets or curly braces.
  - ▶ While a comprehension returns a complete collection, a generator expression returns one item at a time.

Generator expressions are especially useful with functions like `sum()`, `min()`, and `max()` that all reduce an iterable to a single value:

**NOTE** There is an implicit `yield` statement at the beginning of the expression.

`gen_ex.py`

```
1 #!/usr/bin/env python
2
3 limit = 10**8
4
5 # sum the squares of a list of numbers
6 # using list comprehension, entire list is stored in memory
7 s1 = sum([x * x for x in range(limit)])
8
9 # only one square is in memory at a time with generator expression
10 s2 = sum(x * x for x in range(limit))
11 print(s1, s2)
12 print()
13
14 page = open("../data/mary.txt")
15 # Only one line in memory at a time; max() iterates over generated values
16 m = max(len(line) for line in page)
17 page.close()
18 print(m)
```

# String formatting

The traditional (i.e., old) way to format strings in Python was with the `%` operator and a format string containing fields designated with percent signs.

The new, improved method of string formatting uses the `format()` method.

- It takes a format string and a variable number of arguments.
  - ▶ The format string contains placeholders which consist of curly braces, which may contain formatting details.
  - ▶ This new method has much more flexibility.

By default, the placeholders are numbered from left to right, starting at 0.

- This corresponds to the order of arguments to `format()`.

Formatting information can be added, preceded by a colon.

- format the argument as an integer

`{:d}`

- format as an integer, minimum width of 3, zero padded

`{:03d}`

- format as a string, minimum width 25 characters, right-justified

`{:>25s}`

- format as a float, with 3 decimal places

`{:.3f}`

- Placeholders can be manually numbered.

- ▶ This is handy when you want to use a `format()` parameter more than once.

"Try one of these: `{0}.jpg {0}.png {0}.bmp {0}.pdf`".`format('penguin')`

```
1 #!/usr/bin/env python
2
3 color = 'blue'
4 animal = 'iguana'
5
6 # {} placeholders are autonumbered by default, starting at 0
7 # this corresponds to the parameters to format()
8 print('{} {}'.format(color, animal))
9
10 fahr = 98.6839832
11
12 # Formatting directives start with ':'
13 # .1f means format floating point with one decimal place
14 print('{:.1f}'.format(fahr))
15
16 value = 12345
17
18 # {} placeholders can be manually numbered to reuse parameters
19 print('{0:d} {0:04x} {0:08o} {0:016b}'.format(value))
20
21 data = {'A': 38, 'B': 127, 'C': 9}
22
23 for letter, number in sorted(data.items()):
24     # 4d means format decimal integer in a field 4 characters wide
25     print("{} {}".format(letter, number))
```

## f-strings

Literal f-strings were added to Python in version 3.6.

- These are strings that contain placeholders, as used with normal string formatting, but the expression to be formatted is also placed in the placeholder.
  - ▶ This makes formatting strings more readable, with less typing.
- As with formatted strings, any expression can be formatted.

Other than putting the value to be formatted directly in the placeholder, the formatting directives are the same as normal Python 3 string formatting.

In normal str.format() formatting:

```
x = 24
y = 32.2345
name = 'Bill Gates'
company = 'Microsoft'
print("{} founded {}".format(name, company))
print("{:10s} {:.2f}".format(x, y))
```

f-strings let you do this:

```
x = 24
y = 32.2345
name = 'Bill Gates'
company = 'Microsoft'
print(f"{name} founded {company}")
print(f"{x:10s} {y:.2f}")
```

```
1 #!/usr/bin/env python
2
3 import sys
4
5 name = 'Tim'
6 count = 5
7 avg = 3.456
8 info = 2093
9 result = 38293892
10
11 # < means left justify (default for non-numbers)
12 # 10 is the minimum field width, s formats a string
13 print(f'Name is [{name:<10s}]')
14
15
16 print(f'Name is [{name:>10s}]') # > means right justify
17
18 # .2f means round a float to 2 decimal points
19 print(f'count is {count:03d} avg is {avg:.2f}')
20
21 # d is decimal, o is octal, x is hex
22 print(f'info is {info} {info:d} {info:o} {info:x}')
23
24 # , means add commas to numeric value
25 print(f'${result:,d}')
26
27 city = 'Orlando'
28 temp = 85
29
30 # parameters can be selected by name instead of position
31 print(f'It is {temp} in {city}')
```

# unittest and Unit Testing

There are an abundance of testing frameworks and libraries available to choose from in Python.

- A expansive list of choices can be found at

<https://wiki.python.org/moin/PythonTestingToolsTaxonomy>

The `unittest` module is part of the Python standard library and will be the main framework used in this module.

- It is a unit testing framework inspired by a Java testing framework called `Junit`.

The basics of a unit test in this module is to create a test case by creating a subclass of the `unittest.TestCase` class.

There is an automatic discovery of unit tests within Python packages by the `unittest` module.

- The discovery process requires the use of `__init__.py` even though the Python interpreter no longer requires them.

The set of examples that follows are all in a grouped together in a subfolder named 'tdd'

- Each example will use the `unittest` module to create a failing test as the first part of Test-Driven Development (TDD) development cycle.
  - ▶ The next step of each example will be to right the minimum amount of code necessary to make the test pass as the second part of the TDD cycle.
  - ▶ The third step of the TDD cycle, refactoring, will be held off in its implementation until the last example.

The requirement will be to develop an `Employee` class.

- An Employee should have a first name, last name, job title, and hire date, with the ability to get and set each of the four properties.

The example starts by writing a failing test for instantiating an Employee object.

*tdd/version01/employee\_test.py*

```

1 import unittest
2
3
4 class TestEmployee(unittest.TestCase):
5
6     def test_employee_object_creation(self):
7         emp = Employee()
8
9
10 if __name__ == "__main__":
11     unittest.main()
```

By convention a file ending with `_test` in its name is created first.

- A class is then created that extends `unittest.TestCase`.
  - ▶ The failing test is then declared within the class as a method whose name is required to start with `test`.
  - ▶ The remainder of the method name should give some insight as to what the method actually tests.

The test is run by calling the `main()` function within the `unittest` module.

```

> python employee_test.py
E
=====
ERROR: test_employee_object_creation (__main__.TestEmployee)
-----
Traceback (most recent call last):
  File "employee_test.py", line 6, in test_employee_object_creation
    emp = Employee()
NameError: name 'Employee' is not defined
-----
Ran 1 test in 0.000s
FAILED (errors=1)
>
```

The test fails with an error since the `Employee` class has not been defined yet.

- In order for the test to pass, the `Employee` class must be created and imported into the test.

- ▶ Version 2 of the example write the code necessary to allow the above to pass the test.

*tdd/version02/employee.py*

```
1 class Employee:
2     pass
```

*tdd/version02/employee\_test.py*

```
1 import unittest
2 import employee
3
4
5 class TestEmployee(unittest.TestCase):
6
7     def test_employee_object_creation(self):
8         emp = employee.Employee()
9
10
11 if __name__ == "__main__":
12     unittest.main()
```

```
> python3 employee_test.py
```

```
.
```

```
-----
```

```
Ran 1 test in 0.000s
```

```
OK
```

```
>
```

With the `Employee` class written, behavior can begin to be added to meet any additional requirements of what an `Employee` object is supposed to be capable of.

- The original requirements for the example were stated as follows:
  - ▶ An `Employee` should have a first name, last name, job title, and hire date, with the ability to get and set each of the four properties.

Since it is common to have a constructor that takes as parameters the data necessary to represent an employee, the existing test will be modified such that it once again will fail when it attempts to call such a constructor.

- If the need to construct an `Employee` in several different ways was needed, it might be necessary to create an additional test as opposed to modifying the existing test.

- 
- ▶ But in the case of this example, ultimately only a constructor that needs to take in four arguments is needed, so the existing test will be modified.

Version 3 of the example is shown below.

*tdd/version03/employee\_test.py*

```

1 import unittest
2 import datetime
3 import employee
4
5
6 class TestEmployee(unittest.TestCase):
7
8     def test_employee_object_creation(self):
9         emp = employee.Employee("Karen", "Jones", "Manager",
10                               datetime.date.today())
11
12
13 if __name__ == "__main__":
14     unittest.main()

```

Running the above test with the existing `Employee` class will fail as expected.

- The test would fail because the existing constructor is not designed to take the parameters being passed.
- The minimum code necessary to pass the test has already been incorporated into this version 3 and is shown below.

*tdd/version03/employee.py*

```

1 class Employee:
2     def __init__(self, first_name, last_name, job, hired):
3         pass

```

Of course, the real purpose of defining the constructor to accept the four parameters is for the constructor to ultimately store the parameters as instance data inside of the object being created.

The next test that will be developed will use several assert methods from the `unittest.TestCase` class to provide standard testing techniques for use in the test methods.

`tdd/version04/employee_test.py`

```

1 import unittest
2 import datetime
3 import employee
4
5
6 class TestEmployee(unittest.TestCase):
7
8     def test_employee_object_creation(self):
9         emp = employee.Employee("Karen", "Jones", "Manager",
10                             datetime.date.today())
11
12    def test_get_employee_properties(self):
13        emp = employee.Employee("Karen", "Jones", "Manager",
14                             datetime.date.today())
15        self.assertEqual(emp.first_name, "Karen")
16        self.assertEqual(emp.last_name, "Jones")
17        self.assertEqual(emp.job, "Manager")
18        self.assertEqual(emp.hired, datetime.date.today())
19
20
21 if __name__ == "__main__":
22     unittest.main()

```

The `TestEmployee` class above consists of two tests.

- The `test_employee_object_creation` test still passes as it did previously.
- The new `test_get_employee_properties` test fails since the `Employee` class currently does not store the data passed to its constructor.

The tests within a `unittest.TestCase` class are run alphabetically based on the name of the test:

- `test_emp…` comes before `test_get…`
  - ▶ It is important for proper testing that the order the tests are invoked should be irrelevant.
  - ▶ This ensures there are no unseen dependencies between the tests that would lead to improper testing.

The basic design of a test case involves using various assert methods of the `unittest.TestCase` class to prove that the expected results were obtained within the assertion.

The table below lists some of the most common assert methods available.

- A complete list of methods in `unittest.TestCase` can be found in the documentation.

*Table 3. `unittest.TestCase` Assert Methods*

<code>assertEqual(x, y)</code>	<code>x == y</code>
<code>assertNotEqual(x, y)</code>	<code>x != y</code>
<code>assertAlmostEqual(x, y)</code>	<code>round(x - y, 7) == 0</code>
<code>assertTrue(x)</code>	<code>bool(x) is True</code>
<code>assertFalse(x)</code>	<code>bool(x) is False</code>
<code>assertIs(x, y)</code>	<code>x is y</code>
<code>assert IsNot(x, y)</code>	<code>x is not y</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assert IsNotNone(x)</code>	<code>x is not None</code>
<code>assertIn(x, y)</code>	<code>x in y</code>
<code>assertNotIn(x, y)</code>	<code>x not in y</code>
<code>assertIsInstance(x, y)</code>	<code>isinstance(x, y)</code>
<code>assertNotIsInstance(x, y)</code>	<code>not isinstance(x, y)</code>

The next requirement of `Employee` will require that each Employee have an id associated with it.

- This id will be a generated value, as opposed to being passed as a parameter to the constructor.
  - ▶ The id should also be read-only such that its value once assigned cannot be changed.
- TDD dictates that the failing test is written first:

tdd/version05/employee\_test.py

```

1 import unittest
2 import datetime
3 import employee
4
5
6 class TestEmployee(unittest.TestCase):
7
8     def test_employee_object_creation(self):
9         emp = employee.Employee("Karen", "Jones", "Manager",
10                             datetime.date.today())
11
12    def test_get_employee_properties(self):
13        emp = employee.Employee("Karen", "Jones", "Manager",
14                             datetime.date.today())
15        self.assertEqual(emp.first_name, "Karen")
16        self.assertEqual(emp.last_name, "Jones")
17        self.assertEqual(emp.job, "Manager")
18        self.assertEqual(emp.hired, datetime.date.today())
19
20    def test_get_employee_id(self):
21        emp = employee.Employee("Karen", "Jones", "Manager",
22                             datetime.date.today())
23        self.assertTrue(hasattr(emp, "emp_id"))
24
25
26 if __name__ == "__main__":
27     unittest.main()

```

```

> python3 employee_test.py
.F.
=====
FAIL: test_get_employee_id (__main__.TestEmployee)
-----
Traceback (most recent call last):
  File "employee_test.py", line 22, in test_get_employee_id
    self.assertTrue(hasattr(emp, 'emp_id'))
AssertionError: False is not true
-----
Ran 3 tests in 0.001s
FAILED (failures=1)
>

```

The code necessary to pass the previous test is shown below.

*tdd/version06/employee.py*

```
1 class Employee:
2     id_generator = 1000 # Shared Class variable
3
4     def __init__(self, first_name, last_name, job, hired):
5         self.first_name = first_name
6         self.last_name = last_name
7         self.job = job
8         self.hired = hired
9         self.emp_id = Employee.id_generator
10        Employee.id_generator += 1
```

The second part of the requirement for the employee ID is that it should be a read-only property.

- The failing test makes use of “expected exceptions”, a block of code that should raise an exception in Python.
  - ▶ The test fails because the block of code in the context manager does not raise `AttributeError` as expected.

```
1 import unittest
2 import datetime
3 import employee
4
5
6 class TestEmployee(unittest.TestCase):
7
8     def test_employee_object_creation(self):
9         emp = employee.Employee("Karen", "Jones", "Manager",
10                             datetime.date.today())
11
12    def test_get_employee_properties(self):
13        emp = employee.Employee("Karen", "Jones", "Manager",
14                             datetime.date.today())
15        self.assertEqual(emp.first_name, "Karen")
16        self.assertEqual(emp.last_name, "Jones")
17        self.assertEqual(emp.job, "Manager")
18        self.assertEqual(emp.hired, datetime.date.today())
19
20    def test_get_employee_id(self):
21        emp = employee.Employee("Karen", "Jones", "Manager",
22                             datetime.date.today())
23        self.assertTrue(hasattr(emp, "emp_id"))
24
25    def test_cannot_change_employee_id(self):
26        emp = employee.Employee("Karen", "Jones", "Manager",
27                             datetime.date.today())
28        with self.assertRaises(AttributeError):
29            emp.emp_id = 99
30
31
32 if __name__ == "__main__":
33     unittest.main()
```

To fully pass all tests, the following code for Employee can be used:

*tdd/version08/employee.py*

```
1 class Employee:
2     id_generator = 1000 # Shared Class variable
3
4     def __init__(self, first_name, last_name, job, hired):
5         self.first_name = first_name
6         self.last_name = last_name
7         self.job = job
8         self.hired = hired
9         self._emp_id = Employee.id_generator
10        Employee.id_generator += 1
11
12    @property
13    def emp_id(self):
14        return self._emp_id
```

Finally, the tests themselves may be refactored (simplified in this case).

- Each test begins with creating the same object.

The `setUp()` method of the `unittest.TestCase` class provides a refactoring mechanism to factor out the duplicated set-up code into a single method that is automatically called by the framework prior to each test method being called.

- The `setUp(self)` method effectively replaces the need for the `test_employee_object_creation` method.

*tdd/version09/employee\_test.py*

```

1 import unittest
2 import datetime
3 import employee
4
5
6 class TestEmployee(unittest.TestCase):
7
8     def setUp(self):
9         self.emp = employee.Employee("Karen", "Jones", "Manager",
10                               datetime.date.today())
11        # print("Debug: setUp method being called")
12
13    def test_get_employee_properties(self):
14        self.assertEqual(self.emp.first_name, "Karen")
15        self.assertEqual(self.emp.last_name, "Jones")
16        self.assertEqual(self.emp.job, "Manager")
17        self.assertEqual(self.emp.hired, datetime.date.today())
18
19    def test_get_employee_id(self):
20        self.assertTrue(hasattr(self.emp, "emp_id"))
21
22    def test_cannot_change_employee_id(self):
23        with self.assertRaises(AttributeError):
24            self.emp.emp_id = 99
25
26
27 if __name__ == "__main__":
28     unittest.main()

```

---

## Exercises

### Exercise 1 (pres\_upper.py)

Read the file **presidents.txt**, creating a list of the presidents' last names.

- Then, use a list comprehension to make a copy of the list of names in upper case.
- Finally, loop through the list returned by the list comprehension and print out the names one per line.

### Exercise 2 (pres\_by\_birth.py)

Print out all the presidents first and last names, date of birth, and their political affiliations, sorted by date of birth.

- Read the **presidents.txt** file, putting the four fields into a list of tuples.
- Loop through the list, sorting by date of birth, and printing the information for each president.
- Use `sorted()` and a lambda function.

### Exercise 3 (pres\_gen.py)

Write a generator function to provide a sequence of the names of presidents (in "FirstName MiddleName LastName" format) from the **presidents.txt** file.

- They should be provided in the same order they are in the file.
- You should not read the entire file into memory, but one-at-a-time from the file.
- Then iterate over the the generator returned by your function and print the names.

# Chapter 4. Intermediate Classes

## Objectives

- Defining a class and its constructor
- Creating object methods
- Adding properties to a class
- Working with class data and methods
- Leveraging inheritance for code reuse
- Implementing special methods
- Knowing when NOT to use classes

## What is a class?

A class is a definition that represents a *thing*.

- The thing could be a file, a process, a database record, a strategy, a string, a person, or a truck.
  - ▶ When presented with requirements for a project, any noun is a possible candidate for a class.

The class describes both data and methods.

- The data represents the state of one instance of the class.
  - ▶ There can be both class data, which is shared by all instances, and instance data, which is only accessible from the instance.
- The methods are really special functions define the behavior of objects and how they process the data.
  - ▶ As with data, there can be both class methods, and instance methods.

Classes are a very powerful tool to organize code.

- However, there are some circumstances in Python where custom classes are not needed.
  - ▶ If the only thing needed is basic functionality, and there's no need to share or remember data, functions in a module might suffice.
  - ▶ If data is needed, but processing it is not necessary, a nested data structure built out of dictionaries, lists, and tuples, might suffice.

# Defining Classes

The `class` keyword is used to define a class with a given name.

- The PEP8 standard for class names uses what is often referred to as **CamelCase** or **Wordcaps** amongst others.
  - ▶ The first letter and each word in the name should be uppercase - and underscores are not used.
- More about the naming convention can be found in the PEP8 section at the following URL:

```
https://www.python.org/dev/peps/pep-0008/#class-names
```

The simplest form of class definition looks like this:

```
class SomeClassName:  
    pass
```

The contents of a class definition will typically consist of method definitions and shared data.

A class definition creates a new local namespace.

- All variable assignments go into this new namespace.
- All methods are called via the instance or the class name.

A list of base/parent classes may be specified in parentheses after the class name.

## Object Instances

An instance of a class is an object created from the class.

Each instance of an object that is created has its own id which uniquely defines that object among all of the other objects in memory at the time.

*lots\_of\_objects.py*

```

1 #!/usr/bin/env Python
2 class Something:
3     pass
4
5
6 class SomethingElse:
7     pass
8
9
10 class YetAnotherType:
11     pass
12
13
14 # Create a list of various instances of the above classes.
15 list_of_objects = [Something(), SomethingElse(), Something(), YetAnotherType(),
16                     SomethingElse()]
17
18 fmt = "{:15}{}"
19 for obj in list_of_objects:
20     print(fmt.format(type(obj).__name__, id(obj)))

```

The results of running the above examples are shown below:

```

$ python lots_of_objects.py
Something      140371246133360
SomethingElse  140371245729776
Something      140371246030368
YetAnotherType 140371245877856
SomethingElse  140371245878432
$
```

- Each object instance has its own values for the attributes defined in the class.
  - ▶ These attributes are usually created in the `__init__` method of the class.

# Instance attributes

An instance of a class (AKA object) normally contains methods and data.

Instance attributes are dynamic.

- They can be accessed directly from the object using "dot notation": obj.attribute.
- You can create, update, and delete attributes by directly accessing them if desired.

Attributes cannot be made private in Python.

- Names that begin with an underscore are understood by convention to be for internal use only.
  - ▶ Users of the class will not consider methods that begin with an underscore to be part of its API.
- Names that begin with a double underscore prefix causes the Python interpreter to rewrite the attribute name in order to avoid naming conflicts in subclasses.
  - ▶ This is also referred to as **name mangling**
  - ▶ The interpreter changes the name of the variable in a way that makes it harder to create collisions when the class is extended later.

The following examples defines a class named **Access** with various attributes.

- Some of the attributes are methods: `public_method`, `_private_method`, `__mangled_method`.
- Some of the attributes are data: `a`, `_b`, `__c`.

`access.py`

```

1 #!/usr/bin/env python
2 class Access():
3     def public_method(self):
4         print("assigning 5 to attribute a")
5         self.a = 5
6
7     def _private_method(self):    # private!
8         print("assigning 25 to attribute _b")
9         self._b = 25
10
11    def __mangled_method(self):
12        print("assigning 99 to attribute __c")
13        self.__c = 99

```

The following example demonstrates the various ways of interacting with instances of the **Access** class.

```
1 #!/usr/bin/env Python
2 from access import Access
3
4
5 a = Access()
6 # accessing both the 'public_method' attribute
7 # and the instance data attribute 'a'
8 a.public_method()
9 print(a.a)
10 print()
11
12 # accessing both the '_private_method' attribute
13 # and the instance data '_b' attribute
14 a._private_method()      # legal, but is intended to be private
15 print(a._b)              # legal, but is also intended to be private
16 print()
17
18 # attempt to access the '__mangled_method' attribute
19 try:
20     a.__mangled_method()    # does not exist under this name
21 except AttributeError as ae:
22     print(ae)
23 print()
24
25 # knowing how it is mangled though still provides access
26 a._Access__mangled_method()
27 print(a._Access__c)
28 print()
```

# Instance Methods

An instance method is a function defined in a class.

- When a method is called from an object, the object is passed in as the implicit first parameter, named **self** by strong convention.

*rabbit.py*

```

1 #!/usr/bin/env python
2 class Rabbit:
3
4     # self is explicit reference to object implicitly passed by constructor
5     def __init__(self, size, danger):
6         self._size = size
7         self._danger = danger
8
9     # self is explicit reference to object method is called on that is
10    # implicitly passed as the first parameter to the method
11    def threaten(self):
12        print("I am a {} bunny with {}".format(self._size, self._danger))
13
14
15 # Explicitly passing two arguments to constructor that takes 3 parameters
16 # The first parameter to constructor is implicitly passed a reference to 'r1'
17 r1 = Rabbit('large', "sharp, pointy teeth")
18
19 # reference to 'r1' is implicitly passes as param to instance method
20 r1.threaten()
21
22 r2 = Rabbit('small', 'fluffy fur')
23 r2.threaten()
```

## Constructors

If a class defines a method named `__init__`, it will be automatically called when an object instance is created. This is the **constructor**.

The object being created is implicitly passed as the first parameter to `__init__`.

- This parameter is named **self** by very strong convention.
  - ▶ Data attributes can be assigned to **self**.
  - ▶ These attributes can then be accessed by other methods.

This can be seen in the `__init__` method of the `Rabbit` class in the previous example.

- In C++, Java, and C#, **self** would be called **this**.

# Getters and Setters

Getter and setter methods can be used to access an object's data.

- These are traditional in object-oriented programming.

A **getter** retrieves data (private variable) from **self**. A **setter** assigns a value to a variable in **self**.

- Most Python developers use **properties**, described next, instead of getters and setters.

`get_set.py`

```
1 #!/usr/bin/env python
2 class Knight():
3     def __init__(self, name):
4         self._name = name
5
6     def set_name(self, name):
7         self._name = name
8
9     def get_name(self):
10        return self._name
11
12
13 if __name__ == '__main__':
14     k = Knight("Lancelot")
15     print(k.get_name())
```

## Properties

While object attributes can be accessed directly, in many cases the class needs to exercise some control over the attributes.

A more elegant approach is to use properties.

- A property is a kind of managed attribute.

Properties are accessed directly, like normal attributes (variables).

- The getter, setter, and deleter methods are registered as callbacks with the `property` object.
- They are then implicitly called, so that the class can control what values are stored or retrieved from the attributes.

To create a `property`, the first step is to apply the `@property` decorator to a method with the name you want.

- It receives no parameters other than `self`.
- And returns a value of a private attribute of the class.

To create the setter for the `property`, create another method with the property name.

- Yes, there will be two method definitions with the same name.
- Decorate this method with the property name plus ".setter".
  - ▶ In other words, if the property is named "spam", the decorator will be "@spam.setter".
- The setter method will take one parameter (other than self).
  - ▶ The parameter is assigned to a the same private attribute of the object the getter returns.
- It is common for a setter property to raise an error if the value being assigned is invalid.

A deleter method is seldom required to be registered with the property.

- Creating it is the same as for a setter property, but use "`@propertynname.deleter`".

*knight.py*

```
1 #!/usr/bin/env python
2
3
4 class Knight():
5     def __init__(self, name, title, color):
6         # store values in private instance variables
7         self._name = name
8         self._title = title
9         self._color = color
10
11    # decorator creates a property object named 'name'
12    # and registers the method it decorates as getter method of the property
13    @property
14    def name(self):          # public property wraps access to private data
15        return self._name
16
17    # decorator creates a property object named 'title'
18    # and registers the method it decorates as getter method of the property
19    @property
20    def title(self):          # public property wraps access to private data
21        return self._title
22
23    # decorator creates a property object named 'color'
24    # and registers the method it decorates as getter method of the property
25    @property
26    def color(self):          # public property wraps access to private data
27        return self._color
28
29    # decorator references property object named 'color' defined above
30    # and registers the method it decorates as setter method of the property
31    @color.setter
32    def color(self, color):   # public property wraps access to private data
33        self._color = color
```

An application that uses the above Knight class is shown on the following page.

```
1 #!/usr/bin/env python
2 from knight import Knight
3
4
5 def main():
6     k = Knight("Lancelot", "Sir", 'blue')
7
8     # Bridgekeeper's question
9     print(f'Sir {k.name}, what is your...favorite color?') # property getter
10
11    # Knight's answer
12    print(f'red, no -- {k.color}!') # property getter
13
14    k.color = 'red' # property setter
15
16    print(f'color is now:{k.color}') # property getter
17
18
19 if __name__ == '__main__':
20     main()
```

## Class Data

Data can be attached to the class itself, and shared among all instances.

- Class data can be accessed via the class name from inside or outside the class.

Any class attribute not overwritten by an instance attribute is also available through the instance.

- Though it more standard to always access it through the class.

*class\_data.py*

```
1 #!/usr/bin/env python
2
3
4 class Rabbit:
5     LOCATION = "the Cave of Caerbannog" # class data
6
7     def __init__(self, weapon):
8         self.weapon = weapon
9
10    def display(self):
11        fmt = "This rabbit guarding {} uses {} as a weapon"
12
13        # access class data Location from class Rabbit
14        print(fmt.format(Rabbit.LOCATION, self.weapon))
15
16        # access class data Location from class Rabbit
17        # print(fmt.format(self.LOCATION, self.weapon))
18
19
20 r1 = Rabbit("a nice cup of tea")
21 r1.display()
22
23 r1 = Rabbit("big pointy teeth")
24 r1.display()
```

## Class Methods

If a method only needs class attributes, it can be made a class method via the `@classmethod` decorator.

- This alters the method so that it gets a copy of the class object rather than the instance object.
  - ▶ This is true whether the method is called from the class or from an instance.

The parameter to a class method is named `cls` by strong convention.

`class_methods_and_data.py`

```

1 #!/usr/bin/env python
2
3
4 class Rabbit:
5     # Class data - not duplicated in instances
6     LOCATION = "the Cave of Caerbannog"
7
8     def __init__(self, weapon):
9         self.weapon = weapon
10
11    # instance method
12    def display(self):
13        print("This rabbit guarding {} uses {} as a weapon".
14              format(Rabbit.LOCATION, self.weapon))
15
16    # The @classmethod decorator makes the function receive the class object,
17    # not the instance object
18    @classmethod
19    def get_location(cls):    # class method
20        return cls.LOCATION  # Access class data via cls variable
21
22
23 r = Rabbit("a nice cup of tea")
24 print(Rabbit.get_location())  # Call class method from the class
25 print(r.get_location())      # Call class method from the instance

```

# Inheritance

Any language that supports classes supports **inheritance**.

One or more base/parent classes may be specified as part of the class definition.

- All of the previous examples in this chapter have used the default base class, **object**.

The base class must already be imported, if necessary.

If a requested attribute is not found in the class, the search looks in the base class.

- This rule is applied recursively if the base class itself is derived from some other class.
  - ▶ For instance, all classes inherit the implementation from **object**, unless a class explicitly implements it.

Classes may override methods of their base classes.

While a base class method can be accessed by explicitly using the name directly, it is recommended that it be accessed via a call to **super()**:

- Direct access:
  - ▶ `BaseClassName.method_name(self, arguments)`.
- Access via **super()**:
  - ▶ `super().method_name(arguments)`.

The **super()** function can be used in a class to invoke methods in base classes.

- It searches the base classes and their bases, recursively, from left to right until the method is found.
- The advantage of **super()** is that you don't have to specify the base class explicitly, so if you change the base class, it automatically does the right thing.

For classes that have a single inheritance tree, this works great.

- For classes that have a diamond-shaped tree, **super()** may not do what you expect.
  - ▶ In this case, using the explicit base class name is best.

```
class Foo(Bar):
    def __init__(self):
        super().__init__() # same as Bar.__init__(self)
```

The following `Animal` class will act as a base class in a hierarchy of subclasses.

*animal.py*

```
1 #!/usr/bin/env python
2
3
4 class Animal:
5     count = 0 # defines count as Class data
6
7     def __init__(self, species, name, sound):
8         self._species = species
9         self._name = name
10        self._sound = sound
11        Animal.count += 1
12
13    @property
14    def species(self):
15        return self._species
16
17    @property
18    def name(self):
19        return self._name
20
21    def make_sound(self):
22        return self._sound
23
24    @classmethod      # implicitly passes Class object to remove is called on
25    def remove(cls):  # explicitly accepts Class object as cls
26        cls.count -= 1 # update class data from Class referenced by cls
27
28    @classmethod
29    def zoo_size(cls):
30        return cls.count
```

An example application that uses the `Animal` class is shown on the next page.

*creating\_animals.py*

```

1 #!/usr/bin/env python
2 from animal import Animal
3
4
5 def main():
6     animals = [Animal("African lion", "Leo", "Roarrrrrrr"),
7                 Animal("cat", "Garfield", "Meowwww"),
8                 Animal("cat", "Felix", "Meowwww")]
9
10    for animal in animals:
11        print(animal.name, "is a", animal.species, "--", animal.make_sound())
12
13
14 if __name__ == "__main__":
15     main()

```

The next example defines an `Insect` class that extends the `Animal` base class.

*insect.py*

```

1 #!/usr/bin/env python
2
3 from animal import Animal
4
5
6 class Insect(Animal):
7     """
8         An animal with 2 sets of wings and 3 pairs of legs
9     """
10    # constructor (AKA initializer)
11    def __init__(self, species, name, sound, can_fly=True):
12        # pass data shared between parent and child to parent's __init__()
13        super().__init__(species, name, sound)
14        # store what is specific to child class that makes it a special Animal
15        self._can_fly = can_fly
16        self._sets_of_wings = 2
17        self._pairs_of_legs = 3
18
19    @property
20    def can_fly(self): # decorated as property getter
21        return self._can_fly

```

The following application uses the Insect class defined in the previous example.

*insects.py*

```
1 #!/usr/bin/env python
2
3 from insect import Insect
4
5
6 if __name__ == '__main__':
7     monarch = Insect('monarch butterfly', 'Mary', None)
8     scarab = Insect('scarab beetle', 'Rupert', 'Bzzz', False)
9
10    for insect in monarch, scarab:
11        # can_fly is inherited from Insect
12        flying_status = "can" if insect.can_fly else "can't"
13
14        fmt = "Hi! I am {} the {} and I {} fly!  {}"
15        # name, species, and make_sound are all inherited from Animal
16        print(fmt.format(insect.name, insect.species, flying_status,
17                          insect.make_sound()))
```

# Multiple Inheritance

Python classes can inherit from more than one base class.

- This is called "multiple inheritance".
  - Classes designed to be added to a base class are sometimes called "mixin classes", or just "mixins".

Methods are searched for in the first base class, then its parents, then the second base class and parents, and so forth.

- Put the "extra" classes before the main base class, so any methods in those classes will override methods with the same name in the base class.

*multiple\_inheritance.py*

```

1 #!/usr/bin/env python
2 class AnimalBase: # define the primary base/parent class
3     def __init__(self, name):
4         self._name = name
5
6     def get_id(self):
7         return self._name
8
9
10 class CanBark: # define additional (mixin) base/parent class
11     def bark(self):
12         return "woof-woof"
13
14
15 class CanFly: # define additional (mixin) base/parent class
16     def fly(self):
17         return "I'm flying"
```

The example below demonstrates the use of several classes that rely on multiple inheritance.

*dogs\_and\_sparrows.py*

```
1 #!/usr/bin/env python
2 from multiple_inheritance import AnimalBase, CanBark, CanFly
3
4
5 class Dog(CanBark, AnimalBase): # inherit from mixin and primary base class
6     pass
7
8
9 class Sparrow(CanFly, AnimalBase): # inherit from mixin and primary base class
10    pass
11
12
13 def main():
14     d = Dog('Dennis')
15     print(d.get_id()) # Dog inherits get_id() from AnimalBase
16     print(d.bark())   # Dog inherits bark() from CanBark mixin
17     print()
18
19     s = Sparrow('Steve')
20     print(s.get_id()) # Sparrow inherits get_id() from AnimalBase
21     print(s.fly())   # Sparrow inherits fly() from CanFly mixin
22     print()
23
24     print("Sparrow mro:", Sparrow.mro())
25     print()
26     print("Dog mro:", Dog.mro())
27
28
29 if __name__ == '__main__':
30     main()
```

The exact method resolution order (MRO) for a class can be determined by calling the class's `mro()` method as seen in the above script.

# Abstract base classes

The `abc` module provides **abstract base classes**.

When a method in an abstract class is designated **abstract**, it must be implemented in any derived class.

- If a method is not marked abstract, it may be inherited and used as is, overwritten, or extended.

To create an abstract class, import `ABCMeta` and `abstractmethod` from the `abc` module.

- Create the base (abstract) class normally, but assign `ABCMeta` to the class option `metaclass`.
  - ▶ An alternative would be to simple extend the helper `ABC` class that already takes care of `metaclass=ABCMeta`
- Then decorate any desired abstract methods with `@abstractmethod`.
  - ▶ Now, any classes that inherit from the base class must implement any abstract methods.

Non-abstract methods do not have to be implemented, as they are inherited.

*abstract\_animal.py*

```

1 from abc import ABCMeta, abstractmethod
2
3
4 # metaclasses control how classes are created;
5 # metaclass=ABCMeta adds restrictions to classes that inherit from Animal
6 class Animal(metaclass=ABCMeta):
7     @abstractmethod                      # decorates speak as an abstract method
8     def speak(self):
9         pass
10
11 # Often easier to write using the ABC helper class as shown below:
12
13 # from abc import ABC, abstractmethod
14 #
15 # class Animal(ABC):
16 #     @abstractmethod
17 #     def speak(self):
18 #         pass

```

Following are several subclasses of `Animal`.

`animal_subclasses.py`

```
1 from abstract_animal import Animal
2
3
4 class Dog(Animal):           # Inherit from abstract base class Animal
5     def speak(self):          # speak() *must* be implemented
6         print("woof! woof!")
7
8
9 class Cat(Animal):           # Inherit from abstract base class Animal
10    def speak(self):          # speak() *must* be implemented
11        print("Meow meow meow")
12
13
14 class Duck(Animal):          # Inherit from abstract base class Animal
15     pass                      # does NOT implement speak()
16
17
18 d = Dog()
19 d.speak()
20
21 c = Cat()
22 c.speak()
23
24 try:
25     d = Duck()   # raises TypeError as it does not meet requirements of Animal
26     d.speak()
27 except TypeError as err:
28     print(err)
```

## Special Methods

Python has a set of **special methods** that can be used to make user-defined classes emulate the behavior of builtin classes.

- These methods can be used to define the behavior for builtin functions such as `str()`, `len()` and `repr()` among others.
- They can also be used to override many Python operators, such as `+`, `*`, and `==`.

These methods expect the `self` parameter, like all instance methods. \* `self` Is the object being called from the builtin function, or the left operand of a binary operator such as `==`.

- They frequently take one or more additional parameters.

For instance, if your object represented a database connection, you could have `str()` return the hostname, port, and maybe the connection string.

The default for `str()` is to call `repr()`, which returns something like `<main.DBConn object at 0xb7828c6c>`, which is not nearly so user-friendly.

A complete list of special method names and their purpose can be found in the documentation at the following RUL:

<http://docs.python.org/reference/datamodel.html#special-method-names> for detailed documentation on the special methods.

Table 4. Special Methods and Variables

Method or Variables	Description
<code>__new__(cls, ...)</code>	Returns new object instance; Called before <code>__init__()</code>
<code>__init__(self, ...)</code>	Object initializer (constructor)
<code>__del__(self)</code>	Called when object is about to be destroyed
<code>__repr__(self)</code>	Called by <code>repr()</code> builtin
<code>__str__(self)</code>	Called by <code>str()</code> builtin
<code>__eq__(self, other)</code> <code>__ne__(self, other)</code> <code>__gt__(self, other)</code> <code>__lt__(self, other)</code> <code>__ge__(self, other)</code> <code>__le__(self, other)</code>	Implement comparison operators ==, !=, >, <, >=, and <=. <code>self</code> is object on the left.
<code>__hash__(self)</code>	Called by <code>hash()</code> builtin, also used by dict, set, and frozenset operations
<code>__bool__(self)</code>	Called by <code>bool()</code> builtin. Implements truth value (boolean) testing. If not present, <code>bool()</code> uses <code>len()</code>
<code>__unicode__(self)</code>	Called by <code>unicode()</code> builtin
<code>__getattr__(self, name)</code> <code>__setattr__(self, name, value)</code> <code>__delattr__(self, name)</code>	Override normal fetch, store, and deleter
<code>__getattribute__(self, name)</code>	Implement attribute access for new-style classes
<code>__get__(self, instance)</code>	<code>__set__(self, instance, value)</code>
<code>__del__(self, instance)</code>	Implement descriptors
<code>__slots__ = variable-list</code>	Allocate space for a fixed number of attributes.
<code>__metaclass__ = callable</code>	Called instead of <code>type()</code> when class is created.
<code>__instancecheck__(self, instance)</code>	Return true if instance is an instance of class
<code>__subclasscheck__(self, instance)</code>	Return true if instance is a subclass of class
<code>__call__(self, ...)</code>	Called when instance is called as a function.
<code>__len__(self)</code>	Called by <code>len()</code> builtin
<code>__getitem__(self, key)</code>	Implements <code>self[key]</code>
<code>__setitem__(self, key, value)</code>	Implements <code>self[key] = value</code>
<code>__delitem__(self, key)</code>	Implements <code>del self[key]</code>
<code>__iter__(self)</code>	Called when <code>iter()</code> is applied to container

Method or Variables	Description
<code>__next__(self)</code>	Called when <code>next()</code> is applied to iterator
<code>__reversed__(self)</code>	Called by <code>reversed()</code> builtin
<code>__contains__(self, object)</code>	Implements <code>in</code> operator
<code>__add__(self, other)</code> <code>__sub__(self, other)</code> <code>__mul__(self, other)</code> <code>__floordiv__(self, other)</code> <code>__mod__(self, other)</code> <code>__divmod__(self, other)</code> <code>__pow__(self, other[, modulo])</code> <code>__lshift__(self, other)</code> <code>__rshift__(self, other)</code> <code>__and__(self, other)</code> <code>__xor__(self, other)</code> <code>__or__(self, other)</code>	Implement binary arithmetic operators <code>+</code> , <code>-</code> , <code>*</code> , <code>//</code> , <code>%</code> , <code>**</code> , <code>&lt;&lt;</code> , <code>&gt;&gt;</code> , <code>&amp;</code> , <code>^</code> , and <code> </code> . Self is object on left side of expression.
<code>__matmul__(self, other)</code>	Implement the <code>@</code> binary operator
<code>__div__(self,other)</code> <code>__truediv__(self,other)</code>	Implement binary division operator <code>/</code> . <code>__truediv__</code> is called if <code>__future__.division</code> is in effect.
<code>__radd__(self, other)</code> <code>__rsub__(self, other)</code> <code>__rmul__(self, other)</code> <code>__rdiv__(self, other)</code> <code>__rtruediv__(self, other)</code> <code>__rfloordiv__(self, other)</code> <code>__rmod__(self, other)</code> <code>__rdivmod__(self, other)</code> <code>__rpow__(self, other)</code> <code>__rlshift__(self, other)</code> <code>__rrshift__(self, other)</code> <code>__rand__(self, other)</code> <code>__rxor__(self, other)</code> <code>__ror__(self, other)</code>	Implement binary arithmetic operators with swapped operands. (Used if left operand does not support the corresponding operation)

Method or Variables	Description
<code>__iadd__(self, other)</code> <code>__isub__(self, other)</code> <code>__imul__(self, other)</code> <code>__idiv__(self, other)</code> <code>__itruediv__(self, other)</code> <code>__ifloordiv__(self, other)</code> <code>__imod__(self, other)</code> <code>__ipow__(self, other[, modulo])</code> <code>__ilshift__(self, other)</code> <code>__irshift__(self, other)</code> <code>__iand__(self, other)</code> <code>__ixor__(self, other)</code> <code>__ior__(self, other)</code>	Implement augmented (+=, -=, etc.) arithmetic operators
<code>__neg__(self)</code> <code>__pos__(self)</code> <code>__abs__(self)</code> <code>__invert__(self)</code>	Implement unary arithmetic operators -, +, abs(), and ~
<code>__oct__(self)</code> <code>__hex__(self)</code>	Implement oct() and hex() builtins
<code>__index__(self)</code>	Implement operator.index()
<code>__coerce__(self, other)</code>	Implement "mixed-mode" numeric arithmetic.

## specialmethods.py

```

1 #!/usr/bin/env python
2
3
4 class Special():
5
6     def __init__(self, value): # invoked every time Special() is called
7         self._value = str(value)
8
9     def __add__(self, other): # called when the '+' operator is used
10        return self._value + other._value
11
12    def __mul__(self, num): # called when the '*' operator is used
13        return ''.join((self._value for i in range(num)))
14
15    def __str__(self): # called when the str() constructor is called is used
16        return self._value.upper()
17
18    def __eq__(self, other): # called when the '==' operator is used
19        return self._value == other._value
20
21
22 if __name__ == '__main__':
23     s = Special('spam')      # invokes Special.__init__
24     t = Special('eggs')
25     u = Special('spam')
26     v = Special(5)          # invokes Special.__init__
27     w = Special(22)
28
29     print("s + s", s + s)    # invokes Special.__add__(s, s)
30     print("s + t", s + t)    # invokes Special.__add__(s, t)
31     print("t + t", t + t)
32     print("s * 10", s * 10)  # invokes Special.__mult__(s, 10)
33     print("t * 3", t * 3)
34
35     # invokes Special.__str__(s) and special.__str__(t)
36     print("str(s)={} str(t)={}".format(str(s), str(t)))
37
38     print("id(s)={} id(t)={} id(u)={}".format(id(s), id(t), id(u)))
39     print("s == s", s == s)   # invokes Special.__eq__(s, s)
40     print("s == t", s == t)   # invokes Special.__eq__(s, t)
41     print("s == u", s == u)
42     print("v + v", v + v)
43     print("v + w", v + w)
44     print("w + w", w + w)
45     print("v * 10", v * 10)
46     print("w * 3", w * 3)

```

## Static Methods

- Related to class, but doesn't need instance or class object
- Use `@staticmethod` decorator

A (static method) is a utility method that is related to the class, but does not need the instance or class object. Thus, it has no automatic parameter.

One use case for static methods is to factor some kind of logic out of several methods, when the logic doesn't require any of the data in the class.

**NOTE** Static methods are seldom needed.

# Exercises

## Exercise 1 (president.py, president\_main.py)

Create a module that implements a **President** class.

- This class has a constructor that takes the index number of the president (1-45) and creates an object containing the associated information from the **presidents.txt** file.
- Provide the following properties (types indicated after **->**):

```
term_number -> int
first_name -> string
last_name -> string
birth_date -> date object
death_date -> date object (or None, if still alive)
birth_place -> string
birth_state -> string
term_start_date -> date object
term_end_date -> date object (or None, if still in office)
party -> string
```

Write a main script to exercise some or all of the properties. It could look something like

```
from president import President

p = President(1) # George Washington
fmt = "{0} was born at {1}, {2} on {3}"
print(fmt.format(p.first_name, p.birth_place, p.birth_state, p.birth_date))
```



# Chapter 5. Idiomatic Data Handling

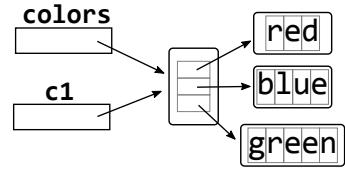
## Objectives

- Distinguish between deep and shallow copying
- Set default dictionary values
- Count items with the `Counter` object
- Enumerate possible allowed values with `Enum`
- Define named tuples
- Use Dataclasses
- Prettyprint data structures
- Create and extract from compressed archives
- Save Python structures to the hard drive

## Deep vs shallow copying

Consider the following code:

```
colors = ['red', 'blue', 'green']
c1 = colors
```

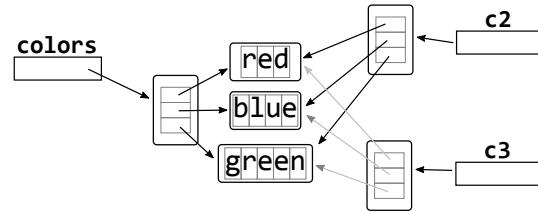


The assignment to variable `c1` does not create a new object, as shown in the diagram.

- It is another name that is *bound* to the same list object as the variable `colors`.

To create a new object, you can either use the list constructor `list()`, or use a slice which contains all elements:

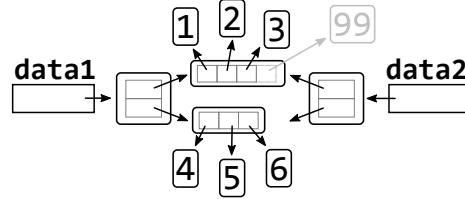
```
c2 = list(colors)
c3 = colors[:]
```



- In both cases, `c2` and `c3` are each distinct objects.
- However, the elements of `c2` and `c3` are not copied.
  - ▶ The variables are still bound to the same objects as the elements of `colors`, as shown in the diagram above.

Another example is shown below.

```
data1 = [ [1, 2, 3], [4, 5, 6] ]
data2 = list(data1)
data2[0].append(99)
print(data1)
```



The output of the above program will show that the first element of `data1` contains the value 99.

- This is because `data1[0]` and `data2[0]` are both bound to the same `list` object.

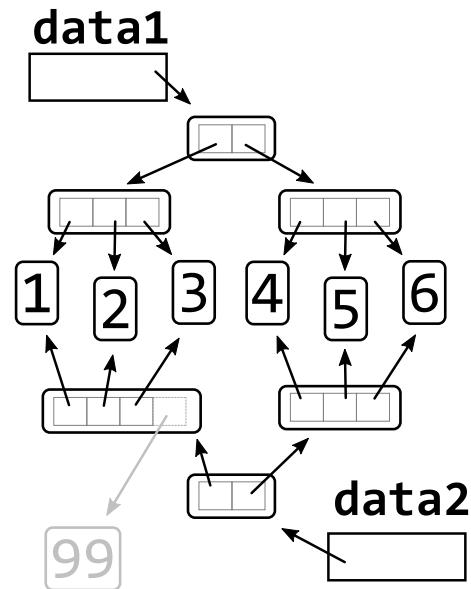
To do a "deep" (recursive) copy, use the `deepcopy` function in the `copy` module:

`deep_copy.py`

```

1 #!/usr/bin/env python
2 import copy
3
4 data1 = [[1, 2, 3], [4, 5, 6]]
5 data2 = copy.deepcopy(data1)
6
7 data2[0].append(99)
8
9 # Outer list is still a unique copy
10 print("data1:", "id:", id(data1), data1)
11 print("data2:", "id:", id(data2), data2)
12 print()
13
14 # Nested lists are now a unique copy also
15 print("Nested list ids of data1")
16 print("\tid of data1[0]: ", id(data1[0]))
17 print("\tid of data1[1]: ", id(data1[1]))
18 print()
19 print("Nested list ids of data2")
20 print("\tid of data2[0]: ", id(data2[0]))
21 print("\tid of data2[1]: ", id(data2[1]))

```



The output of the above script is shown below.

```

$ python deep_copy.py
data1: id: 140050574231040 [[1, 2, 3], [4, 5, 6]]
data2: id: 140050573582272 [[1, 2, 3, 99], [4, 5, 6]]

Nested list ids of data1
    id of data1[0]: 140050573432896
    id of data1[1]: 140050573360832

Nested list ids of data2
    id of data2[0]: 140050573582016
    id of data2[1]: 140050573611648
$ 

```

## Using OrderedDict

Prior to Python 3.6, the order of elements in a dictionary was undefined.

- This could create issues when converting dictionaries to JSON, XML, and other formats, where you want to create fields in a specific order.

As of Python 3.6 (unofficially) and 3.7 (officially), all dictionaries maintain the keys in insertion order.

## Default dictionary values

Normally, when you use an invalid key with a dictionary, it raises a `KeyError`.

- While this could be avoided by using the `get()` method of a dictionary, a `defaultdict` might be a better choice.

The `defaultdict` class in the `collections` module allows you to provide a default value.

- This ensures there will never be a `KeyError` raised.

The constructor for a `defaultdict` takes a reference to a function and uses its return value for the default value.

- A lambda function can also be used in place of a reference to an existing function.

`defaultdict_ex.py`

```

1 #!/usr/bin/env python
2
3 from collections import defaultdict
4
5 dd = defaultdict(lambda: 0) # create defaultdict with function that returns 0
6
7 dd['spam'] = 10 # assign some values to the dict
8 dd['eggs'] = 22
9
10 print(dd['spam'], dd['eggs'])
11 print(dd['foo']) # missing key 'foo' invokes function and returns 0

```

# Counting with Counter

For ease in counting, the `collections` module provides a `Counter` class.

- This is essentially a `defaultdict` whose default value is zero.
  - ▶ This makes it simple to increment the value for any key, whether it's been seen before or not.

`count_with_counter.py`

```
1 #!/usr/bin/env python
2
3 from collections import Counter
4
5 counts = Counter() # create Counter object (defaults to 0 for missing keys)
6
7 with open("../data/breakfast.txt") as breakfast_in:
8     for line in breakfast_in:
9         item = line.rstrip() # remove EOL from line
10        counts[item] += 1    # increment count for current item
11
12 for item, count in counts.items(): # iterate over results
13     print(item, count)
```

## The array module

The `array` module provides an array class which implements an efficient numeric array where each element is the same type.

- There are a number of different numeric types that can be used.
  - ▶ The actual representation of values is determined by the machine architecture.

The actual size can be accessed through the `itemsize` attribute.

This is efficient when you have a large number of numeric values to store.

- It can take much less space than a normal `list` object, and is faster.

The numeric type flags are nearly the same as those used by the `struct` module.

`array_examples.py`

```

1 #!/usr/bin/env python
2 from sys import getsizeof
3 from array import array
4 from random import randint
5
6 values = [randint(1, 30000) for i in range(1000)] # Create list of 1000 ints
7
8 print(f'Size of integer list: {getsizeof(values)}\n')
9
10 fmt = "[{}, {}, {}, {}, {}, ...]"
11 for datatype in 'i', 'h', 'L', 'Q', 'd':
12     data_array = array(datatype, values) # Create array of specified type
13     print(f'Size of {datatype} array: {getsizeof(data_array):,} Contents:',
14           fmt.format(*data_array[:5])) # Note memory usage

```

```

$ python array_examples.py
Size of integer list: 9016

Size of i array: 4,064  Contents: [18004, 20333, 9295, 27367, 15173, ...]
Size of h array: 2,064  Contents: [18004, 20333, 9295, 27367, 15173, ...]
Size of L array: 8,064  Contents: [18004, 20333, 9295, 27367, 15173, ...]
Size of Q array: 8,064  Contents: [18004, 20333, 9295, 27367, 15173, ...]
Size of d array: 8,064  Contents: [18004.0, 20333.0, 9295.0, 27367.0, 15173.0, ...]

```

Table 5. array object type codes

Format	C Type	Python Type	Standard size (bytes)
b	signed char	integer	1
B	unsigned char	integer	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer	4
l	long	integer	4
L	unsigned long	integer	4
q	long long	integer	8
Q	unsigned long long	integer	8
f	float	float	4
d	double	float	8

**NOTE**

The 'q' and 'Q' type codes are available only if the platform C compiler used to build Python supports C long long or, on Windows, supports, \_\_int64.

## Enumerated Types

In many cases, it is beneficial to have an enumerated set of values that cannot change.

- Consider: possible colors on a display, days of the week, months of the year, etc.

The improper way to do this would be creating variables of the possibilities.

```
# BAD IDEA
MONDAY = 1
TUESDAY = 2
WEDNESDAY = 3
THURSDAY = 4
FRIDAY = 5
SATURDAY = 6
SUNDAY = 7
```

- There is no dynamic way to iterate over the possible entries.
- There is no way to extract the name of the enumerated value.
- Not only are the symbols variables (and can be reassigned to), but the values lack any context or type, leading to confusing results.

```
if day == 4: # Valid and will work, but confusing and should be disallowed
    pass
```

The `enum` module allows the declaration of enumerated values through the `Enum` data type.

*month\_enum.py*

```

1 #!/usr/bin/env python
2 from enum import Enum
3
4
5 class Month(Enum):
6     January = (1, 31)
7     February = (2, 28)
8     March = (3, 31)
9     April = (4, 30)
10    May = (5, 31)
11    June = (6, 30)
12    July = (7, 31)
13    August = (8, 31)
14    September = (9, 30)
15    October = (10, 31)
16    November = (11, 30)
17    December = (12, 31)
18
19    @property
20    def short_name(self):
21        return self.name[:3].title()
22
23    @property
24    def days(self):
25        return self.value[1]

```

These symbols are now unique enumerated values of type `Month`.

- They will not compare as equal to any value but themselves.
- They may not be modified or deleted, making them true constant values in Python.
  - ▶ While the symbol is constant, the value associated with it may be mutable such as a list or any other mutable type.

Python allows the enumerated type to have any properties and methods desired.

- A default property provided is `name`, which will produce a string of the instance.
- Another property is `value`, in case the value of the enumerated type is significant.
  - ▶ The value can be auto generated with the `auto()` function, or manually assigned.
  - ▶ By default, integer values are used by the `auto()` function

An `Enum` is iterable so its members can be iterated over using a for loop.

- This iteration order follows the natural sort order of the `value` of each element in the `Enum`.

*months.py*

```

1 #!/usr/bin/env python
2 from month_enum import Month
3
4 feb = Month.February
5
6 print(feb.name, "can be shortened to", feb.short_name,
7       f"and has {feb.value[1]} days in it")
8
9 for month in Month:
10    print(f'{month:^20}', end="")
11    if month.value[0] % 4 == 0:
12        print()

```

```

$ python months.py
February can be shortened to Feb and has 28 day(s) in it
Month.January      Month.February      Month.March      Month.April
Month.May          Month.June         Month.July       Month.August
Month.September    Month.October      Month.November   Month.December

```

The `Enum` type is a distinct type, and will compare as not-equal to other enumerated types, even if the underlying value is the same.

The next example uses the `auto()` function to automatically assign values to each member and compares two `Enum` objects that have the same value.

## comparing\_enums.py

```

1 #!/usr/bin/env python
2 from enum import Enum, auto
3
4
5 class Color(Enum):
6     WHITE = auto()
7     RED = auto()
8     GREEN = auto()
9     BLUE = auto()
10    ORANGE = auto()
11
12
13 class Direction(Enum):
14    NORTH = auto()
15    EAST = auto()
16    SOUTH = auto()
17    WEST = auto()
18
19
20 def main():
21     fmt = "{:>6}:{:<10}"
22     for color in Color:
23         print(fmt.format(color.name, color.value), end="")
24     print()
25     for direction in Direction:
26         print(fmt.format(direction.name, direction.value), end="")
27     print('\n')
28
29     # Compare different Enum's that have the same ValueError
30     print("Equal" if Color.WHITE == Direction.NORTH else "Not Equal")
31
32
33 if __name__ == '__main__':
34     main()

```

```

$ python comparing_enums.py
WHITE:1          RED:2          GREEN:3          BLUE:4          ORANGE:5
NORTH:1          EAST:2         SOUTH:3         WEST:4

Not Equal
$
```

## Dataclasses

Dataclasses drastically simplify the definition of classes.

- Like an Enum, each class decorated with `@dataclass` may define its own methods.
- Each field in the class is declared at the top-level of the class, along with a type annotation.
  - ▶ The fields may be assigned default values.
  - ▶ Type annotations or type hinting will be covered in more detail in a coming chapter.
- The `__init__` method will automatically take each field as its arguments.

*basic\_fraction.py*

```

1 from dataclasses import dataclass
2
3
4 @dataclass
5 class Fraction:
6     num: int
7     denom: int
8
9
10 f1 = Fraction(1, 2)
11 f2 = Fraction(3, 4)
```

Rather than re-implementing the `__init__` method, it is possible to have a **post-init function** called `__post_init__`.

Rather than manually defining `__init__`, `__repr__`, `__eq__`, `__lt__`, `__hash__`, and other such dunder methods, using the `@dataclass` decorator will generate them automatically.

```
@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False)
```

- `init` - When true, generate an `__init__` method
- `repr` - When true, generate a `__repr__` method
- `eq` - When true, generate an `__eq__` method
- `order` - When true, generate `__lt__`, `__le__`, etc. methods
- `frozen` - When true, make instances immutable
- `unsafe_hash` - When true, generate `__hash__`, even if it is a bad idea

The `unsafe_hash` parameter needs a bit of explanation.

- If the `eq` and `frozen` parameters are true, then the `@dataclass` will automatically generate a `__hash__` method.
  - ▶ This will allow instances of the class may be used as keys safely.
- If `frozen` is false, or no `__eq__` is defined, then the `__hash__` method could be dangerous to use, and thus is not generated.
  - ▶ Specifying `unsafe_hash=True` forces generation of the `__hash__` method.

```
from dataclasses import dataclass

@dataclass
class Fraction:
    num: int
    denom: int = 1

    def __post_init__(self):
        if not self.denom:
            raise ValueError('Denominator may not be empty')

f1 = Fraction(1, 2)
f2 = Fraction(3, 4)
```

## Named Tuples

A named tuple is a tuple where each element has a name, and can be accessed via the name in addition to the normal access by index.

- Thus, if `p` were a named tuple representing a point, you could say `p.x` and `p.y`, or you could say `p[0]` and `p[1]`.

A named tuple is created with the `namedtuple` class in the `collections` module.

- It essentially creates a new data type.

Pass the name of the tuple followed by a string containing the individual field names separated by spaces.

- `namedtuple()` return a class which with you can create instances of the tuple.
- A named tuple is the closest thing Python has to a C struct.

You can convert a named tuple into a dictionary with the `_asdict()` method.

`named_tuple_ex.py`

```

1 #!/usr/bin/env python
2 from collections import namedtuple
3
4 # create named tuple class with specified fields
5 # (could also provide fieldnames as iterable)
6 Knight = namedtuple('Knight', 'name title color quest comment')
7
8 # create named tuple instance (must specify all fields)
9 k = Knight('Bob', 'Sir', 'green', 'whirled peas', 'Who am i?')
10
11 print(k.title, k.name, k[1], k[0]) # access fields by name or index
12 print("*" * 30)
13 knights = {} # initialize dict for knight data
14 with open('../data/knights.txt') as knights_in:
15     for line in knights_in:
16         flds = line.rstrip().split(';')
17
18         # add knight tuple to dictionary where key is knight name
19         knights[flds[0]] = Knight(*flds)
20
21 # iterate over dictionary; info is knight tuple
22 for key, knight in knights.items():
23     print('{0} {1}: {2}'.format(knight.title, key, knight.color))

```

# Printing data structures

When debugging data structures, the print command is not so helpful.

- A complex data structure is just printed out all jammed together, one element after another, and is hard to read.

The `pprint` (pretty print) module will analyze a structure and print it out with indenting, to make it much easier to read.

- Output can be customized with some named parameters:
  - ▶ `indent` (default 1) specifies how many spaces to indent nested structures.
  - ▶ `width` (default 80) constrains the width of the output
  - ▶ `depth` (default unlimited) says how many levels to print—levels beyond depth are shown with '...'

*pretty\_printing.py*

```

1 #!/usr/bin/env python
2 from pprint import pprint
3
4 # nested data structure
5 data = {
6     'part1': [['a', 'b', 'c'], ['d', 'e', 'f']],
7     'part2': {'red': 55, 'blue': [8, 98, -3],
8               'purple': ['Chicago', 'New York', 'L.A.']},
9     'part3': ['g', 'h', 'i']
10 }
11
12 print('Without pprint:', data, "#" * 30, sep="\n")
13
14 print('With pprint:')
15 pprint(data)           # pretty-print
16 print()
17
18 print('With pprint (depth=2):')
19 pprint(data, depth=2)    # only print top two levels of structure
20 print()
```

## Zipped archives

The `zipfile` module allows you to read and write to zipped archives.

- In either case you first create a zipfile object.
  - ▶ Specifying a mode of "w" to write to an archive
  - ▶ Specifying a mode of "r" (the default) to read from existing zip file.

Modules for `gzipped` and `bzipped` files exist for creating gzip, bz2 in a similar fashion to the `zipfile` module

`zipfile_ex.py`

```

1 #!/usr/bin/env python
2 from zipfile import ZipFile, ZIP_DEFLATED
3 import os
4 import os.path
5
6
7 # creating a zip file using context manager (with ... as)
8 with ZipFile("example.zip", mode="w", compression=ZIP_DEFLATED) as zip_file:
9     for base in "parrot tyger knights alice breakfast mary".split():
10         filename = f'{base}.txt'
11         entry = os.path.join("../data", filename)
12         print(f"adding {entry} as {filename}")
13         zip_file.write(entry, filename) # Add member to zip file
14
15
16 # reading & extracting
17 destination = "extracted_files"
18 os.makedirs(destination, exist_ok=True)
19 with ZipFile("example.zip") as zip_file: # Open zip file for reading
20     print(zip_file.namelist()) # Print list of members in zip file
21     print()
22
23     # Read (raw binary) data from member and convert from bytes to string
24     tyger = zip_file.read('tyger.txt').decode()
25     print(tyger[:50])
26     zip_file.extract('parrot.txt', path=destination) # Extract member

```

# Tar Archives

To work with a tar archive, use the `tarfile` module.

- It can analyze a file to see whether it's a valid tar file, extract files from the archive, add files to the archive, and other tar chores.

The `is_tarfile()` function will check a tar file and return True if it is a valid tar file.

- This will work even if it is gzip bzip2 compressed.

For other actions, use the `open()` function to create a `TarFile` object.

- From this object you can list, add, or extract members, depending on how the tar file was opened.

A `TarFile` object is iterable as a list of `TarInfo` objects, which contain the details about each member.

- Use `getmembers()` to get a list of members as `TarInfo` objects.
- Use `extract()` to extract a file to disk.
  - ▶ The first argument to `extract()` is the member name
  - ▶ The named parameter `path` specifies the destination directory (default '!').
- Use `extractall()` to extract all members to the current directory, or a specified path.
  - ▶ A list of members may be specified; it must be a subset of the list returned by `getmembers()`.

To create a tar file, use `tarfile.open()` with a mode of `w` for an uncompressed archive.

- Use modes `w:gz` or `w:bz2` for a compressed archive.
- Use the `add()` method to add a file.

```
1 #!/usr/bin/env python
2 import tarfile
3 import os
4
5 # iterate over sample files
6 for tar_file in ('pres.tar', 'not_a.tar', 'textfiles.tar.gz'):
7     filename = os.path.join('../data', tar_file)
8     is_valid = tarfile.is_tarfile(filename) # check to see if file is tarfile
9     text = 'IS' if is_valid else 'IS NOT'
10    print("{} {} a tarfile".format(filename, text))
11 print()
12
13 with tarfile.open('../data/pres.tar') as tarfile_in: # open tar file
14     for member in tarfile_in:                      # iterate over members
15         print(member.name, member.size)            # access member data
16     print()
17
18 with tarfile.open('../data/pres.tar') as tarfile_in:
19     # extract member to local file
20     tarfile_in.extract('presidents.txt', path='../temp')
21
22 with tarfile.open('../data/textfiles.tar.gz') as tarfile_in:
23     # extract member to local file
24     tarfile_in.extract('knights.txt', path='../temp')
25
26 # open new tar archive for writing
27 with tarfile.open('../temp/text_files.tar', 'w') as tarfile_out:
28     tarfile_out.add('../data/parrot.txt')        # add member
29     tarfile_out.add('../data/alice.txt')         # add member
30
31 # open new tar archive for writing; archive is compressed with gzip
32 with tarfile.open('../temp/more_text_files.tar.gz', 'w:gz') as TAR:
33     TAR.add('../data/parrot.txt')                # add member
34     TAR.add('../data/alice.txt')                 # add member
```

# Archives the easy way

The `make_archive()` function in the `shutil` module makes it easy to create archives.

To use it, specify the `base_name` of the archive and the `format`.

- `base_name` is the name of the file to create
- `format` is the archive format as a string
  - ▶ By default `shutil` provides these formats:
    - ◆ `zip`: ZIP file (if the `zlib` module is available).
    - ◆ `tar`: Uncompressed tar file. Uses POSIX.1-2001 pax format for new archives.
    - ◆ `gtar`: gzip'ed tar-file (if the `zlib` module is available).
    - ◆ `bztar`: bzip2'ed tar-file (if the `bz2` module is available).
    - ◆ `xztar`: xz'ed tar-file (if the `lzma` module is available).
- The third argument (optional) is the `root_dir`.
  - ▶ `root_dir` is a directory that will be the root directory of the archive
    - ◆ All paths in the archive will be relative to it is the `root_dir` to archive.

Full documentation for the `shutil.make_archive` can found at the following URL:

<https://docs.python.org/3/library/shutil.html#archiving-operations>

The next example creates two archives of the contents of the `data` directory

`easy_archive.py`

```

1 #!/usr/bin/env python
2 import shutil
3
4 # Create an archive with a base name of ~/data_dir and .tar.gz extension
5 # compress the contents of the "../data" directory
6 shutil.make_archive('data_dir', 'gztar', '../data')
7
8
9 # Create an archive with a base name of ~/data_dir and .zip extension
10 # compress the contents of the "../data" directory
11 shutil.make_archive('data_dir', 'zip', '../data')
```

## Serializing Data

Serializing data means taking a data structure and transforming it so it can be written to a file or other destination, and later read back into the same data structure.

Python uses the `pickle` module for data serialization.

To create pickled data, use either `pickle.dump()` or `pickle.dumps()`.

- Both functions take a data structure as the first argument.

`dump()` writes the data to a file-like object which has been specified as the second argument.

- The file-like object must be opened for writing in binary mode.

`dumps()` returns the pickled data as a `bytes` object instead of writing to a file.

To read pickled data, use `pickle.load()` or `pickle.loads()`.

`load()` returns an object from a file-like object specified as its first argument.

- `load()` does not load the contents of the file, but rather returns one object at a time from the file.

`loads()` returns an object from a `bytes` object passed as the first argument.

- `load()` does not load the contents of the `bytes` object, but rather one object at a time from the file.

Both functions return the original data structure that had been pickled.

*pickling.py*

```
1 #!/usr/bin/env python
2 import pickle
3 from pprint import pprint
4
5 airports = {'RDU': 'Raleigh-Durham', 'IAD': 'Dulles', 'MGW': 'Morgantown',
6              'EWR': 'Newark', 'LAX': 'Los Angeles', 'ORD': 'Chicago'}
7
8 colors = ['red', 'blue', 'green', 'yellow', 'black', 'white', 'orange']
9
10 lucky_number, lucky_day = 13, "Thursday"
11
12 # list of objects
13 data = [airports, colors, lucky_number, lucky_day]
14
15 # open pickle file for writing in binary mode
16 with open('pickled_data.pickle', 'wb') as pic_out:
17     for obj in data:
18         pickle.dump(obj, pic_out) # serialize data structures to pickle file
19
20 # open pickle file for reading in binary mode
21 with open('pickled_data.pickle', 'rb') as pic_in:
22     while True: # Don't necessarily know how many things to read
23         try:
24             # de-serialize pickle file back into data structures
25             pickled_data = pickle.load(pic_in)
26             print(type(pickled_data))
27             pprint(pickled_data)
28             print()
29         except EOFError:
30             # raised when no data remains to be reading
31             # use this as an indicator to stop calling load()
32             break
```

---

## Exercises

### Exercise 1 (count\_ext.py)

Write a script which will count number of files with each extension in a file tree.

- It should take the initial directory as a command line argument, and then display the total number of files with each distinct file extension that it finds.
- Files with no extension should be skipped. Use a Counter object to do the counting.

### Exercise 2 (save\_potus\_info.py)

Write a script which creates a named tuple President, with fields lastname, firstname, birthplace, birthstate, and party.

- Read the data from Presidents.txt into a list of 44 President tuples.
- Write the list out to a file named potus.pic (use the Pickle module).

### Exercise 3 (read\_potus\_info.py)

Write a script to open potus.pic, and restore the data back into a list.

- Then loop through the list and print out each president's first name, last name, and party.

### Exercise 4 (make\_zip.py)

Write a script which creates a zip file containing the three .py files created in the first three exercises above.

## Exercise 5 (maxlist.py)

Create a new type, `MaxList`, that will only grow to a certain size.

- It should raise an `IndexError` if the user attempts to add an item beyond the limit.

HINT:

- You will need to override the `append()` and `extend()` methods
- To be thorough, you would need to override `__init__()` as well, so that the initializer doesn't have more than the maximum number of items., and `pop()`, and maybe some others.

For the ambitious (ringlist.py):

- Modify `MaxList` so that once it reaches maximum size, appending to the list also removes the first element
  - ▶ So the list stays constant size once it is at its maximum.

## Exercise 6 (strdict.py)

Create a new type, `NormalStringDict`, that only allows strings as keys *and* values.

- Values are normalized by making them lower case and remove all whitespace.
- Remember to support the `.update()` method!



# Chapter 6. Generators and Coroutines

## Objectives

- Unpack iterables with wildcards
- Create generators in 3 different ways
- Create iterable Coroutines

## Extended iterable unpacking

Before we begin an in depth discussion of generators and coroutines, we will take a look at the topic of **extended iterable unpacking**

- Since generators and coroutines are iterable objects, they can benefit as can all iterables from extended unpacking.

When unpacking iterables, sometimes you want to grab parts of the iterable as a group.

- This is what extended iterable unpacking provides.

One (and only one) variable in the result of unpacking can have a star prepended.

- This variable will receive all values from the iterable that do not go to other variables.

*extended\_iterable\_unpacking.py*

```

1 #!/usr/bin/env python
2
3 values = ['a', 'b', 'c', 'd', 'e'] # *values* has 6 elements
4
5 x, y, *z = values # *z takes all extra elements from iterable
6 print("x: {}    y: {}    z: {}".format(x, y, z))
7 print()
8
9 x, *y, z = values # *y takes all extra elements from iterable
10 print("x: {}    y: {}    z: {}".format(x, y, z))
11 print()
12
13 *x, y, z = values # *x takes all extra elements from iterable
14 print("x: {}    y: {}    z: {}".format(x, y, z))
15 print()
16
17 people = [('Bill', 'Gates', 'Microsoft'), ('Steve', 'Jobs', 'Apple'),
18             ('Paul', 'Allen', 'Microsoft'), ('Larry', 'Ellison', 'Oracle'),
19             ('Mark', 'Zuckerberg', 'Facebook'), ('Sergey', 'Brin', 'Google'),
20             ('Larry', 'Page', 'Google'), ('Linus', 'Torvalds', 'Linux')]
21
22 for *name, _ in people: # *name* gets all but the last field
23     print(name)
24 print()
```

# What exactly is an iterable?

An **iterable** is an object that provides an **iterator** (via the special method `__iter__`).

An iterator is an object that responds to the `next()` builtin function, via the special method `__next__()`.

- In other words, an iterator is an object which can be looped over with a `for` loop.

All generators are iterables.

- Most sequence and mapping types are also iterables.

Generators are also iterators.

- `next()` can be used on them directly.

For some iterables (including most collections), you can not use `next()` on them directly.

- They can be passed to the `iter()` function to convert them into an iterator.

*iterable\_to\_iterator.py*

```

1 #!/usr/bin/env python
2 r = range(1, 4)
3 i = iter(r)
4 print(f"Type of 'r':{type(r)}      Type of 'i':{type(i)}\n")
5
6 print("While loop:", end=" ")
7 while True: # Explicitly testing for StopIteration to know when to stop.
8     try:
9         print(next(i), end=" ")
10    except StopIteration: # iterable is depleted, code should stop iterating
11        break
12
13 i = iter(range(1, 4))
14 print("\tFor loop:", end=" ")
15 for value in i:           # For loop implicitly calls next() and
16     print(value, end=" ")  # automatically stops and handles exception quietly
17 print()

```

```

$ python iterable_to_iterator.py
Type of 'r':<class 'range'>      Type of 'i':<class 'range_iterator'>

While loop: 1 2 3   For loop: 1 2 3
$
```

## Generators

A generator is an object that provides values on demand (AKA "lazy"), rather than storing all the values (AKA "eager").

Generators are usually based on some other iterable.

- Another way of saying this is that a generator returns an iterator that returns a new value each time `next()` is called on it, until there are no more values.

The big advantage of generators is saving memory.

- They act as a **view** over a collection of data.

Generators may only be used once.

- After the last value is provided, the generator must be recreated in order to start over.

Generators may not be indexed, and have no length.

- The only thing to do with a generator is to loop over it with **for** loop.

There are three ways to create generators in Python:

- Generator functions
- Generator expressions
- Generator classes

# Generator functions

The earlier chapter entitled Pythonic Programming introduced generator functions.

- Recall that a generator function is like a normal function, but instead of a `return` statement, it has a `yield` statement.
  - ▶ Each time the `yield` statement is reached, it provides the next value in the sequence.
  - ▶ When there are no more values, the function calls `return`, and the loop stops.

A generator function maintains state between calls, unlike a normal function.

The next example combines an `Enum` with a `namedtuple` and a generator function to iterate through a user supplied Unicode block of characters.

- While each Unicode block may have many characters, the generator function does not need to hold all of them.
  - ▶ It will simply be able to yield one at a time to the caller as needed.

The generator function is defined as a instance method named `characters` in the `UnicodeBlock` class.

- 10 different `UnicodeBlock` types will be declared in the `Enum`.
  - ▶ Each one will store a `namedtuple` named `BlockBoundary` as its `value`.
    - ◆ Each `BlockBoundary` stores the `start` and `stop`(inclusive) values as a hexadecimal number of the Unicode block of code they represent.

```
1 #!/usr/bin#!/usr/bin/env python
2 from collections import namedtuple
3 from enum import Enum
4
5 BlockBoundary = namedtuple('BlockBoundary', 'start stop')
6
7
8 class UnicodeBlock(Enum):
9     Arrows = BlockBoundary(0x2190, 0x21FF)
10    MathematicalOperators = BlockBoundary(0x2200, 0x22FF)
11    ControlPictures = BlockBoundary(0x2400, 0x2426)
12    EnclosedAlphanumerics = BlockBoundary(0x2460, 0x24FF)
13    MiscellaneousSymbols = BlockBoundary(0x2600, 0x26FF)
14    SupplementalArrowsA = BlockBoundary(0x27F0, 0x27FF)
15    SupplementalArrowsB = BlockBoundary(0x2900, 0x297F)
16    Emoticons = BlockBoundary(0x1F600, 0x1F64F)
17    MiscSymbolsAndPictographs = BlockBoundary(0x1F300, 0x1F5FF)
18    Dominos = BlockBoundary(0x1F030, 0x1F093)
19
20    @property
21    def start(self):
22        return self.value.start
23
24    @property
25    def stop(self):
26        return self.value.stop
27
28    def characters(self):
29        for unicode_value in range(self.start, self.stop + 1):
30            yield chr(unicode_value)
31
32
33 def main():
34     characters = UnicodeBlock.Emoticons.characters()
35     for char in characters:
36         print(char, end=" ")
37     print()
38
39
40 if __name__ == '__main__':
41     main()
```

A more robust application that uses the `UnicodeBlock` class is shown below

`unicode_block_app.py`

```
1 #!/usr/bin/env python
2 from unicode_blocks import UnicodeBlock
3
4
5 def create_menu():
6     menu = {}
7     for ascii_value, unicode_block in enumerate(UnicodeBlock, start=65):
8         ascii_char = chr(ascii_value)
9         menu[ascii_char] = unicode_block
10    return menu
11
12
13 def print_menu(menu):
14     for char, unicode_block in menu.items():
15         print(f'{char} -- {unicode_block.name}')
16     print()
17
18
19 def prompt_user():
20     prompt = "Enter a letter from the menu for\nthe desired Unicode block > "
21     return input(prompt).upper()
22
23
24 def main():
25     # print MiscellaneousSymbols
26     menu = create_menu()
27     print_menu(menu)
28     user_choice = prompt_user()
29
30     if user_choice not in menu:
31         print(user_choice, " is not one of the menu options")
32         return
33
34     for char in menu[user_choice].characters():
35         print(char, end=" ")
36     print()
37
38
39 if __name__ == "__main__":
40     main()
```

## Generator Expressions

The earlier chapter entitled Pythonic Programming also introduced generator expressions.

Recall a generator expression is similar to a comprehension, but it provides a generator instead of a collection.

- That is, while a comprehension returns a complete collection (**list, set, dict**), the generator expression returns one item at a time.
  - ▶ This gives the generator expression a much smaller memory footprint than a comprehension
  - ▶ In many cases generators are also faster than list comprehensions.

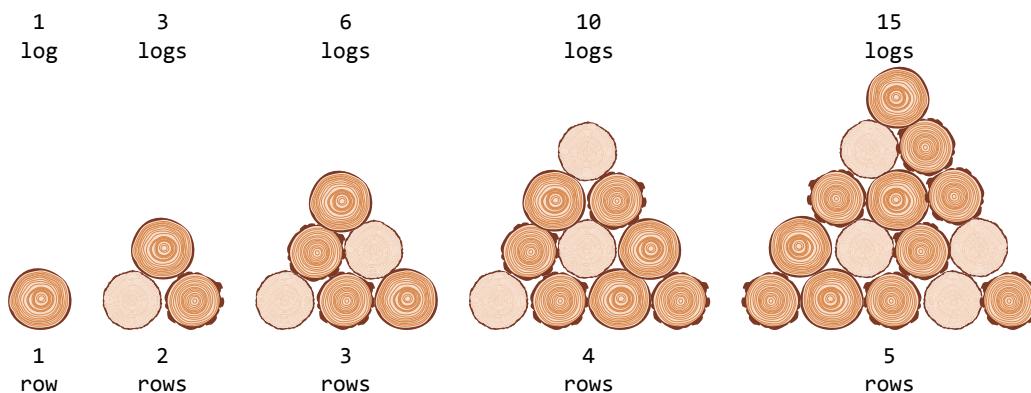
The generator expression uses parentheses rather than square brackets or curly braces.

- There is an implied yield statement at the beginning of the expression.
- If a generator expression is passed as the only argument to a function, the extra parentheses are not needed.

Generator expressions are especially useful with functions like `sum()`, `min()`, and `max()` that reduce an iterable input to a single value.

The next example uses a generator expression to generate what is known as the **Triangular Number Sequence**"

- The Triangular Number Sequence is generated from a pattern of objects which form a triangle:
  - ▶ By adding another row of objects and counting all the objects we can find the next number of the sequence.
  - ▶ This can be seen in the image below that shows the stacking of tree logs.



The mathematical formula for the above sequence is **logs = rows(rows+1)/2**

*triangle\_sequence.py*

```
1 #!/usr/bin/env python
2
3 max_rows = int(input("Max logs that will be placed on the bottom row? "))
4
5 # create a generator expression that generates the triangle sequence.
6 logs = (row * (row + 1) // 2 for row in range(1, max_rows + 1))
7
8 fmt = '{:^10}' * 2
9 print(fmt.format("#Rows", "#Logs"))
10 for rows, log in enumerate(logs, start=1):
11     print(fmt.format(rows, log))
```

```
$ python triangle_sequence.py
Max logs that will be placed on the bottom row? 9
#Rows      #Logs
 1          1
 2          3
 3          6
 4         10
 5         15
 6         21
 7         28
 8         36
 9         45
$
```

## Coroutines

When writing generator functions, you can send a value **back** to the generator.

- This is accomplished by calling `generator.send()`
  - ▶ In the generator it becomes the return value of the `yield` statement.
  - ▶ This makes the generator into a **coroutine**.

While generators and coroutines are similar, they are not the same thing.

- Generators tend to be producers
- Coroutines tend to be consumers.

Assigning a variable to the `yield` statement is what makes it a coroutine

- Its `next()` needs to be called once to "prime" the coroutine
  - ▶ This prepares it to receive the first input, and thus send the first output.
  - ◆ There's an "off-by-one" feeling to this.

Coroutines are not designed for iteration.

- They can be used to set up pipelines
  - ▶ Pipelines are sequences of coroutines that each do one interesting thing to your data.
  - ▶ `send()` puts data into the pipeline.
- This can be useful for async-like coding.

The following link provides an in-depth look at coroutines.

<http://dabeaz.com/coroutines/>

The following example defines a simple coroutine that consumes and prints what is sent to it.

*coroutine\_example.py*

```
1 #!/usr/bin/env python
2 def coroutine(): # Define coroutine function
3     while True:
4         # Assigning to yield makes it coroutine, not generator
5         value_received = yield
6         print('value_consumed:', value_received)
```

The application below uses the coroutine defined in the previous example.

#### *coroutine\_app.py*

```
1 #!/usr/bin/env python
2 from coroutine_example import coroutine
3 c = coroutine() # Create instance of coroutine
4
5 print(type(c), c) # Print coroutine object
6 next(c) # Prime (start) the coroutine
7
8 for letter in "alpha", "beta", 'gamma':
9     c.send(letter) # Send in data
10 print()
```

```
$ python coroutine_app.py
<class 'generator'> <generator object coroutine at 0x7f3db1761970>
value_consumed: alpha
value_consumed: beta
value_consumed: gamma
$
```

The next examples defines several coroutines.

- Each coroutines takes and optional coroutine as a parameter.
  - ▶ This allows the data consumed by one coroutine to piped to another.

## coroutines\_as\_pipes.py

```

1 #!/usr/bin/env python
2
3
4 def segment_a(pipe=None): # Define coroutine function
5     if pipe:
6         next(pipe) # prime the coroutine
7     while True:
8         # Assigning to yield makes it coroutine, not generator
9         value_received = yield
10        print('value_consumed by segment_a:', value_received)
11        # convert what it consumes to title case version
12        all_title_case = [value.title() for value in value_received]
13        print('processed result of segment_a:', all_title_case)
14        if pipe: # pipe it to next coroutine
15            pipe.send(all_title_case)
16
17
18 def segment_b(pipe=None): # Define coroutine function
19     if pipe:
20         next(pipe) # prime the coroutine
21     while True:
22         # Assigning to yield makes it coroutine, not generator
23         value_received = yield
24         print('value_consumed by segment_b:', value_received)
25         sorted_result = sorted(value_received) # sort what it consumes
26         print('processed result of segment_b:', sorted_result)
27         if pipe: # pipe it to next coroutine
28             pipe.send(sorted_result)
29
30
31 pipe = segment_a() # Create instance of coroutine
32 next(pipe) # Prime (start) the coroutine
33
34 for letter in "alpha", "beta", 'gamma':
35     pipe.send(letter) # Send in data
36 print()
37
38 # Create instance of coroutine, by piping it to another coroutine
39 pipe = segment_a(segment_b())
40 next(pipe) # Prime (start) the coroutine
41
42 for letter in "alpha", "beta", 'gamma':
43     pipe.send(letter) # Send in data
44 print()

```

## Generator classes

The most flexible way to create a generator is by defining a **generator class**.

- Such a class must implement a special method:

- ▶ `__iter__(self)` must return an object that implements `__next__(self)`.
  - ◆ In most cases, this is the same class, so `__iter__(self)` just returns `self`.
- ▶ `__next__(self)` returns the next value from the generator.
  - ◆ This can be in a loop, or in sequential statements, or any combination.
  - ◆ When there are no more values to return, `__next__(self)` should raise a `StopIteration` exception.

*trimmedfile.py*

```

1 #!/usr/bin/env python
2 class TrimmedFile:
3     '''Trims trailing white space read from each line of a file.'''
4     def __init__(self, file_name):
5         '''file_name - name of file to open for reading'''
6         self._file_in = open(file_name)
7
8     # __iter__ returns the iterator (i.e., object that implements __next__)
9     def __iter__(self):
10        '''Returns self as iterator object'''
11        return self
12
13    # __next__ returns next value
14    def __next__(self):
15        '''Returns each line of text read with trailing whitespace removed'''
16        line = self._file_in.readline()
17
18        # When no more values to provide, raise StopIteration
19        if line == '':
20            raise StopIteration
21
22        return line.rstrip() # Return next value
23
24
25 if __name__ == '__main__':
26     trimmed = TrimmedFile('../data/mary.txt') # Create instance of generator
27     for line in trimmed:
28         print(line) # Line is now trimmed

```

# Exercises

## Exercise 1 (pres\_upper\_gen.py)

Redo `pres_upper.py` from previous chapter exercises, but use two generator expressions rather than two list comprehensions.

## Exercise 2 (siblings.py)

Write a function `sibling_rivalry` that takes in two lists: one of siblings, and the other of friends.

Return an iterable containing only siblings that are *not* friends.

## Exercise 3 (fauxrange.py)

Write a generator class named `FauxRange` that emulates the builtin `range()` function.

- Instances should take up to three parameters, and provide a range of integers
  - ▶ Consider using floats—does that change the class?.



# Chapter 7. Type Hinting

## Objectives

- Learn how to annotate variables with their type
- Understand what the type hints do **not** provide
- Employ the `typing` module to annotate collections
- Use the `Union` and `Optional` types correctly
- Write stub interfaces using type hints

## Type Hinting

Python supports optional **type hinting** of variables, functions, and parameters.

- This is not enforced by the Python interpreter.

Types may be specified by following the variable name with a `:` and a datatype.

- Default values can also be assigned when it is declared
  - ▶ Once again, this is type **hinting** only
  - ▶ The Python interpreter does not check the types in any way:

*basic\_hinting.py*

```

1 #!/usr/bin/env python
2 """Basic type hinting example"""
3
4
5 first_name: str           # declare first_name as being of type str
6 last_name: str = "Doe"     # declare alst_name as str with initial value
7
8 number: int = 99          # declare and initialize number as an int
9
10 data: list = [5, 6, 7]     # declare and initialize data as a list
11
12 first_name = -1234
13 # While the following are valid Python statements
14 # they violate the intent of the hinting.
15
16 # value is not the intended type of str
17 number = "This is not a number"
18 key_values: dict = ["these", "are", "not", "key", "value", "pairs"]
19
20 print(first_name, last_name, number, data, key_values, sep=" | ")
21 print()
22 print(f'{last_name + number}') # Type hinting suggests this cannot happen

```

The output of the above program shows that the Python interpreter does not enforce the type hinting.

```

$ python basic_hinting.py
-1234 | Doe | This is not a number | [5, 6, 7] | ['these', 'are', 'not', 'key', 'value',
'pairs']
DoeThis is not a number
$
```

# Static Analysis Tools

If these type hints are not used by the Python interpreter, how are they useful?

While Python (currently) does not use the type hints in any way, static analysis tools do.

- The most common tool for static type analysis is the `mypy` module.
  - ▶ This is not part of the standard distribution and is a separate `pip` install.

```
pip install mypy
Collecting mypy
  Downloading mypy-0.782-cp38-cp38-manylinux1_x86_64.whl (21.7 MB)
    |████████████████████████████████| 21.7 MB 9.0 MB/s
Collecting typing-extensions>=3.7.4
  Downloading typing_extensions-3.7.4.3-py3-none-any.whl (22 kB)
Collecting typed-ast<1.5.0,>=1.4.0
  Downloading typed_ast-1.4.1-cp38-cp38-manylinux1_x86_64.whl (768 kB)
    |████████████████████████████████| 768 kB 44.0 MB/s
Collecting mypy-extensions<0.5.0,>=0.4.3
  Downloading mypy_extensions-0.4.3-py2.py3-none-any.whl (4.5 kB)
Installing collected packages: typing-extensions, typed-ast, mypy-extensions, mypy
Successfully installed mypy-0.782 mypy-extensions-0.4.3 typed-ast-1.4.1 typing-extensions-3.7.4.3
```

The `mypy` module scans the code - technically, the **Abstract Syntax Tree( AST)** of the code

- It determines, at "compile" time, whether the types expected and used match up correctly.
- It will emit errors when it detects static typing problems in the code.

The example below shows how to run `mypy` as a module and pass to it the module to scan.

```
$ python -m mypy basic_hinting.py
basic_hinting.py:12: error: Incompatible types in assignment (expression has type "int",
variable has type "str")
basic_hinting.py:17: error: Incompatible types in assignment (expression has type "str",
variable has type "int")
basic_hinting.py:18: error: Incompatible types in assignment (expression has type
"List[str]", variable has type "Dict[Any, Any]")
basic_hinting.py:22: error: Unsupported operand types for + ("str" and "int")
Found 4 errors in 1 file (checked 1 source file)
$
```

While type hinting can be checked at runtime, such checking involves a very high performance cost.

## Type Hinting Functions

Functions use a `->` symbol to indicate a return type.

- Function parameters are annotated with type hints the same way as the variables in the previous example.

*function\_hinting.py*

```

1 #!/usr/bin/env python
2 """Function type hinting example"""
3
4
5 def repeat(word: str, times: int = 1) -> str:
6     return word * times
7
8
9 def main() -> None:
10    print(repeat("Absolutely", 4))
11
12    # Python interpreter permits the following
13    # But the mypy module will flag it as an issue
14    print(repeat(5, 6))
15
16
17 if __name__ == '__main__':
18    main()

```

The python interpreter has no problem running the above code successfully.

```

$ python function_hinting.py
AbsolutelyAbsolutelyAbsolutelyAbsolutely
30
$
```

The `mypy` module will flag the call to `repeat(5, 6)` as an error as shown below.

```

$ python -m mypy function_hinting.py
function_hinting.py:14: error: Argument 1 to "repeat" has incompatible type "int";
expected "str"
Found 1 error in 1 file (checked 1 source file)
$
```

The variable number of arguments and keyword arguments parameters may also be type hinted.

- The values are then expected to all be of the specified type.
  - The keyword arguments parameter only type hints the values, the keys are still strings.

*shouting.py*

```

1 #!/usr/bin/env python
2 """Function type hinting example"""
3
4
5 def shout(times: int = 1, *args: str, **kwargs: int) -> str:
6     for word in args:
7         print(word.upper() * times)
8     print("-" * 30)
9     for word, times in kwargs.items():
10        print(word.upper() * times)
11
12
13 def main() -> None:
14     shout(2, "Hello", "Goodbye")
15     print()
16     print("#" * 30)
17     shout(3, "Something", "Something Else", alpha=3, beta=5, gamma=4)
18
19
20 if __name__ == '__main__':
21     main()
```

```

$ python shouting.py
HELLOHELLO
GOODBYEGOODBYE
-----
#####
SOMETHINGSOMETHINGSOMETHING
SOMETHING ELSE SOMETHING ELSE SOMETHING ELSE
-----
ALPHAALPHAALPHA
BETABETABETABETABETA
GAMMAGAMMAGAMMAGAMMA
$
```

## \_\_annotations\_\_

The mechanism behind how type hints are tracked is through a dictionary called \_\_annotations\_\_.

- There is one at the global level, for any global variables declared with type hinting.
- There also exist \_\_annotations\_\_ dictionary attributes on both functions and classes.
- While annotations may be made on local variables, Python does not track these programmatically.
  - ▶ There is no local \_\_annotations\_\_ dictionary.
    - ◆ Static tools such as `mypy` are expected to parse these annotations themselves, rather than having the runtime cost of the interpreter parsing them and filling in another dictionary.

This \_\_annotations\_\_ dictionary actually just stores key: value pairs.

- The key is simply the string version of the variable name.
  - ▶ The return type of the function is stored in the key named `return`.
  - ▶ This ensures that no parameter name will conflict with the return type (as `return` is a keyword).
- The value is the type specified.
  - ▶ Interestingly, the "type" could be any Python value!

*getting\_annotations.py*

```

1 #!/usr/bin/env python
2
3
4 def regular_hints(times: int = 1, *args: str, **kwargs: int) -> str:
5     pass
6
7
8 # Although weird looking, this is perfectly valid Python code.
9 def weird_hints(word: len, times: 0 = 1) -> 'unk':
10    pass
11
12
13 # The __annotations__ dictionary stores the key and value of each annotation.
14 print(regular_hints.__annotations__, weird_hints.__annotations__, sep='\n')
```

```
$ python getting_annotations.py
{'times': <class 'int'>, 'args': <class 'str'>, 'kwargs': <class 'int'>, 'return': <class 'str'>}
{'word': <built-in function len>, 'times': 0, 'return': 'unk'}
```

# Forward References

Not all types may be available at the time that a given piece of Python code is compiled to bytecode.

- In other words, **forward references**, where a type is referred to before it is defined, are needed.

Starting with Python 3.7, this is handled automatically with a particular `__future__` import.

- It is slated to be the default behavior in Python 4.0.

*forward\_refs\_new.py*

```

1 #!/usr/bin/env python
2 from __future__ import annotations
3
4
5 class First:
6     def process(self, item: Second) -> str:
7         pass
8
9
10 class Second:
11     def create(self, data: First) -> str:
12         pass

```

The prior way to do this in Python is by using strings; tools are expected to handle this forward reference.

- The `mypy` tool deals correctly with these forward references.

*forward\_refs\_old.py*

```

1 #!/usr/bin/env python
2 class First:
3     # The type Second is not yet available to python, so it must be
4     # forward-declared using a string
5     def process(self, item: 'Second') -> str:
6         pass
7
8
9 class Second:
10    # The type First is available to python, so it can just reference
11    # the First symbol directly
12    def create(self, data: First) -> str:
13        pass

```

Forward references are also how various special dunder methods may need to be written, to refer to the current class.

*forward\_unders.py*

```
1 #!/usr/bin/env python
2
3 class Fraction:
4     def __add__(self, item: Fraction) -> Fraction:
5         pass
6
7     def __sub__(self, item: Fraction) -> Fraction:
8         pass
```

# typing Module

The `typing` module makes it easier to refer to containers as a kind of type in Python code.

The `typing` module makes available the following type-wrapper classes, by default all are invariant:

*Table 6. typing Type-wrapper Classes*

<code>ABCMeta</code>	<code>AbstractSet</code>	<code>Any</code>	<code>AnyStr</code>
<code>AsyncContextManager</code>	<code>AsyncGenerator</code>	<code>AsyncIterable</code>	<code>AsyncIterator</code>
<code>Awaitable</code>	<code>BinaryIO</code>	<code>ByteString</code>	<code>CT_co</code>
<code>Callable</code>	<code>ChainMap</code>	<code>ClassVar</code>	<code>Collection</code>
<code>Container</code>	<code>ContextManager</code>	<code>Coroutine</code>	<code>Counter</code>
<code>DefaultDict</code>	<code>Deque</code>	<code>Dict</code>	<code>Final</code>
<code>ForwardRef</code>	<code>FrozenSet</code>	<code>Generator</code>	<code>Generic</code>
<code>Hashable</code>	<code>IO</code>	<code>ItemsView</code>	<code>Iterable</code>
<code>Iterator</code>	<code>KT</code>	<code>KeysView</code>	<code>List</code>
<code>Literal</code>	<code>Mapping</code>	<code>MappingView</code>	<code>Match</code>
<code>MethodDescriptorType</code>	<code>MethodWrapperType</code>	<code>MutableMapping</code>	<code>MutableSequence</code>
<code>MutableSet</code>	<code>NamedTuple</code>	<code>NamedTupleMeta</code>	<code>NewType</code>
<code>NoReturn</code>	<code>Optional</code>	<code>OrderedDict</code>	<code>Pattern</code>
<code>Protocol</code>	<code>Reversible</code>	<code>Sequence</code>	<code>Set</code>
<code>Sized</code>	<code>SupportsAbs</code>	<code>SupportsBytes</code>	<code>SupportsComplex</code>
<code>SupportsFloat</code>	<code>SupportsIndex</code>	<code>SupportsInt</code>	<code>SupportsRound</code>
<code>T</code>	<code>T_co</code>	<code>T_contra</code>	<code>Text</code>
<code>TextIO</code>	<code>Tuple</code>	<code>Type</code>	<code>TypeVar</code>
<code>TypedDict</code>	<code>Union</code>	<code>VT</code>	<code>VT_co</code>
<code>V_co</code>	<code>ValuesView</code>	<code>WrapperDescriptorType</code>	

The complete documentation for type hinting can be found in detail at the following URL:

<https://docs.python.org/3/library/typing.html>

## Type Hinting of Parameters

`Tuple` objects generally specify exactly which type each positional value is, such as `Tuple[str, int, str]`.

*basic\_containers.py*

```
1 #!/usr/bin/env python
2 from typing import Tuple
3
4
5 def process(record: Tuple[str, int, float]) -> None: # Expects a three-tuple
6     pass
```

A Tuple of arbitrary length (but the same type throughout) may be specified using the `Ellipsis` object.

*combinations.py*

```
1 #!/usr/bin/env python
2 from typing import Tuple, List
3
4
5 def fullname_combos(first_names: Tuple[str, ...],
6                      last_names: Tuple[str, ...]) -> List[str]:
7     # Join the first and last names using a nested list comprehension
8     return [' '.join([first_name, last_name])
9            for first_name in first_names
10           for last_name in last_names]
11
12
13 print(fullname_combos(("Joe", "Diane"), ("Smith", "Williams", "Johnson")))
```

Most parameter types should not be of a specific container type such as `Dict`, `Set`, and `List`.

- The type hint should be how the container is used, such as an `Iterable` or a `Collection`.
- Most of the more-specific containers should generally only find use as return values.

PEP484 discusses how to hint parameters that take a variable number of arguments.

<https://www.python.org/dev/peps/pep-0484/#arbitrary-argument-lists-and-default-argument-values>

- For a variable number of positional arguments (`*args`) or keyword arguments (`**kwargs`), only the expected value for one such argument needs to be specified.

The following example demonstrates how to type hint a variable number of postional arguments.

*variable\_args.py*

```
1 #!/usr/bin/env python
2 from typing import Tuple, List
3
4
5 def average(*numbers: float) -> float:
6     return sum(numbers)/len(numbers)
7
8
9 print(f'{average(4, 5, 6.2, 7, 8.9):.3f}')
```

## Generator type hinting

The `typing.Generator` type takes exactly three types for its template:

- The type yielded
- The type sent
  - ▶ Unusually, the send-type for a generator is contravariant (because a generator is a function).
- The type returned by the generator.
  - ▶ If any of the three types are not used, they should be set to `None`.

`generator_coroutine.py`

```

1 #!/usr/bin/env python
2 from typing import Tuple, Generator, Any
3
4
5 def fullname_combos(first_names: Tuple[str, ...],
6                      last_names: Tuple[str, ...]) -> Generator[str, None, None]:
7     # Join the first and last names using a generator expression
8     yield from (' '.join([first_name, last_name])
9                 for first_name in first_names
10                for last_name in last_names)
11
12
13 def coroutine() -> Generator[None, Any, None]:
14     while True:
15         # Assigning to yield makes it coroutine, not generator
16         value_received = yield
17         print('value_consumed:', value_received)
18
19
20 for name in fullname_combos(("Joe", "Diane"), ("Smith", "Williams", "Jones")):
21     print(name)
22
23
24 c = coroutine() # Create instance of coroutine
25 next(c) # Prime (start) the coroutine
26
27 for letter in "alpha", "beta", 'gamma', 99, 45.3:
28     c.send(letter) # Send in data

```

# Creating Types

Creating new type aliases in Python follows the same process as creating any other kind of variable.

`aliasing.py`

```

1 #!/usr/bin/env python
2 from typing import Tuple, Generator, Any
3 Names = Tuple[str, ...]                      # Create an alias named Name
4 GenA = Generator[str, None, None]            # Create an alias named GenA
5 GenB = Generator[None, Any, None]            # Create an alias named GenB
6
7
8 def fullname_combos(first_names: Names, last_names: Names) -> GenA:
9     # Join the first and last names using a generator expression
10    yield from (' '.join([first_name, last_name])
11                for first_name in first_names
12                for last_name in last_names)
13
14
15 def coroutine() -> GenB:
16     while True:
17         # Assigning to yield makes it coroutine, not generator
18         value_received = yield
19         print('value_consumed:', value_received)

```

When working with generic types, it can be important to signal that a given type is maintained throughout a function call.

- In this case, a generic type is needed (rather than inheriting from a specific type), so a new one is created with `TypeVar`.

`generic_types.py`

```

1 #!/usr/bin/env python
2 from typing import TypeVar, Sequence
3 Choice = TypeVar('Choice')
4
5
6 def third(coll: Sequence[Choice]) -> Choice:
7     return coll[2]
8
9
10 print(third([1, 2, 3]), third(("A", "B", 5, "C")), third('Something'))

```

The `TypeVar` can restrict itself to a specific set of types:

*restricted\_generic\_types.py*

```

1 #!/usr/bin/env python
2 from typing import TypeVar, Sequence
3 Choice = TypeVar('Choice', int, str) # Restrict the choice to int or str
4 Numeric = TypeVar('Numeric', int, float)
5
6
7 def first(numbers: Sequence[Numeric]) -> Numeric:
8     return numbers[0]
9
10
11 def third(coll: Sequence[Choice]) -> Choice:
12     return coll[2]
13
14
15 print(first([99, 77, 55, 88]))
16
17 print(third([1, 2, 3]))
18 print(third(["A", "B", "C", "D"]))
19
20 # Python is ok with these two - but mypy will complain
21 print(third(["A", 1, "B", 2]))
22 print(third([2.3, 5.4, 6.9]))
```

```

$ python restricted_generic_types.py
99
3
C
B
6.9
$
$ python -m mypy restricted_generic_types.py
restricted_generic_types.py:20: error: Value of type variable "Choice" of "third" cannot
be "object"
restricted_generic_types.py:21: error: Value of type variable "Choice" of "third" cannot
be "float"
Found 2 errors in 1 file (checked 1 source file)
$
```

## Variance

When dealing with types and inheritance, certain interactions need to be made explicit.

*animal\_kingdom.py*

```

1 #!/usr/bin/env python
2 from __future__ import annotations
3 from enum import Enum, auto
4
5
6 class Move(Enum):
7     walk = auto()
8     run = auto()
9     slither = auto()
10    swim = auto()
11    fly = auto()
12
13
14 class Animal:
15     def __init__(self: Animal, move: Move):
16         self._move = move
17
18     @property
19     def move(self) -> Move:
20         return self._move
21
22
23 class Mammal(Animal):
24     def __init__(self: Mammal, move: Move, hair: str):
25         super().__init__(move)
26         self._hair = hair
27
28     @property
29     def hair(self) -> str:
30         return self._hair
31
32
33 class Cat(Mammal):
34     pass
35
36
37 class Dog(Mammal):
38     pass

```

- The example above defines the relationships of **Cat**, **Dog**, **Mammal**, and **Animal**.

## Covariance

Consider the following example:

*brush\_hair.py*

```

1 #!/usr/bin/env python
2 from animal_kingdom import Move, Mammal, Cat, Dog
3
4
5 def brush_hair(mammal: Mammal) -> None:
6     print(f"The {type(mammal).__name__}'s hair is {mammal.hair}")
7
8
9 def main() -> None:
10    cat = Cat(Move.walk, "short and grey")
11    dog = Dog(Move.run, "long and shaggy")
12    brush_hair(cat)
13    brush_hair(dog)
14
15
16 if __name__ == '__main__':
17     main()

```

The `brush_hair` function allows any child class of `Mammal` to have its hair brushed.

It makes logical sense that any subtype of `Mammal` would be able to have its hair brushed.

- So, passing a subtype of `Mammal` to `brush_hair` should be acceptable to both the Python interpreter and the typing system.

This kind of relationship between an argument's type and its inheritance is known as **covariance**.

- The subtype of an argument can be used in the place of the argument's type.
- This is also known as the **Liskov Substitution Principle**.

But a function with a parameter that takes a generic type, such as a `List`, involves subtle traps.

# Invariance

Consider the following example:

*invariant\_brush\_hair.py*

```

1 #!/usr/bin/env python
2 from animal_kingdom import Move, Mammal, Cat, Dog
3 from typing import List
4
5
6 def brush_hair_all(mammals: List[Mammal]) -> None:
7     for mammal in mammals:
8         print(f"The {type(mammal).__name__}'s hair is {mammal.hair}")
9     print()
10
11
12 cats_and_dogs: List[Mammal] = [Cat(Move.walk, "grey"), Dog(Move.run, "shaggy")]
13 brush_hair_all(cats_and_dogs)
14
15 all_cats: List[Cat] = [Cat(Move.run, "orange"), Cat(Move.walk, "scraggly")]
16 brush_hair_all(all_cats)
17
18 all_dogs: List[Dog] = [Dog(Move.run, "short and curly"),
19                         Dog(Move.walk, "white with black dots")]
20 brush_hair_all(all_cats)
```

It makes sense that there might be both cats and dogs that make up a collection of Mammals, all of which get their hair brushed.

- Similarly, a collection of only cats or only dogs could still all have their hair brushed

The problem is that many of the built-in mutable objects in Python such as `List`, `Set`, and `Dict` are **invariant**.

- It is true that the relationship between a `Cat` and a `Mammal` is covariant, that relationship does not carry over to mutable generic containers
- In other words:
  - ▶ A `Cat` is a subclass of `Mammal`, but a `List[Cat]` is not a subclass of `List[Mammal]`.
  - ▶ While the Python interpreter does not enforce the type hinting relationship, `mypy` will flag it as an error as shown on the following page.

```
$ python invariant_brush_hair.py
The Cat's hair is grey
The Dog's hair is shaggy

The Cat's hair is orange
The Cat's hair is scraggly

The Cat's hair is orange
The Cat's hair is scraggly
$

$ python -m mypy invariant_brush_hair.py
invariant_brush_hair.py:16: error: Argument 1 to "brush_hair_all" has incompatible type
"List[Cat]"; expected "List[Mammal]"
invariant_brush_hair.py:16: note: "List" is invariant -- see
http://mypy.readthedocs.io/en/latest/common_issues.html#variance
invariant_brush_hair.py:16: note: Consider using "Sequence" instead, which is covariant
invariant_brush_hair.py:20: error: Argument 1 to "brush_hair_all" has incompatible type
"List[Cat]"; expected "List[Mammal]"
invariant_brush_hair.py:20: note: "List" is invariant -- see
http://mypy.readthedocs.io/en/latest/common_issues.html#variance
invariant_brush_hair.py:20: note: Consider using "Sequence" instead, which is covariant
Found 2 errors in 1 file (checked 1 source file)
$
```

As suggested in the [mypy](#) notes above - the `Sequence` type is covariant and as such why it is a better choice than a specific type such as `List`

#### `covariant_brush_hair.py`

```
1#!/usr/bin/env python
2 from animal_kingdom import Move, Mammal, Cat, Dog
3 from typing import List, Sequence
4
5
6 def brush_hair_all(mammals: Sequence[Mammal]) -> None:
7     for mammal in mammals:
8         print(f"The {type(mammal).__name__}'s hair is {mammal.hair}")
9     print()
10
11
12 cats_and_dogs: List[Mammal] = [Cat(Move.walk, "grey"), Dog(Move.run, "shaggy")]
13 brush_hair_all(cats_and_dogs)
14
15 all_cats: List[Cat] = [Cat(Move.run, "orange"), Cat(Move.walk, "scraggly")]
16 brush_hair_all(all_cats)
```

But, suppose a function named `add_cat` was defined that also takes `Sequence[Mammal]` as a parameter.

### `covariant_trap.py`

```

1 #!/usr/bin/env python
2 from animal_kingdom import Move, Mammal, Cat, Dog
3 from typing import List, Sequence
4 import operator
5
6
7 def add_cat(mammals: Sequence[Mammal]) -> None:
8     # mammals.append(Cat(Move.walk, "semi-longhair with a silky coat"))
9     operator.add(mammals, [Cat(Move.walk, "semi-longhair with a silky coat")])
10
11
12 cats_and_dogs: List[Mammal] = [Cat(Move.walk, "grey"), Dog(Move.run, "shaggy")]
13 all_cats: List[Cat] = [Cat(Move.run, "orange"), Cat(Move.walk, "scraggly")]
14 all_dogs: List[Dog] = [Dog(Move.run, "short and curly"),
15                         Dog(Move.walk, "white with black dots")]
16
17 add_cat(cats_and_dogs)
18 add_cat(all_cats)
19 add_cat(all_dogs)

```

The `add_cat` function takes a `Sequence[Mammal]` as a parameter and adds a `Cat` to the sequence, which at first glance seems acceptable.

- The first issue is that while a `List` can be appended to - a `Sequence` does not have an append.
  - ▶ The application gets around this by relying on the `operator.add` function.
    - ◆ This should be an indicator though, that what is being done might not be best practice.

The real problem is that this function can be passed a `List[Dog]` collection, and adding a `Cat` to it would violate its type!

- Some languages handle this at runtime, even though it can be discovered statically.
- The `mypy` tool will not flag this as an issue though it violates the type hinting.

## Contravariant

Consider the following example:

*not\_contravariant.py*

```

1 #!/usr/bin/env python
2 from animal_kingdom import Move, Animal, Mammal, Cat, Dog
3 from typing import List, Sequence
4 import operator
5
6
7 def add_cat(mammals: List[Mammal]) -> None:
8     operator.add(mammals, [Cat(Move.walk, "semi-longhair with a silky coat")])
9
10
11 cats_and_dogs: List[Animal] = [Cat(Move.walk, "grey"), Dog(Move.run, "shaggy")]
12 add_cat(cats_and_dogs)

```

On the other hand, if a `List[Animal]` collection was passed, the `add_cat` function would work perfectly well.

- Note that `Animal` is a supertype of `Mammal` rather than subtype.
  - ▶ That means that the `add_cat` function is **contravariant**

This form of variance can be noted expressly in Python's type-hinting system as **contravariant**.

- Since neither a `List` or a `Sequence` is **contravariant**, `mypy` will flag the calling of the function.

```

python -m mypy --disallow-untyped-defs contravariant.py
contravariant.py:12: error: Argument 1 to "add_cat" has incompatible type "List[Animal]";
expected "List[Mammal]"
Found 1 error in 1 file (checked 1 source file)
$ 

```

# Specifying variance

*contravariant\_types.py*

```

1 #!/usr/bin/env python
2 from animal_kingdom import Move, Animal, Mammal, Cat, Dog
3 from typing import List, Sequence, TypeVar
4
5 T_co = TypeVar('T_co', covariant=True)
6 T_contra = TypeVar('T_contra', contravariant=True)
7
8
9 class ListOrMoreSpecific(List[T_co]):
10    pass
11
12
13 class ListOrLessSpecific(List[T_contra]):
14    pass
15
16
17 def brush_hair_all(mammals: ListOrMoreSpecific[Mammal]) -> None:
18    for mammal in mammals:
19        print(f"The {type(mammal).__name__}'s hair is {mammal.hair}")
20    print()
21
22
23 def add_cat(mammals: ListOrLessSpecific[Cat]) -> None:
24    mammals.append(Cat(Move.walk, "semi-longhair with a soft and silky coat"))
25
26
27 mammals: ListOrMoreSpecific[Mammal] = \
28     ListOrMoreSpecific([Cat(Move.walk, "grey"), Dog(Move.run, "shaggy")])
29
30 animals: ListOrLessSpecific[Animal] = \
31     ListOrLessSpecific([Animal(Move.slither), Cat(Move.walk, "grey"),
32                         Dog(Move.run, "shaggy")])
33
34 brush_hair_all(mammals)
35 add_cat(animals)
```

The default variance of the collections in the `typing` module are **invariant**, which means that only the exact type specified is permitted.

## Union Types

A **Union** type is allowed to be one of a number of possible types.

*unions.py*

```

1 #!/usr/bin/env python
2 from typing import Union
3
4
5 def apartment_info(apartment: Union[int, str]) -> None:
6     prefix = "#" if isinstance(apartment, int) else ""
7     return f'Apartment {prefix}{apartment}'
8
9
10 print(apartment_info(12), apartment_info("F"))

```

*junk.py*

```

1 #!/usr/bin/env python
2 from typing import Union
3
4
5 class Engine:
6     def drain_oil(self):
7         print("Draining Oil")
8
9
10 class Refrigerator:
11     def remove_door(self):
12         print("Removing door for safety reasons")
13
14
15 class ACUnit:
16     def recycle_freon(self):
17         print("Recycling freon")
18
19
20 def destroy(junk: Union[Refrigerator, ACUnit, Engine]) -> None:
21     if isinstance(junk, Refrigerator):
22         junk.remove_door()
23     elif isinstance(junk, Engine):
24         junk.drain_oil()
25     else:
26         junk.recycle_freon()

```

## Optional Types

An important specific case of `Union` types is the `Optional` type.

An `Optional` type is one that is either `None`, or a specified type.

The following example defines a function that filters states on their name.

- Although not shown, the source code contains all 50 states

The return value is one of two types: \* A `Union` of `str` or `List[str]` based on the number of matches found \* `None` if there is no match.

`states.py`

```

1 from typing import Dict, List, Union, Optional
2
3 STATES: Dict[str, str] = {
4     'Alabama': 'Montgomery', 'Alaska': 'Juneau',
5     'Arizona': 'Phoenix', 'Arkansas': 'Little Rock',
6     'Texas': 'Austin', 'Utah': 'Salt Lake City',
7     'Vermont': 'Montpelier', 'Virginia': 'Richmond',
8     'Washington': 'Olympia', 'West Virginia': 'Charleston',
9     'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'
10 }
11
12
13 Match = Union[str, List[str]]
14
15
16 def states_that_with(letters: str) -> Optional[Match]:
17     result = [state for state in STATES.keys() if state.startswith(letters)]
18     if not result:
19         return None
20     elif len(result) == 1:
21         return result[0]
22     else:
23         return result

```

With Python's support for exceptions, the following may seem unusual.

- But, there are cases where a value may be present, or absent.
- The `Optional` type is excellent for type-checking these cases, as `mypy` will detect if the wrapped type is being used without a branch for checking the `None` possibility.

*annoyed.py*

```
1 from typing import Optional
2
3
4 def annoy_cat(times: Optional[int]) -> str:
5     return 'meow' * times
```

```
$ python -m mypy annoyed.py
annoyed.py:5: error: Unsupported operand types for * ("str" and "None")
annoyed.py:5: note: Right operand is of type "Optional[int]"
Found 1 error in 1 file (checked 1 source file)
$
```

This allows for dealing with detectable default values in a type-safe way.

*type\_safe\_annoyed.py*

```
1 from typing import Optional
2
3
4 def annoy_cat(times: Optional[int] = None) -> str:
5     if times is None:
6         return 'meow'
7     else:
8         return 'meow' * times
9
10
11 print(annoy_cat())
12 print(annoy_cat(4))
```

## functools.singledispatch

Many programming languages have the ability to pick a specific function or method based on its arguments' types.

- Baked into the standard library is the `functools.singledispatch` decorator.
  - ▶ This involves decorating a function, and then defining multiple implementations with different types for the `first` argument.

*single\_dispatch.py*

```

1 import functools
2
3
4 class Twistable:
5     pass
6
7
8 class Person:
9     pass
10
11
12 @functools.singledispatch
13 def wound(x):
14     print(f'The {x} feels that hurts')
15
16
17 @wound.register
18 def _(x: Twistable):
19     print(f'Wind the {type(x).__name__}')
20
21
22 @wound.register
23 def anything(x: Person):
24     print(f'{type(x).__name__} needs some time off to relax')
25
26
27 wound("Enemy")
28 wound(Twistable())
29 wound(Person())

```

All additional registered functions must **not** share the same function or method name as the originally decorated function, unlike the `@property` decorator.

The functions are typed **only** on the basis of the first argument, differently-typed later arguments, will not be used by `@singledispatch`.

## multimethod

`multimethod` is a third-party module that provides even greater support for multiple dispatch functions. It provides a decorator, `@multimethod`, that takes into account the types of all function arguments.

```
https://pypi.org/project/multimethod/
```

The following shows using `pip` to install `multimethod`.

```
$ pip install multimethod
Collecting multimethod
  Downloading multimethod-1.4-py2.py3-none-any.whl (7.3 kB)
Installing collected packages: multimethod
Successfully installed multimethod-1.4
$
```

*multi\_method\_example.py*

```
1 from multimethod import multimethod, DispatchError
2 from some_dataclasses import Location, Music, Fish
3
4
5 @multimethod
6 def drop_bass(loc: Location, m: Music):
7     print('WUBWUBWUBWUBWUB WUB DRRRRRR')
8
9
10 @multimethod
11 def drop_bass(loc: Location, f: Fish):
12     print('Splloosh')
13
14
15 drop_bass(Location("Band Practice"), Music("Classic Rock"))
16 drop_bass(Location("State Park"), Fish("Trout"))
17
18 try:
19     drop_bass("This should", "Not be allowed")
20 except DispatchError as de:
21     print("Dispatch Error:", de)
```

Alternatively, functions can be explicitly registered in the same style as `functools.singledispatch`.

- This makes them compatible with `mypy`, which by default checks that each name is defined once.

# Stub Type Hinting

While type hinting can be a useful tool, it can be limiting when integrating with other source code.

- If a third-party library does not use type hints, then the type errors generated by that module may not be correctable in the current project.

The `mypy` utility can actually read a separate file to find out the type interface of a different module.

- One way to accomplish this is to write a stub file for the library (or an arbitrary module) and store it as a `.pyi` file in the same directory as the library module.
  - ▶ Then fill the Python interface file with an outline of the public types of the module to annotate.

The following examples are in a subdirectory named `stub_example`.

The function below contains no type hinting.

`stub_example/third_party_code.py`

```
1 # No type hinting exists in this module
2 def take_an_int(a):
3     return a * 20
```

The file below is an interface with the same name as the module above, but has a `.pyi` extension.

- The ellipses are **literal** ellipses used by `mypy` to allow Python to parse the interface without running any code.

`stub_example/third_party_code.pyi`

```
1 def take_an_int(a: int) -> int:...
```

Now, `mypy` is able to parse this interface file to determine the correct annotated types, and can perform static analysis on code using the referenced module.

This can be seen in the simple application below.

`stub_example/the_app.py`

```
1 import third_party_code
2
3 third_party_code.take_an_int(99)
4 third_party_code.take_an_int("Hello")
```

Running the `mypy` utility with `the_app.py` shows that `mypy` flags the attempt to pass a `str` where an `int` is expected.

```
$ python -m mypy the_app.py
the_app.py:4: error: Argument 1 to "take_an_int" has incompatible type "str"; expected
"int"
Found 1 error in 1 file (checked 1 source file)
$
```

More information about the use of stubs can be found at the following URL:

<https://mypy.readthedocs.io/en/stable/stubs.html>

# Exercises

## Exercise 1

Write type hints for the following code, rewriting the code as necessary:

- The code can be found in the `starter_code` subdirectory of the `examples`

`./starter_code/grep_sed_starter.py`

```

1 def grep(needle):
2     '''Only emits lines of text that contain x'''
3     haystack = None
4     while True:
5         haystack = yield haystack
6
7         if needle not in haystack:
8             haystack = None
9
10
11 def sed(pattern, replacement):
12     '''Replace any lines containing pattern with replacement'''
13     result = None
14     while True:
15         result = yield result
16
17         if pattern in result:
18             result = replacement
19
20
21 search = grep('foo')
22 replace = sed('bar', 'baz')
23 next(search)
24 next(replace)
25
26 for line in file:
27     found = search.send(line)
28     if found:
29         line = replace.send(line)
30
31     print(line)

```

## Exercise 2 (`sower.py`)

Write an overloaded function `sow` that behaves differently when it is passed a `Pig` object versus a `Seed` object.

## Exercise 3 (add\_42.py)

Write and annotate a function named `append_42` that appends a `42` to the parameter, a list.

- If no list is supplied, a new one should be created and returned.

## Exercise 4

Write the correct supports in the type-hinting system to support a function `add_mammal(x)` that takes a list of `Mammals` or `Animals` and adds a new `Mammal` to the list, returning nothing.

- Passing a `List[Cat]` should produce an error (since adding a `Mammal` would violate the list's type).

# Chapter 8. Functional Tools

## Objectives

- Conceptualize higher-order functions
- Learn what's in functools and itertools
- Transform data with map/reduce
- Chain multiple iterables together
- Implement function overloading with single dispatch
- Use iterative tools to work with iterators
- Chain multiple iterators together
- Compute combinations and permutations
- Zip multiple iterables with default values

## Higher-order functions

Higher-order functions operate on or return functions.

- The most traditional higher-order functions in other languages are map and reduce.

These are used for functional programming, which prevents side effects (modifying data outside the function).

- Some languages make it difficult or impossible to pass functions into other functions, but in Python, since functions are first-class objects, it is easy.

Higher-order functions may be nested, to have multiple transformations on data.

# Lambda functions

A lambda function is an inline anonymous function declaration.

- It evaluates as a function object, in the same way that def function(...): ... does.

A lambda declaration has only parameters and the return value.

- Blocks are not allowed, nor is the return keyword.

Lambdas are useful as predicates (callbacks) in higher-order functions.

*higher\_order\_functions.py*

```

1 #!/usr/bin/env python
2
3
4 def main():
5     fruits = ["pomegranate", "cherry", "apricot", "date", "apple", "lemon",
6               "kiwi", "orange", "lime", "watermelon", "guava", "papaya", "fig",
7               "pear", "banana", "tamarind", "persimmon", "elderberry", "peach"]
8     double = "\n" * 2
9
10    # Pass str.upper as the callback
11    print(process_list(fruits, str.upper), end=double)
12
13    # Pass a lambda function as the callback
14    print(process_list(fruits, lambda s: s[0].upper()), end=double)
15
16    # Pass builtin function len() as the callback
17    print(process_list(fruits, len), end=double)
18
19    # Pass the result of process_list() to sum() to sum all the values
20    print(sum(process_list(fruits, len)))
21
22
23 def process_list(alist, func): # Accepts list and a callback function
24     # Call the callback function on each item as part of list comprehension
25     return [func(item) for item in alist]
26
27
28 if __name__ == '__main__':
29     main()

```

## The operator module

The operator module provides functional versions of Python's standard operators.

- This saves the trouble of creating trivial lambda functions.

Instead of

```
lambda x, y: x + y
```

You can just use

```
operator.add
```

Both of these functions take two operators and add them.

## The `functools` module

The `functools` module provides higher-order functions.

- It provides tools for higher-ordering programming, which means functions that operate on, or return functions (some do both).

Functional programming avoids explicit loops.

- `map()` and `reduce()` are two builtin functions that are the basis of many functional algorithms.
  - ▶ In Python 3, they are also available via `functools`.

Most of these functional algorithms can be implemented with list comprehensions or generator expressions.

## map()

`map()` creates a `map` object by applying a function to every element of an iterable.

- `map(func, list)` returns `[func(list[0]), func(list[1]), func(list[2], ...)]`.
  - ▶ The first argument to `map()` is a function that takes one argument.
  - ▶ Each element of the iterable is passed to the function, and the return value is added to the result list.

*using\_map.py*

```

1 #!/usr/bin/env python
2
3
4 def main():
5     strings = ['wombat', 'koala', 'kookaburra', 'blue-ringed octopus']
6
7     print(list(map(str.upper, strings))) # Map list to upper case
8     print(list(map(len, strings))) # Map to list of string lengths
9
10    # Using a list comprehension, which is usually simpler than map()
11    print([s.upper() for s in strings])
12    print("=" * 30)
13    fancy_mapping()
14
15
16 def fancy_mapping():
17     tests = {"Sally": (89, 78, 99, 88, 92, 98, 95, 78, 88),
18              "Doug": (68, 87, 72, 60, 80, 65),
19              "Kesha": (98, 87, 99, 78, 99, 80, 98, 50),
20              "John": (89, 78, 99, 88, 92, 99, 95, 88, 95, 99)}
21
22    # Find the average of the first test taken by each student,
23    # followed by the second test taken by each, ... to the length
24    # of the shortest iterable passed to map()
25    x = map(averages, *tests.values())
26    print("Averages:", list(x))
27
28 def averages(*grades):
29     qty = len(grades)
30     return sum(grades)/qty
31
32
33 if __name__ == '__main__':
34     main()

```

## reduce()

`reduce()` returns the value created by applying a function to every element of a list and the previous function's result.

- `reduce(func, list)` returns `func(list[n], func(list[n-1], func(list[2]…func(list[0]))))`.
- A third argument to `reduce()` can be used to provide an initial value.
  - ▶ By default, the initial value is the first element of the list.

Other functions such as `sum()` or `str.join()` can be defined in terms of `map()` or `reduce()`.

The mapreduce approach to massively parallel processing, such as Hadoop, was inspired by `map()` and `reduce()`.

*using\_reduce.py*

```

1 #!/usr/bin/env python
2 from operator import add, mul
3 from functools import reduce
4
5
6 def main():
7     values = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
8     strings = ['fi', 'fi', 'fo', 'fum']
9
10    # continually reduce by adding to initial value of first element
11    print("result is", reduce(add, values))
12
13    # continually reduce by adding to initial value of 1000
14    print("result is", reduce(add, values, 1000))
15
16    # continually reduce by multiplying by initial value of 1
17    print("result is", reduce(mul, values, 1))
18
19    # continually reduce by adding to initial string of ""
20    print("result is", reduce(add, strings, ""))
21
22    # continually reduce by adding uppercase to initial string of ""
23    result = reduce(add, list(map(str.upper, strings)), "")
24    print("result is", result)
25
26
27 if __name__ == '__main__':
28     main()

```

## Partial functions

Partial functions are wrappers that have some arguments already filled in for the "real" function. \* This is especially useful when creating callback functions. \* It is also nice for creating customized functions that rely on functions from the standard library.

While you can create partial functions by hand, using closures, the `partial()` function simplifies creating such a function.

The arguments to `partial()` are the function and one or more argument.

- It returns a new function object, which will call the specified function and pass in the provided argument.

*partial\_functions.py*

```

1 #!/usr/bin/env python
2 import re
3 from functools import partial
4
5
6 def main():
7     # create partial function that "preloads" range() with arguments 0 and 25
8     count_by = partial(range, 0, 25)
9
10    # call partial function with parameter, 0 and 25 automatically passed in
11    print(list(count_by(1)), list(count_by(3)), list(count_by(5)), sep='\n')
12
13    # create partial function that embeds pattern in re.search()
14    has_a_number = partial(re.search, r'\d+')
15
16    strings = ['abc', '123', 'abc123', 'turn it up to 11', 'blah blah']
17
18    for s in strings:
19        print("{}:{}".format(s), end=' ')
20        if has_a_number(s): # call re.search() with specified pattern
21            print("YES")
22        else:
23            print("NO")
24
25
26 if __name__ == '__main__':
27     main()

```

# Single dispatch

The `singledispatch` module provides generic functions.

- A generic function is defined once with the `@singledispatch` decorator, then other functions may be registered to it.

To register a function, decorate it with `@original_function.register(type)`.

- Then, when the original function is passed an argument of type `type` as its first parameter, it will call the registered function instead.
  - ▶ If no registered functions are found for the type, it will call the original function.

This can of course, be done manually by checking argument types, then calling other methods, but using `singledispatch` is less cumbersome.

`singledispatch` may not be used with class methods, although some developers have worked around that limitation:

To check which function would be chosen for a given type, use the `dispatch()` method of the original function. To get a list of all registered functions, use the `registry` attribute.

The `singledispatch` module is part of the standard library beginning with 3.4.

```
1 #!/usr/bin/env python
2 from functools import singledispatch
3 from io import TextIOWrapper
4
5
6 def main():
7     mary_in = open('../data/mary.txt') # open a file and get a file object
8     for x in mary_in, '../data/mary.txt', b'../data/mary.txt', 52:
9         try:
10             # correct single dispatch handler will automatically be called
11             with xopen(x) as file_in:
12                 result = file_in.read()
13                 print(f"Length: {len(result)}")
14             except TypeError as err:
15                 print('ERROR:', err)
16
17             print('-' * 60)
18             print(xopen.dispatch(str), "\n") # show handler function for str
19             for arg_type, func in xopen.registry.items():
20                 print(arg_type, func) # show functions for each registered type
21
22
23 @singledispatch
24 def xopen(source, mode="r"): # generic function that is actually called
25     raise TypeError("Invalid arg: must be file, str, or bytes, not",
26                     type(source).__name__)
27
28
29 @xopen.register(TextIOWrapper) # handler for text files
30 def xopen_file(fileobj):
31     return fileobj
32
33
34 @xopen.register(str) # handler for string type
35 def xopen_str(str, mode="r"):
36     return open(str, mode)
37
38
39 @xopen.register(bytes) # handler for bytes type
40 def xopen_bytes(bytes, mode="r"):
41     return open(bytes.decode(), mode)
42
43
44 if __name__ == '__main__':
45     main()
```

## The `itertools` module

The `itertools` module provides many different iterators.

- Some of the tools work on existing iterators, while others create them from scratch.
- These tools work well with functions from the `operator` module, as well as interacting with the functional tools described earlier.

Many of the functions in `itertools` were inspired by Haskell, SML, and APL.

The documentation for the `itertools` module includes a group of utility functions (recipes) for extending the toolset.

- The `itertools` recipes that are documented can be found at the following URL:

<https://docs.python.org/3/library/itertools.html#itertools-recipes>

## Infinite iterators

Infinite iterators return iterators that will iterate a specified number of times, or infinitely.

- These iterators are similar to generators; they do not keep all the values in memory.

`islice()` selects a slice of an iterator.

- You can specify an iterable and up to 3 slice arguments – start, stop, and increment, similar to the slice arguments of a list.
  - ▶ If just stop is provided, it stops the iterator after than many values.

`count()` is similar to `range()`.

- It provides a sequence of numbers with a specified increment.
  - ▶ The big difference is that there is no end condition, so it will increment forever.
  - ▶ You will need to test the values and stop, or use `islice()`.

`cycle()` loops over an iterable repeatedly, going back to the beginning each time the end is reached.

`repeat()` repeats a value infinitely, or a specified number of times.

*infinite\_iterators.py*

```
1 #!/usr/bin/env python
2 from itertools import islice, count, cycle, repeat
3
4
5 def main():
6     for i in count(0, 10):           # count by tens starting at 0 forever
7         if i > 50:
8             break                    # without a check, will never stop
9         print(i, end=' ')
10    print("\n")
11
12    # saner, using islice to get just the first 6 results
13    for i in islice(count(0, 10), 6):
14        print(i, end=' ')
15    print("\n")
16
17    giant = ['fee', 'fi', 'fo', 'fum']
18
19    # cycle over values in list forever (use islice to stop)
20    for i in islice(cycle(giant), 10):
21        print(i, end=' ')
22    print("\n")
23
24    # repeat value 10 times (default is repeat forever)
25    for i in repeat('tick', 10):
26        print(i, end=' ')
27    print("\n")
28
29
30 if __name__ == '__main__':
31     main()
```

## Extended iteration

Another group of iterator functions provides extended iteration.

`chain()` takes two or more iterables, and treats them as a single iterable.

- To chain the elements of a single iterable together, use `chain.from_iterable()`.

`dropwhile()` skips leading elements of an iterable until some condition is reached.

`takewhile()` stops iterating when some condition is reached.

## extended\_iteration.py

```

1 #!/usr/bin/env python
2 from itertools import chain, takewhile, dropwhile
3
4
5 def main():
6     spam, ham = ['alpha', 'beta', 'gamma'], ['delta', 'epsilon', 'zeta']
7     # treat spam and ham as a single iterable
8     print_iterable(chain(spam, ham))
9
10    eggs = [spam, ham]
11    # treat all elements of eggs as a single iterable
12    print_iterable(chain.from_iterable(eggs))
13
14    fruits = ["pomegranate", "cherry", "apricot", "date", "apple", "lemon",
15              "kiwi", "orange", "lime", "watermelon", "guava", "papaya", "fig",
16              "pear", "banana", "tamarind", "persimmon", "elderberry", "peach"]
17    # iterate over elements of fruits as long as length of current item > 4
18    print_iterable(takewhile(lambda f: len(f) > 4, fruits))
19
20    # iterate over elements of fruits as long as fruit does not start with 'k'
21    print_iterable(takewhile(lambda f: f[0] != 'k', fruits))
22
23    values = [5, 18, 22, 31, 44, 57, 59, 61, 66, 70, 72, 78, 90, 99]
24    # skip over elements as long as value is < 50, then iterate over rest
25    print_iterable(dropwhile(lambda f: f < 50, values))
26
27
28 def print_iterable(iter_obj): # helper function to print each iterable
29     print(*iter_obj, sep=' ', end="\n\n")
30
31
32 if __name__ == '__main__':
33     main()

```

# Grouping

The `groupby()` function groups consecutive elements of an iterable by value.

- As with `sorted()`, the value may be determined by a key function.
  - ▶ This is similar to sort -u or the uniq commands in Linux.

`groupby()` returns an iterable of subgroups, which can then be converted to a list or iterated over. \* Each subgroup has a key, which is the common value, and an iterable of the values for that key.

`groupby_examples.py`

```

1 #!/usr/bin/env python
2 from itertools import groupby
3
4
5 def main():
6     with open('../data/words.txt') as words_in: # open file for reading
7         # create generator of all words, stripped of the trailing '\n'
8         all_words = (w.rstrip() for w in words_in)
9
10    # create a groupby() object where key is first character in the word
11    g = groupby(all_words, key=lambda e: e[0])
12
13    # Dictionary where the key is the first character, and the value
14    # is the number of words that start with that character; groupby groups
15    # all the words, then len counts the number of words for that character
16    counts = {letter: len(list(wlist)) for letter, wlist in g}
17
18    # sort the counts dictionary by value (i.e., number of words, not the
19    # letter itself) into a list of tuples
20    letters = sorted(counts.items(), key=lambda e: e[1], reverse=True)
21
22    # loop over the list of tuples and print the letter and its count
23    for letter, count in letters:
24        print(letter, count)
25
26    # sum all the individual counts and print the result
27    print("\nTotal words counted:", sum(counts.values()))
28
29
30 if __name__ == '__main__':
31     main()

```

A more real-life example of grouping is shown on the following page:

## group\_dates\_by\_week.py

```

1 #!/usr/bin/env python
2 from datetime import date as Date
3 from random import randint
4 from itertools import groupby
5
6 # number of days per month, for generating random dates
7 date_details = [(1, 31), (2, 28), (3, 31), (4, 30), (5, 31), (6, 30), (7, 31),
8                 (8, 31), (9, 30), (10, 31), (11, 30), (12, 31)]
9
10
11 def main():
12     sample_dates = generate_sample_dates()
13     display_groups(sample_dates)
14
15
16 def generate_sample_dates():
17     """
18     Generate sorted list of random dates.
19     @return: sorted list of datetime.date objects
20     """
21     dates = []
22     for month, days in date_details:
23         for i in range(1, days + 1):
24             for _ in range(randint(1, 3)):
25                 date = Date(2014, month, i)
26                 dates.append(date)
27     # print(dates) # uncomment to see raw date list
28     return dates
29
30
31 def display_groups(dates):
32     """
33     Display dates, grouped by ISO week #
34     DATE.isocalendar() returns ISO year, ISO week number, ISO weekday tuple.
35     @param dates: a sorted list of Python datetime.date objects
36     @return: None
37     """
38     for week_number, date_list in groupby(dates, lambda e: e.isocalendar()[1]):
39         print("Week:", week_number)
40         for dl in date_list:
41             print("\t" + str(dl))
42
43
44 if __name__ == '__main__':
45     main()

```

# Combinatoric generators

Several functions provide products, combinations, and permutations.

`product()` return an iterator with the Cartesian product of two iterables.

`combinations()` returns the unique n-length combinations of an iterator.

`permutations()` returns all n-length sub-sequences of an iterator.

- If the length is not specified, all sub-sequences are returned.

*combinations\_permutations.py*

```

1 #!/usr/bin/env python
2 from itertools import product, permutations, combinations
3
4
5 def main():
6     SUITS = 'CDHS'
7     RANKS = '2 3 4 5 6 7 8 9 10 J Q K A'.split()
8
9     # Cartesian product
10    # (match every item in one list to every item in the other list)
11    cards = product(SUITS, RANKS)
12    print(list(cards), '\n')
13
14    # reverse order and concatenate elements using list comprehension
15    cards = [f"{suit}{rank}" for rank, suit in product(SUITS, RANKS)]
16    print(cards, '\n')
17
18    giant = ['fee', 'fi', 'fo', 'fum']
19
20    # all distinct combinations of 4 items taken 2 at a time
21    result = combinations(giant, 2)
22    print(list(result), "\n")
23
24    # all distinct permutations of 4 items taken 2 at a time
25    result = permutations(giant, 2)
26    print(list(result), "\n")
27
28
29 if __name__ == '__main__':
30     main()

```

# Exercises

## Exercise 1 (sum\_of\_values.py)

Read in the data from float\_values.txt and print out the sum of all values. Do this with functional tools – there should be no explicit loops in your code.

**TIP** use reduce() + operator.add on the file object.

## Exercise 2 (pres\_by\_state\_functional.py)

Using presidents.txt, print out a list of the number of presidents from each state.

**TIP** Use map() + lambda to split lines from presidents.txt on the 7th field, then use groupby() on that.

## Exercise 3 (count\_all\_lines.py)

Count all of the lines in all the files specified on the command line without using any loops.

**TIP** use map() + chain.from\_iterable() to create an iterable of all the lines, then use reduce to count them.



# Chapter 9. Metaprogramming

## Objectives

- Learn what metaprogramming means
- Access local and global variables by name
- Inspect the details of any object
- Use attribute functions to manipulate an object
- Design decorators for classes and functions
- Define classes with the type() function
- Create metaclasses
- Use descriptors to simplify repetitive code

# Metaprogramming

Metaprogramming is writing code that generates or modifies other code.

- It includes fetching, changing, or deleting attributes, and writing functions that return functions (AKA factories).

Metaprogramming is easier in Python than many other languages.

- Python provides explicit access to objects, even the parts that are hidden or restricted in other languages.

For instance, you can easily replace one method with another in a Python class, or even in an object instance.

- In Java, this would be deep magic requiring many lines of code.

## globals() and locals()

The `globals()`` builtin function returns a dictionary of all global objects.

- The keys are the object names, and the values are the objects values.
- The dictionary is "live"—changes to the dictionary affect global variables.

The `locals()` builtin function returns a dictionary of all objects in local scope.

*globals\_locals.py*

```

1 #!/usr/bin/env python
2 from pprint import pprint      # import prettyprint function
3
4 spam = 42                      # global variables
5 ham = 'Smithfield'
6
7
8 def main():
9     eggs('mango')
10
11
12 def eggs(fruit):               # function parameters are always local
13     name = 'Lancelot'          # local variable
14     idiom = 'swashbuckling'    # local variable
15     print("Globals:")
16     pprint(globals())         # globals() returns dict of all globals
17     print()
18     print("Locals:")
19     pprint(locals())          # locals() returns dict of all locals
20
21
22 if __name__ == '__main__':
23     main()

```

## The `inspect` module

The `inspect` module provides user-friendly functions for accessing Python metadata.

*inspect\_ex.py*

```

1 #!/usr/bin/env python
2 import inspect
3
4
5 def main():
6     fmt = "{:>10}: {:10}{:10}{:10}{:10}"
7     print(fmt.format("Reference", "Module?", "Function?", "Class?", "Method?"))
8     for thing in (inspect, Spam, Spam().eggs, ham):
9         print(fmt.format(thing.__name__,
10                         str(inspect.ismodule(thing)),      # test for module
11                         str(inspect.isfunction(thing)),    # test for function
12                         str(inspect.isclass(thing)),      # test for class
13                         str(inspect.ismethod(thing))))   # test for method
14
15     # get argument specifications for a function
16     argspec = inspect.getfullargspec(ham)
17     print("\nFunction spec for ham:", argspec)
18
19     # get frame (function call stack) info
20     frame = inspect.getframeinfo(inspect.currentframe())
21     print("\nCurrent frame:", frame)
22
23
24 class Spam:                                # defines a class
25     def eggs(self):
26         pass
27
28
29 def ham(p1, p2='a', *p3, p4, p5='b', **p6): # define a function
30     print(p1, p2, p3, p4, p5, p6)
31
32
33 if __name__ == '__main__':
34     main()

```

A summary of the various functions available in the `inspect` module are shown on the following page. <<<

Table 7. `inspect` module convenience functions

Function(s)	Description
<code>ismodule()</code> , <code>isclass()</code> , <code>ismethod()</code> , <code>isfunction()</code> , <code>isgeneratorfunction()</code> , <code>isgenerator()</code> , <code>istraceback()</code> , <code>isframe()</code> , <code>iscode()</code> , <code>isbuiltin()</code> , <code>isroutine()</code>	check object types
<code>getmembers()</code>	get members of an object that satisfy a given condition
<code>getfile()</code> , <code>getsourcefile()</code> , <code>getsource()</code>	find an object's source code
<code>getdoc()</code> , <code>getcomments()</code>	get documentation on an object
<code>getmodule()</code>	determine the module that an object came from
<code>getclasstree()</code>	arrange classes so as to represent their hierarchy
<code>getargspec()</code> , <code>getargvalues()</code>	get info about function arguments
<code>formatargspec()</code> , <code>formatargvalues()</code>	format an argument spec
<code>getouterframes()</code> , <code>getinnerframes()</code>	get info about frames
<code>currentframe()</code>	get the current stack frame
<code>stack()</code> , <code>trace()</code>	get info about frames on the stack or in a traceback

# Working with attributes

All Python objects are essentially dictionaries of attributes.

- There are four special builtin functions for managing attributes.
  - ▶ These may be used to programmatically access attributes when you have the name as a string.

`getattr()` returns the value of a specified attribute, or `None` if the object does not have that attribute.

`hasattr()` returns the value of a specified attribute, or `None` if the object does not have that attribute.

`setattr()` sets an attribute to a specified value.

`delattr()` deletes an attribute and its corresponding value.

*attributes.py*

```

1 #!/usr/bin/env python
2
3
4 def main():
5     s = Spam()
6     s.eggs("fried")
7     print("hasattr()", hasattr(s, 'eggs')) # check whether attribute exists
8
9     e = getattr(s, 'eggs')                # retrieve attribute
10    e("scrambled")
11    setattr(Spam, 'eggs', toast)          # set (or overwrite) attribute
12    s.eggs("buttered!")
13    delattr(Spam, 'eggs')                # remove attribute
14
15
16 class Spam():
17
18     def eggs(self, msg):
19         print("eggs!", msg)
20
21
22     def toast(self, msg):
23         print("toast!", msg)
24
25
26 if __name__ == '__main__':
27     main()

```

## Adding instance methods

Using `setattr()`, it is easy to add instance methods to classes.

- Just add a function object to the class.
- Because it is part of the class itself, it will automatically be bound to the instance.
  - ▶ Remember that an instance method expects `self` as the first parameter.
  - ▶ In fact, this is the meaning of a bound instance—it is "bound" to the instance, and therefore when called, it is passed the instance as the first parameter.
- Once added, the method may be called from any existing or new instance of the class.

To add an instance method to an instance takes a little more effort.

- Because it's not being added to the class, it is not automatically bound.
- The function needs to know what instance it should be bound to.
  - ▶ This can be accomplished with the `types.MethodType()` function.

```
1 #!/usr/bin/env python
2 from types import MethodType
3
4
5 def main():
6     d1 = Dog()                      # Create instance of Dog
7     setattr(Dog, "bark", bark)      # Binds function as an instance method
8     d2 = Dog()                      # Define another instance of Dog
9     d1.bark()                      # New method can be called from either instance
10    d2.bark()
11
12    # Add function to instance after passing it through MethodType()
13    setattr(d1, "wag", MethodType(wag, d1))
14    d1.wag()           # Call instance method
15    try:
16        d2.wag() # Instance method not available - only bound to d1
17    except AttributeError as err:
18        print(err)
19
20
21 class Dog(): # Define Dog type
22     pass
23
24
25 def bark(self): # Define (unbound) function
26     print("Woof! woof!")
27
28
29 def wag(self): # Create another unbound function
30     print("Wagging...")
31
32
33 if __name__ == '__main__':
34     main()
```

# Decorators

In Python, many decorators are provided by the standard library, such as `property()` or `classmethod()`.

A decorator is a component that modifies some other component.

- The purpose is typically to add functionality, but there are no real restrictions on what a decorator can do.

Many decorators register a component with some other component.

- For instance, the `@app.route()` decorator in Flask maps a URL to a view function.
- As another example, the `unittest` module provides decorators to skip tests.

A very common decorator is `@property`, which converts a class method into a `property` object.

A decorator can be any callable, which means it can be a normal function, a class method, or an instance of a class which implements the `__call__()` method (AKA callable class).

A simple decorator expects the item being decorated as its parameter, and returns a replacement.

- Typically, the replacement is a new function, but there is no restriction on what is returned.

If the decorator itself needs parameters, then the decorator returns a wrapper function.

- The wrapper function expects the item being decorated, and then returns the replacement.

The `@`` sign is used to apply a decorator to a function or class.

- A decorator only applies to the next definition in the script.

The example below defines a function that can be used as a decorator function.

*spam\_decorator.py*

```
1 #!/usr/bin/env python
2 def spam(old_func): # Wraps one function in another and returns it
3     def new_func():
4         return f"{old_func()}\nand eggs"
5     return new_func
```

The example on the next page shows how the above function can be called with and without a decorator.

The example below uses the `@spam` decorator to wrap the `ham()` function call

*with\_decorator.py*

```
1 #!/usr/bin/env python
2 from spam_decorator import spam
3
4
5 def main():
6     print(ham())
7
8
9 @spam # The spam decorator automatically wraps ham()
10 def ham():
11     return "Ham"
12
13
14 if __name__ == '__main__':
15     main()
```

The following syntax accomplishes the same result without the use of the decorator.

*without\_decorator.py*

```
1 #!/usr/bin/env python
2 from spam_decorator import spam
3
4
5 def main():
6     global ham
7     print(ham())
8     ham = spam(ham)
9     print(ham())
10
11
12 def ham():
13     return "Ham"
14
15
16 if __name__ == '__main__':
17     main()
```

The example on the following page demonstrates adding arguments to the mix.

*spam\_decorator\_with\_args.py*

```
1 #!/usr/bin/env python
2 def spam(a, b):          # Decorator function takes parameters
3     def wrapper(old_func): # Wrapper function wraps original function
4         def new_func():    # Typically invokes the original function
5             return f"{a} * {old_func()} and {b} eggs"
6         return new_func
7     return wrapper
```

*arguments\_with\_decorator.py*

```
1 #!/usr/bin/env python
2 from spam_decorator_with_args import spam
3
4
5 def main():
6     print(ham())
7
8
9 @spam(2, 3)
10 def ham():
11     return "Ham"
12
13
14 if __name__ == '__main__':
15     main()
```

```
1 #!/usr/bin/env python
2 from spam_decorator_with_args import spam
3
4
5 def main():
6     global ham
7     print(ham())
8     ham = spam(2, 3)(ham)
9     print(ham())
10
11
12 def ham():
13     return "Ham"
14
15
16 if __name__ == '__main__':
17     main()
```

Table 8. Decorators in the standard library

Decorator	Description
<code>@abc.abstractmethod</code>	Indicate abstract method (must be implemented).
<code>@abc.abstractproperty</code>	Indicate abstract property (must be implemented).
<code>@atexit.register</code>	Register function to be executed when interpreter (script) exits.
<code>@classmethod</code>	Indicate class method (receives class object, not instance object)
<code>@contextlib.contextmanager</code>	Define generator for <code>with</code> statement context managers (no need for <code>__enter__</code> / <code>__exit__</code> ).
<code>@functools.lru_cache</code>	Wrap a function with a memoizing callable
<code>@functools.singledispatch</code>	Transform function into a single-dispatch generic function.
<code>@functools.total_ordering</code>	Supply all other comparison methods if class defines at least one.
<code>@functools.wraps</code>	Invoke <code>update_wrapper()</code> so decorator's replacement function keeps original function's properties.
<code>@property</code>	Indicate a class property.
<code>@staticmethod</code>	Indicate static method (passed neither instance nor class object).
<code>@types.coroutine</code>	Transform generator function into a coroutine function.
<code>@unittest.mock.patch</code>	Patch target with a new object. When the function/ <code>with</code> statement exits patch is undone.
<code>@unittest.mock.patch.dict</code>	Patch dictionary (or dictionary-like object), then restore to original state after test.
<code>@unittest.mock.patch.multiple</code>	Perform multiple patches in one call.
<code>@unittest.mock.patch.object</code>	Patch object attribute with mock object.
<code>@unittest.skip()</code>	Skip test unconditionally
<code>@unittest.skipIf()</code>	Skip test if condition is true
<code>@unittest.skipUnless()</code>	Skip test unless condition is true
<code>@unittest.expectedFailure()</code>	Mark Test as expected failure
<code>@unittest.removeHandler()</code>	Remove Control-C handler

## Trivial Decorator

A decorator does not have to be elaborate.

- It can return anything, though typically decorators return the same type of object they are decorating.

In the example below, the decorator returns the integer value 42.

- This is not particularly useful, but illustrates that the decorator always replaces the object being decorated with *something*.

`deco_trivial.py`

```

1 #!/usr/bin/env python
2
3
4 def main():
5     name = "Guido"
6     x = void(name)      # Calling void directly not actually passing a function
7     print(x, type(x))  # but void() does not care it always returns 42
8
9     print(hello, type(hello)) # hello is now the integer 42, not a function
10
11
12 def void(old_function):
13     return 42           # replace function with 42
14
15
16 @void                  # decorate hello() function
17 def hello():
18     print("Hello, world")
19
20
21 if __name__ == '__main__':
22     main()

```

## Decorator functions

A decorator function acts as a wrapper around some object (usually function or class).

- It allows you to add features to a function without changing the function itself.
  - ▶ For instance, the `@property`, `@classmethod`, and `@staticmethod` decorators are used in classes.

A decorator function expects only one argument—the function to be modified.

- It should return a new function, which will replace the original.
  - ▶ The replacement function typically calls the original function as well as some new code.
  - ▶ The new function should be defined with generic arguments (`*args`, `**kwargs`) so it can handle the original function's arguments.
- The `wraps` decorator from the `functools` module in the standard library should be used with the function that returns the replacement function.
  - ▶ This makes sure the replacement function keeps the same properties (especially the name and docstring) as the original (target) function.
  - ▶ Otherwise, the replacement function keeps all of its own attributes.

```
1 #!/usr/bin/env python
2 from functools import wraps
3
4
5 def main():
6     hello('hello', 'world') # call new function
7     print()
8
9     hello('hi', 'Earth')
10    print()
11
12    hello('greetings')
13
14
15 # decorator function -- expects decorated (original) function as a parameter
16 def debugger(old_func):
17
18     @wraps(old_func) # preserves name of original function after decoration
19     def new_func(*args, **kwargs): # replacement; with generic params
20         print("*" * 40) # new functionality
21         print("** function", old_func.__name__, "**") # new functionality
22
23         if args: # new functionality
24             print("\targs are ", args)
25         if kwargs: # new functionality
26             print("\tkwargs are ", kwargs)
27
28         print("*" * 40) # new functionality
29
30         return old_func(*args, **kwargs) # call the original function
31
32     return new_func # return the new function object
33
34
35 @debugger # apply the decorator to a function
36 def hello(greeting, whom='world'):
37     print("{} , {} ".format(greeting, whom))
38
39
40 if __name__ == '__main__':
41     main()
```

## Decorator Classes

A class can also be used to implement a decorator.

- The advantage of using a class for a decorator is that a class can keep state, so that the replacement function can update information stored at the class level.
- Implementation depends on whether the decorator needs parameters.

If the decorator does not need parameters, the class must implement two methods:

- The `__init__()` is passed the original function, and can perform any setup needed.
- The `__call__()` replaces the original function.
  - ▶ In other word, after the function is decorated, calling the function is the same as calling `CLASS.__call__()`.

If the decorator needs parameters:

- The `__init__()` is passed the parameters,
- The `__call__()` is passed the original function, and must return the replacement function.

The following example uses a decorator class to log how many times a function has been called, and keep track of the arguments it is called with.

`deco_debug_class.py`

```

1 #!/usr/bin/env python
2
3
4 class debugger():          # class implementing decorator
5     function_calls = []
6
7     def __init__(self, func):    # original function passed into decorator
8         self._func = func
9
10    def __call__(self, *args, **kwargs): # __call__() is replacement function
11        # add function name and args to saved list
12        self.function_calls.append((self._func.__name__, args, kwargs))
13        result = self._func(*args, **kwargs) # call the original function
14        return result # return result of calling original function
15
16    @classmethod
17    def get_calls(cls): # define method to get saved function call information
18        return cls.function_calls

```

An application that decorates several functions with the previous decorator is shown below.

*deco\_debug\_class\_test.py*

```
1 #!/usr/bin/env python
2 from deco_debug_class import debugger
3
4
5 def main():
6     hello('hello', 'world') # call replacement function
7     print()
8
9     hello('hi', 'Earth')
10    print()
11
12    hello('greetings')
13
14    bark("woof", repeat=3)
15    bark("yip", repeat=4)
16    bark("arf")
17
18    hello('hey', 'girl')
19
20    print('-' * 60)
21
22    # display function call info from class
23    for i, info in enumerate(debugger.get_calls(), 1):
24        print(f"{i:2}. {info[0]:10} {str(info[1]):20} {str(info[2]):20}")
25
26
27 @debugger # apply debugger to function
28 def hello(greeting, whom="world"):
29     print("{} , {} .format(greeting, whom))
30
31
32 @debugger # apply debugger to function
33 def bark(bark_word, *, repeat=2):
34     print("{}! ".format(bark_word) * repeat)
35
36
37 if __name__ == '__main__':
38     main()
```

## Decorator parameters

A decorator can be passed parameters.

- This requires a little extra work.

For decorators implemented as functions, the decorator itself is passed the parameters.

- It contains a nested function that is passed the decorated function (the target), and it returns the replacement function.

For decorators implemented as classes, init is passed the parameters.

- `__call__()` is passed the decorated function (the target)
- `__call__`` returns the replacement function.

There are many combinations of decorators (8 total, to be exact).

- This is because decorators can be implemented as either functions or classes, they may take parameters, or not, and they can decorate either functions or classes.
- An example of all 8 approaches can be seen in the file named `decorama.py`.

A shorter example, that uses one fo the approaches is shown in the next example.

```
1 #!/usr/bin/env python
2 from functools import wraps # Preserves properties of original function
3
4
5 def main():
6     a = spam()
7     b = ham()
8     print(a, b)
9
10
11 def multiply(multiplier):    # actual decorator - receives decorator parameters
12
13     def deco(old_func):        # inner decorator - receives decorated function
14
15         @wraps(old_func)      # retain name, etc. of original function
16         def new_func(*args, **kwargs):    # replacement function for original
17             # call original function and get return value
18             result = old_func(*args, **kwargs)
19
20             # multiple result of original function by multiplier
21             return result * multiplier
22
23     return new_func # new function returned by deco()
24
25
26
27
28 @multiply(4)
29 def spam():
30     return 5
31
32
33 @multiply(10)
34 def ham():
35     return 8
36
37
38 if __name__ == '__main__':
39     main()
```

# Creating classes at runtime

A class can be created programmatically, without the use of the class statement.

- The syntax is as follows:

```
type("name", (base_class, ...), {attributes})
```

- The first argument is the name of the class.
- The second is a tuple of base classes (use `object` if not inheriting from a specific class)
- The third is a dictionary of the class's attributes.

**NOTE**

Instead of `type`, any other *metaclass* can be used.

*creating\_types.py*

```
1 #!/usr/bin/env python
2 def function_1(self): # create method (not inside a class - could be a lambda)
3     print("Hello from f1()")
4
5
6 def function_2(self): # create method (not inside a class - could be a lambda)
7     print("Hello from f2()")
8
9
10 # create class using type()
11 # parameters are class name, base classes, dictionary of attributes
12 NewClass = type("NewClass", (), {
13     'hello1': function_1,
14     'hello2': function_2,
15     'color': 'red',
16     'state': 'Ohio'
17 })
```

The example that follows tests the `type` created above and creates a new subtype from it.

```
1 #!/usr/bin/env python
2 from creating_types import NewClass
3
4
5 def main():
6     n1 = NewClass()    # create instance of new class
7     n1.hello1()        # call instance method
8     n1.hello2()
9     print(n1.color)   # access class data
10    print()
11
12    # create subclass of first class
13    SubClass = type("SubClass", (NewClass,), {'fruit': 'banana'})
14    s1 = SubClass()    # create instance of subclass
15    s1.hello1()        # call method on subclass
16    print(s1.color)   # access class data
17    print(s1.fruit)
18
19
20 if __name__ == '__main__':
21     main()
```

# Monkey Patching

"Monkey patching" refers to the technique of changing the behavior of an object by adding, replacing, or deleting attributes from outside the object's class definition.

It can be used for:

- Replacing methods, attributes, or functions
- Modifying a third-party object for which you do not have access
- Adding behavior to objects in memory

If you are not careful when creating monkey patches, some hard-to-debug problems can arise:

- If the object being patched changes after a software upgrade, the monkey patch can fail in unexpected ways.
- Conflicts may occur if two different modules monkey-patch the same object.
- Users of a monkey-patched object may not realize which behavior is original and which comes from the monkey patch.

Monkey patching defeats object encapsulation, and so should be used sparingly.

```
1 #!/usr/bin/env python
2
3
4 def main():
5
6     s = Spam('Mrs. Higgenbotham') # create instance of class
7     s.eggs() # call method
8
9     def scrambled(self): # define new method outside of class
10        print("Hello, {}. Enjoy your scrambled eggs".format(self._name, ))
11
12    setattr(Spam, "eggs", scrambled) # monkey patch class with the new method
13
14    s.eggs() # call the monkey-patched method from the instance
15
16
17 class Spam(): # create normal class
18
19     def __init__(self, name):
20         self._name = name
21
22     def eggs(self): # add normal method
23         print(f'Good morning, {self._name}. Here are your fried eggs.')
24
25
26 if __name__ == '__main__':
27     main()
```

# Is a Metaclass needed?

Before covering the details of metaclasses, a disclaimer:

- **YAGNI** (You Ain't Gonna Need It) (probably)

When you think you might need a metaclass, consider using inheritance or a class decorator.

However, metaclasses may be a more elegant approach to certain kinds of tasks.

- Such as registering classes when they are defined.

There are two use cases where metaclasses are always an appropriate solution, because they must be done before the class is created:

- Modifying the class name
- Modifying the the list of base classes.

Several popular frameworks use metaclasses, Django in particular.

- In Django they are used for models, forms, form fields, form widgets, and admin media.

Remember that metaclasses can be a more elegant way to accomplish things that can also be done with inheritance, composition, decorators, and other techniques that are less “magic”.

## About metaclasses

Just as a class is used to create an instance, a *metaclass* is used to create a class.

The primary reason for a metaclass is to provide extra functionality at *class creation* time, not *instance creation* time.

Just as a class can share state and actions across many instances, a metaclass can share (or provide) data and state across many *classes*.

The metaclass might modify the list of base classes, or register the class for later retrieval.

The builtin metaclass that Python provides is `type`.

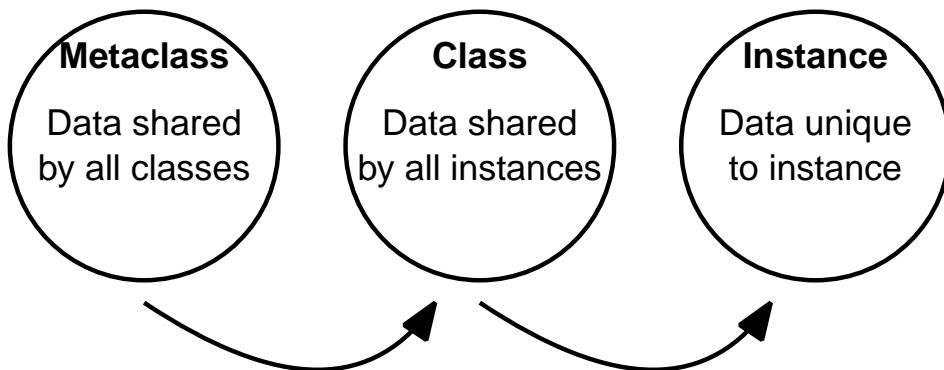
As we saw earlier ,you can create a class from a metaclass by passing in the new class's name, a tuple of base classes (which can be empty), and a dictionary of class attributes (which also can be empty).

```
class Spam(Ham):
    id = 1
```

is exactly equivalent to

```
Spam = type('Spam', (Ham,), {"id": 1})
```

Replacing `type` with the name of any other metaclass works the same.



# Mechanics of a metaclass

To create a metaclass, start by defining a normal class.

- Most metaclasses implement the `__new__` method.
  - ▶ This method is called with the type, name, base classes, and attribute dictionary (if any) of the new class.
  - ▶ It should return a new class, typically using `super().__new__()`, which is very similar to how normal classes create instances.
- This is one place you can modify the class being created.
  - ▶ You can add or change attributes, methods, or properties.

For instance, the Django framework uses metaclasses for Models.

- When you create an instance of a Model, the metaclass code automatically creates methods for the fields in the model.
  - ▶ This is called “declarative programming”, and is also used in SQLAlchemy’s declarative model, in a way pretty similar to Django.

Assuming there is a metaclass named `SomeMeta` that has been defined:

- When you execute the following code:

```
class SomeClass(metaclass=SomeMeta):
    pass
```

- `SomeMeta(name, bases, attrs)` is executed.
  - ▶ Where `SomeMeta` is the metaclass (normally `type()`).
- Then:
  - ▶ The `__prepare__` method of the metaclass is called
  - ▶ The `__new__` method of the metaclass is called
  - ▶ The `__init__` method of the metaclass is called.
- Next, after the following code runs:

```
obj = SomeClass()
```

- `SomeMeta.call()` is called. It returns whatever `SomeMeta.__new__()` returned.

```

1 class Meta(type):
2
3     def __prepare__(class_name, bases):
4         """ "Prepare" the new class. Here you can update the base classes.
5
6         :param name: Name of new class as a string
7         :param bases: Tuple of base classes
8         :return: Initializes the namespace for the new class (must be a dict)
9         """
10        print('=' * 60)
11        print(f"In metaclass (class={class_name}) __prepare__()", 
12              f"with: name={class_name}, bases={bases}, sep=' ==> '")
13        return {'animal': 'wombat', 'id': 100}
14
15    def __new__(metatype, name, bases, attrs):
16        """Create the new class. Called after __prepare___. Note this is
17        only called when classes
18
19        :param metatype: The metaclass itself
20        :param name: The name of the class being created
21        :param bases: bases of class being created (may be empty)
22        :param attrs: Initial attributes of the class being created
23        """
24        print('=' * 60)
25        print(f"In metaclass (class={name}) __new__()", 
26              f"with: type={metatype} name={name} bases={bases} attrs={attrs}",
27              sep=' ==> ')
28        return super().__new__(metatype, name, bases, attrs)
29
30    def __init__(cls, *args):
31        """
32        :param cls: Class being created (compare with 'self' in normal class)
33        :param args: Any arguments to the class
34        """
35        print('=' * 60)
36        print(f"In metaclass (class={cls.__name__}) __init__()")
37        print(f"params: cls={cls}, args={args}", sep=' ==> ')
38        super().__init__(cls)
39
40    def __call__(self, *args, **kwargs):
41        """ Called when the metaclass is called, like NewClass = Meta(...) """
42        print('-' * 60)
43        print("in metaclass (class={self.__name__})__call__()")

```

*metaclass\_use.py*

```
1 #!/usr/bin/env python
2 import metaclass_generic
3
4
5 def main():
6     m1 = A()
7     m2 = B()
8     m3 = A()
9     m4 = B()
10    print("animal: {} id: {}".format(A.animal, B.id))
11
12
13 class MyBase():
14     pass
15
16
17 class A(MyBase, metaclass=metaclass_generic.Meta):
18     id = 5
19
20     def __init__(self):
21         print("In class A __init__()")
22
23
24 class B(MyBase, metaclass=metaclass_generic.Meta):
25     animal = 'wombat'
26
27     def __init__(self):
28         print("In class B __init__()")
29
30
31 if __name__ == '__main__':
32     main()
```

## Singleton with a metaclass

One of the classic use cases for a metaclass in Python is to create a *singleton* class.

- A singleton is a class that only has one actual instance, no matter how many times it is instantiated.
- Singletons are often used for loggers, config data, and database connections, for instance.

To create a singleton, implement a metaclass by defining a class that inherits from `type`.

- The class should have a class-level dictionary to store each class's instance.
- When a new instance of a class is created, check to see if that class already has an instance.
  - ▶ If it does not, call `__call__` to create the new instance, and add the instance to the dictionary.
- In either case, then return the instance where the key is the class object.

*metaclass\_singleton.py*

```

1 #!/usr/bin/env python
2 class Singleton(type): # Define metaclass (must inherit from type)
3     _instances = {} # Define dict to hold list of instances
4
5     # Use __call__(), because __new__() is too early in class creation
6     def __call__(cls, *args, **kwargs):
7         # Check to see if an instance of this class already exists
8         if cls not in cls._instances:
9             # If instance doesn't exist, create instance and add to dict
10            cls._instances[cls] = super().__call__(*args, **kwargs)
11
12        # Always returns the first (and only) instance created.
13        return cls._instances[cls]
14
15
16 class ThingA(metaclass=Singleton): # Create classes using the metaclass
17     pass
18
19
20 class ThingB(metaclass=Singleton): # Create classes using the metaclass
21     pass

```

*create\_singletons.py*

```
1 #!/usr/bin/env python
2 from metaclass_singleton import ThingA, ThingB
3
4
5 def main():
6     # Create multiple instances of each class
7     things = [ThingA(), ThingA(), ThingA(), ThingB(), ThingB(), ThingB()]
8
9     for thing in things:
10         # No matter how many times class is instantiated, it's same instance
11         print(type(thing).__name__, id(thing))
12
13
14 if __name__ == '__main__':
15     main()
```

# Descriptors

Descriptors are the underlying foundation behind a lot of Pythonic magic.

A **descriptor** is an attribute that has custom methods for getting, setting, or deleting itself.

- Note that a `@property` causes such custom behavior on attribute access.
- The return value of `property` is a descriptor.

Any descriptor must implement at least one of the following methods:

- `__get__(self, instance, owner=None)`
- `__set__(self, instance, value)`
- `__delete__(self, instance)`
  - ▶ A descriptor may choose to implement `__set_name__(self, owner, name)`.
  - ▶ This special dunder method is invoked when the descriptor is bound to the attribute `name` in the `owner` class.

The descriptor's methods are invoked when they are attributes on an object.

- This is determined by a transformation done by the `__getattribute__` method.

```
obj.a      # transforms to type(obj).__dict__['a'].__get__(obj, type(obj))
obj.a = v # transforms to type(obj).__dict__['a'].__set__(obj, v)
del obj.a # transforms to type(obj).__dict__['a'].__delete__(obj)
```

Consider a class that tracks inventory.

*inventory\_basic.py*

```
1 class Inventory:
2     def __init__(self, name, count, weight):
3         self.name = name
4         self.count = count
5         self.weight = weight
```

- Two of its attributes, `count` and `weight`, should never be negative.
  - ▶ This could be enforced with getter/setter properties.

*inventory\_tedious.py*

```
1 #!/usr/bin/env python
2
3
4 class Inventory:
5     def __init__(self, name, count, weight):
6         self.name = name
7         self.count = count
8         self.weight = weight
9
10    @property
11    def count(self):
12        return self.__count
13
14    @count.setter
15    def count(self, value):
16        if value < 0:
17            raise ValueError('Cannot be negative')
18        self.__count = value
19
20    @property
21    def weight(self):
22        return self.__weight
23
24    @weight.setter
25    def weight(self, value):
26        if value < 0:
27            raise ValueError('Cannot be negative')
28        self.__weight = value
```

Doing so has greatly increased the amount of code, most of it boilerplate.

Instead, by creating a descriptor, the behavior can be defined in one location, as shown in the next example.

```

1 #!/usr/bin/env python
2
3
4 class NonNegative:
5     def __get__(self, instance, owner):
6         return instance.__dict__[self.name]
7
8     def __set__(self, instance, value):
9         if value < 0:
10             raise ValueError('Cannot be negative')
11         instance.__dict__[self.name] = value
12
13     def __set_name__(self, owner, name):
14         self.name = f'_{name}'
15
16
17 class Inventory:
18     count = NonNegative()
19     weight = NonNegative()
20
21     def __init__(self, name, count, weight):
22         self.name = name
23         self.count = count
24         self.weight = weight

```

This separates out the concern of non-negative values and encapsulates it in a descriptor.

Adding another descriptor to a class occupies only a single line to that class, rather than ten.

When an attribute is accessed, the `__get__` method of the descriptor is invoked, passing in the instance.

When an attribute is updated, the `__set__` method of the descriptor is invoked, passing in the instance and the new value.

If an attribute is deleted from the instance with `del`, the `__delete__` method is invoked, passing in the instance.

# Exercises

## Exercise 1 (pres\_attr.py)

Instantiate the President class.

- Get the first name, last name, and party attributes using getattr().

## Exercise 2 (pres\_monkey.py, pres\_monkey\_amb.py)

Monkey-patch the President class to add a method `full_name` which returns a single string consisting of the first name and the last name, separated by a space.

**TIP** Instead of a method, make `full_name` a property.

## Exercise 3 (sillystring.py)

Without using the `class` keyword, create a class named `SillyString`, which is initialized with any string.

- Include an instance method called `every_other` which returns every other character of the string.

Instantiate your string and print the result of calling the `every_other()` method.

Your test code should look like this:

```
ss = SillyString('this is a test')
print(ss.every_other())
```

It should output

```
ti sats
```

## Exercise 4 (doubledeco.py)

Write a decorator to double the return value of any function.

- If a function returns 5, after decoration it should return 10.
- If it returns "spam", after decoration it should return "spamspam", etc.

## Exercise 5 (word\_actions.py)

Write a decorator, implemented as a class, to register functions that will process a list of words.

- The decorated functions will take one parameter—a string—and return the modified string.

The decorator itself takes two parameters—minimum length and maximum length.

- The class will store the min/max lengths as the key, and the functions as values, as class data.

The class will also provide a method named `process_words`, which will open `../data/words.txt` and read it line by line.

- Each line contains a word.

For every registered function, if the length of the current word is within the min/max lengths, call all the functions whose key is that min/max pair.

In other words, if the registry key is (5, 8), and the value is [func1, func2], when the current word is within range, call `func1(w)` and `func2(w)`, where `w` is the current word.

Example of class usage:

```
word_select = WordSelect() # create callable instance

@word_select(16, 18) # register function for length 16-18, inclusive
def make_upper(s):
    return s.upper()

word_select.process_words() # loop over words, call functions if selected
```

Suggested functions to decorate:

- make the word upper-case
- put stars before or around the word
- reverse the word

Remember all the decorated functions take one argument, which is one of the strings in the word list, and return the modified word.

## Exercise 6 (metacamel.py)

Write a metaclass such that, when specified as the metaclass of a class, it converts the name of the class to UpperCamelCase. Test it with classes whose names are not in standard UpperCamelCase format, such as 'spam\_ham', 'spam\_ham\_eggs', etc.

**TIP**

(Don't read unless you're stuck!) Use `__prepare__()` to initialize the class's dictionary.

## Exercise 7 (nondesaulter.py)

Write a descriptor that defaults to `None`, may be updated to a new value, but cannot be explicitly set to `None` without raising an exception.



# Chapter 10. Parallelism and Concurrency

## Objectives

- Describe the difference between concurrency and parallelism
- Identify when to use multiprocessing
- Identify when to use multithreading
- Understand the implications of the GIL
- Use a worker pool to distribute work
- Use a queue to gather and distribute work

## Concurrency vs. Parallelism

There are many terms when discussing parallelism in programming:

- concurrent, parallel, processes, threads, multiprocessing, and more.
  - ▶ It is important to carefully distinguish these terms from one another.

**Concurrent** programming is a matter of design.

- A concurrent system is one where work or progress can be made on multiple tasks.

**Parallel** programming is a matter of execution.

- A parallel system is one where work or progress happens simultaneously.

A **process** is an execution environment.

- Every process contains at least one thread.
- At the technical level, a process contains the following:
  - ▶ program instructions (text)
  - ▶ static data (data)
  - ▶ dynamic data (heap).

A **thread** is a worker that is carrying out instructions within a process. \* At the technical level, a thread is an execution stack and its registers.

## The Kitchen Metaphor

Consider the metaphor of a process being a kitchen.\* It has tools and stations that can be used to prepare food.\* A thread is a cook in that kitchen.\* Some kitchens can support multiple cooks, but a meal can still be prepared by a single chef.

Concurrent programming corresponds to a well-designed kitchen, where the different workstations are well-separated: a prep station, the stove, etc.

- A chef can work at a station, or be idle at it, waiting for ingredients, resources, or orders.
- A program that was not designed concurrently implies a kitchen with a poor layout, so the stove can't be used while someone is chopping vegetables on top of it.

To strain the metaphor, nothing requires there be just a single kitchen.

- There could be a single room for chopping, a single room for baking, etc.
- Then other mechanisms would be brought into play to ship the ingredients from one room to another.
  - ▶ This corresponds to **multiprocessing**.

Parallel programming means having multiple chefs doing work at once.

- While this sounds like it implies **multithreading**, multiple chefs in the same kitchen, it does not require it.
- Chefs in different kitchens might be preparing different parts of the same meal, coordinated by some other agent.

## Simple fork/exec Model

The most straightforward approach to parallelism is to let the Operating System handle it.

Multiple processes are started, via the `os.fork()` function.

- Almost all operating systems allow for multitasking, and each process is run interleavedly.
- If the processes need to communicate with each other, an OS-level resource (message queue, pipe, shared memory) is used.

The `os.fork()` call creates a copy of the currently-running process.

- The sole difference between the original and forked processes is the return value from that `os.fork()` call.
  - ▶ The newly-created child process receives a 0 value
  - ▶ The parent process receives the process ID (PID) of the newly-created process.

By having different lines of execution predicated on the return value, different processes can tackle different jobs.

*forking.py*

```

1 #!/usr/bin/env python
2 import os
3
4
5 def main():
6     print("Process ID:", os.getpid())
7     print("*" * 30)
8     returned_pid = os.fork()      # 2 processes running after fork() call
9
10    if returned_pid != 0:
11        print_info("parent (original)", returned_pid)
12    else:
13        print_info("child (new one)", returned_pid)
14
15
16 def print_info(msg, returned_pid):
17     fmt = "Inside {} process\n\tIt's pid is:{}\n\tIt received: {} from fork()"
18     print(fmt.format(msg, os.getpid(), returned_pid))
19     print()
20
21
22 if __name__ == '__main__':
23     main()

```

**WARNING**

While this is a simple port of concepts from other languages, it is **not** idiomatic Python. The `multiprocessing` module is a more Pythonic approach.

## Multiprocessing with `multiprocessing`

Python additionally makes available the `multiprocessing` module, which is an easy translation of the `threading` module.

- It allows for the creation of `Process` objects, which will execute a specific function with a given set of arguments.
  - ▶ Processes are started with `.start()` and waited for completion with `.wait()`.
  - ▶ This is the preferred way of working with processes over raw `fork()` calls.

*better\_forking.py*

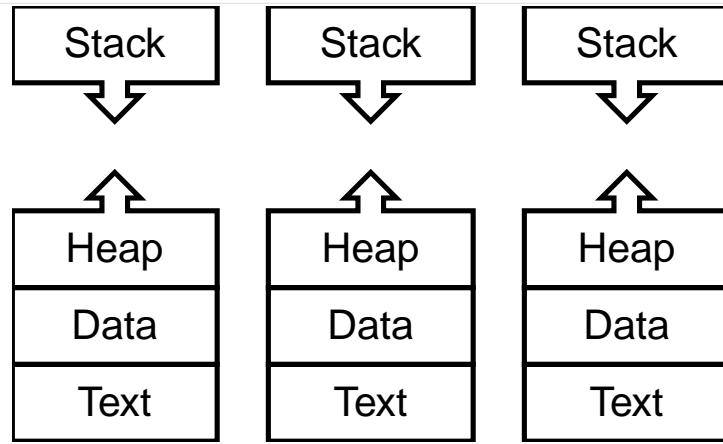
```

1 #!/usr/bin/env python
2 import multiprocessing
3
4
5 def main():
6     p = multiprocessing.Process(target=child_work, args=(1, 2))
7     p.start()
8     print("Inside Parent Process\n")
9     p.join()
10
11
12 def child_work(alpha, bravo):
13     print("Inside Child Process:", alpha + bravo, "\n")
14
15
16 if __name__ == '__main__':
17     main()

```

Multiple processes do not overlap in memory.

- Each process gets its own virtual memory to execute in.
  - ▶ This means that data to be shared between them must use OS-level objects to share data.
  - ▶ These objects (pipes, queues, pools, etc.) are exposed in the `multiprocessing` module.



Note that the implementation of the `Process` interface is quite straightforward:

#### *Theoretical Implementation of Process*

```
class Process:
    def __init__(self, target, args):
        self._target = target
        self._args = args

    def start(self):
        self.child_pid = os.fork()

        if self.child_pid == 0:
            self._target(*self._args)

    def join(self):
        os.waitpid(self.child_pid)
```

But, by providing this straightforward interface to multiprocessing, it becomes much easier to reason about what the target process will do.

#### NOTE

On Windows, processes must be started in the `if __name__ == '__main__'` block, or they will not work.

## Using Pipes

For arbitrary communication between processes, a pipe may be the most flexible choice.

- Like a regular unnamed pipe, the `multiprocessing.Pipe()` function returns two connected objects, and any new process is given one end, while the parent process uses the other end.

*using\_pipes.py*

```

1 #!/usr/bin/env python
2 from multiprocessing import Process, Pipe
3
4
5 def main():
6     parent, child = Pipe()
7     p = Process(target=quack, args=(child,))
8     p.start()
9     print(parent.recv())           # Receiving a picklable object
10    p.join()
11
12
13 def quack(conn):
14     conn.send(['quack', 'quack', 'honk'])  # Sending a picklable object
15     conn.close()
16
17
18 if __name__ == '__main__':
19     main()

```

Any picklable object may be sent over the connection objects.

Pipes are low-level, it is generally preferred to use a higher-level object for communication, such as a queue or a worker pool.

## Using Pools

Many concurrent tasks may process a collection of data, one item at a time.

- This is easily accomplished with the `multiprocessing.Pool` object.
  - ▶ A set number of workers are created in a context manager, and then data may be passed to them en masse.

The pool is used by calling the `.imap()` method with a function that will do the work, and an iterable of data.

- `.imap()` will return a generator the same length as the iterable that was passed in,
  - ▶ The returned generator contains the results returned by the function for each item in the original iterable.
- `.imap()` also returns the results in the same order as the initial iterable, which may incur some overhead.
  - ▶ By using the `.imap_unordered()` method, the results are returned as fast as possible, but may be out of order compared to the initial iterable.

```
1 #!/usr/bin/env python
2 import random
3 from multiprocessing import Pool
4
5
6 def main():
7     POOL_SIZE = 30 # Number of processes
8
9     # Read a line at a time from file into a list (removing new lines)
10    with open('../data/words.txt') as words_in:
11        word_list = [w.strip() for w in words_in]
12
13    random.shuffle(word_list) # Randomize word list
14
15    pool = Pool(POOL_SIZE) # Create pool with POOL_SIZE processes
16
17    # Pass word_list to pool and get results
18    # imap_unordered assigns values from list to process as needed
19    upper_words = pool imap_unordered(the_task, word_list)
20
21    for word, _ in zip(upper_words, range(20)): # Print just 20 the words
22        print(word)
23
24
25 def the_task(word): # Actual task to be completed
26     return word.upper()
27
28
29 if __name__ == '__main__':
30     main()
```

## Using Queues

A pool works well when there is one source of work.

If multiple agents may add work to be done, a `multiprocessing.Queue` object may be the best choice.

- This queue object is thread- and process-safe, so work items may be added and removed from it by multiple workers.

`proc_queue.py`

```

1 #!/usr/bin/env python
2 from multiprocessing import Process, Queue
3 from datetime import datetime
4
5
6 def main():
7     line = Queue()
8     procs = [Process(target=append, args=(line,)),
9              Process(target=append, args=(line,)),
10             Process(target=remove, args=(line,)),
11             Process(target=remove, args=(line,))]
12
13     for p in procs:
14         p.start()
15
16     for p in procs:
17         p.join()
18     print("All child processes have completed their work")
19
20
21 def append(q):
22     date = datetime.now()
23     print('Sending', date)
24     q.put(date)
25
26
27 def remove(q):
28     item = q.get()
29     print('Got', item)
30
31
32 if __name__ == '__main__':
33     main()
```

Queues are very flexible, allowing for any number of producers and consumers.

---

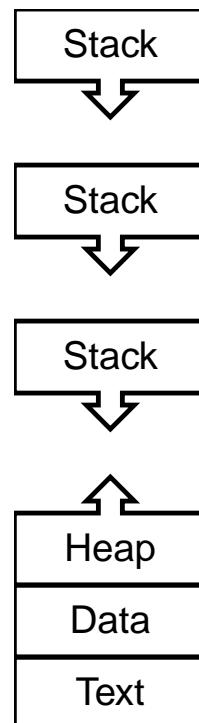
Closing a queue requires a certain level of coordination between workers.

- It is traditional to insert a signal value (such as **None**) to let consumers know that they should shut down.
  - ▶ Note that there would need to be as many such signals inserted as there are consumers.

## Multithreading with `threading`

Multithreading, where multiple threads of execution run in a process, can be more difficult to manage and think about than multiprocessing.

- Threads in the same process share parts of their memory, notably the heap, where Python objects are stored.
  - ▶ Multiple threads reading or writing the same object can result in difficult, hard-to-find bugs.



Static or global variables may be touched by many threads in unintuitive orders.

- While this sounds ominous, the interface to using threads in Python is through the `threading` module, whose interface looks nearly identical to the `multiprocessing` module.

```
1 #!/usr/bin/env python
2 import threading
3
4
5 def main():
6     t = threading.Thread(target=childs_work, args=(1, 2))
7     t.start()
8     print("Inside Parent Thread\n")
9     t.join()
10
11
12 def childs_work(alpha, bravo):
13     print("Inside Child Thread:", alpha + bravo, "\n")
14
15
16 if __name__ == '__main__':
17     main()
```

A thread has its own stack and set of registers.

- It shares the other parts of a process (code, data, resources, file descriptor tables, etc.) with other threads in the same process.
  - ▶ This means that coordination between threads is very important.

Any piece of data that is modified by more than one thread needs the ability to ensure that such operations are **atomic**

- An atomic operation is one that will not be interrupted by other threads of execution, leaving the value in an invalid or indeterminate state.
  - ▶ This can also apply to data modified by only one thread, but read by many. <<<

*failed\_threads.py*

```

1 #!/usr/bin/env python3
2 from threading import Thread
3
4 target = 0
5
6
7 def unit():
8     global target
9     for _ in range(100000):
10         target += 1
11         target -= 1
12
13
14 def main():
15     thread_count = 100
16     threads = set()
17     for _ in range(thread_count):
18         threads.add(Thread(target=unit))
19     for t in threads:
20         t.start()
21     for t in threads:
22         t.join()
23     print('got {}, should be 0'.format(target))
24
25
26 if __name__ == '__main__':
27     main()

```

In the above code, a number of threads are started that all modify a global variable.

- The addition/subtraction operations are **not** atomic, which cause some jitter on the actual value of the global.
  - ▶ The correct execution would be to identify sections of code that work with global variables, and gate those sections with locks or semaphores.

```
1 #!/usr/bin/env python3
2 import threading
3
4 target = 0
5 lock = threading.Lock()
6
7
8 def unit():
9     for _ in range(10000):
10         count_up()
11         count_down()
12
13
14 def count_up():
15     global target
16     with lock:
17         target += 1
18
19
20 def count_down():
21     global target
22     with lock:
23         target -= 1
24
25
26 def main():
27     thread_count = 100
28     threads = set()
29     for _ in range(thread_count):
30         threads.add(threading.Thread(target=unit))
31     for t in threads:
32         t.start()
33     for t in threads:
34         t.join()
35     print('got {}, should be {}'.format(target))
36
37
38 if __name__ == '__main__':
39     main()
```

A **lock** in Python can be created as a `threading.Lock` object.

- The object can be used as a context manager to ensure that only one thread is executing that block of code at any time.

- If a second thread tries to obtain the context manager, it blocks at that point until the current lock exits.

## Creating a thread class

A thread class is a class that starts a thread, and performs some task.

- Such a class can be repeatedly instantiated, with different parameters, and then started as needed.

The class can be as elaborate as the business logic requires.

- There are only two rules:
  - ▶ The class must call the base class's `__init__()`.
  - ▶ It must implement a `run()` method.
- Other than that, the `run()` method can do pretty much anything it wants to.

The preferred way to invoke the base class `__init__()` is to use `super()`.

The `run()` method is invoked when you call the `start()` method on the thread object.

The `start()` method does not take any parameters, and thus `run()` has no parameters as well.

Any per-thread arguments can be passed into the constructor when the thread object is created.

*extending\_thread.py*

```
1 #!/usr/bin/env python
2 from threading import Thread
3 import random
4 import time
5
6
7 def main():
8     for i in range(10):
9         t = SimpleThread(i) # Create the thread
10        t.start()          # Start the thread
11
12    print("Done.")
13
14
15 class SimpleThread(Thread):
16     def __init__(self, num):
17         super().__init__() # Required call to parent class constructor
18         self._threadnum = num
19
20     def run(self): # This is where the work of the thread is done
21         time.sleep(random.randint(1, 3))
22         print(f"Hello from thread {self._threadnum}")
23
24
25 if __name__ == '__main__':
26     main()
```

## The GIL

There is one major drawback in Python when it comes to multithreading: Python doesn't execute threads in parallel.

- Python is only single-threaded.
- This is by design, using the **Global Interpreter Lock**, or GIL.

Any mutable object is at-risk for a **race condition**, where two threads of execution could modify the same object at the same time, causing one of those changes not to take effect.

- This problem is normally solved in other languages with semaphores or mutexes.
  - ▶ This approach still requires manual programmer intervention and is easy to write incorrectly.

In other platforms, an individual data structure or object may be locked, preventing access to it until a given thread of execution is done modifying it.

- Python takes this idea to the extreme by locking **everything** in the interpreter, automatically, for a given thread of execution.
- It does this to prevent thread-unsafe access of Python's builtin types.

This lock makes it easier to port thread-unsafe libraries to Python, by simulating a single-threaded environment.

- Unfortunately, this is still not guaranteed to result in correct parallel code!

*failed\_threads.py*

```

1 #!/usr/bin/env python3
2 from threading import Thread
3
4 target = 0
5
6
7 def unit():
8     global target
9     for _ in range(100000):
10         target += 1
11         target -= 1
12
13
14 def main():
15     thread_count = 100
16     threads = set()
17     for _ in range(thread_count):
18         threads.add(Thread(target=unit))
19     for t in threads:
20         t.start()
21     for t in threads:
22         t.join()
23     print('got {}, should be 0'.format(target))
24
25
26 if __name__ == '__main__':
27     main()

```

As already shown, this code will produce non-zero results for `target`, as the addition and subtraction operations are not atomic.

Certain operations in Python are atomic; generally each Python bytecode is an atomic operation.

Certain functions are also atomic, e.g. `list.sort`, but this is more an accident of implementation than intentional behavior.

- It is best to write correct code first, and not attempt to optimize away unnecessary synchronization.

In Python 3, a thread may run for up to 5ms before it gives up ownership of the GIL to allow another thread to run.

- The GIL is also relinquished when a thread is waiting for I/O.
- This “I/O Waiting” can be depended upon, and is elevated in visibility when using `asyncio`.

The overhead in locking and unlocking the GIL means that CPU-intensive tasks in Python have lower throughput when parallelized via threads.

- Since only one thread runs at a time, performance will never be better than single-threaded.

*thread\_slowdown.py*

```

1 #!/usr/bin/env python
2 from time import time
3 from threading import Thread
4
5
6 def main():
7     threads, TOTAL = 2, 100000000
8
9     t = [Thread(target=count, args=(TOTAL//threads,)) for _ in range(threads)]
10
11    start = time()
12    for item in t:
13        item.start()
14    for item in t:
15        item.join()
16    end = time()
17
18    print('{} threads: {}s'.format(threads, end - start))
19
20
21 def count(n):
22     while n > 1:
23         n -= 1
24
25
26 if __name__ == '__main__':
27     main()

```

The GIL, at this point, is a fundamental design aspect of the standard Python implementation, and will not be going away.

- This means that all threading in Python is generally done when waiting on I/O, rather than for splitting up a CPU-intensive task.
- In IronPython and Jython, the GIL does not exist.

## Thread Pools

Even though the GIL will limit thread execution to one per process, Python does expose an API for working with thread pools.

- It is an identical interface to the `multiprocessing.Pool` object.

The threading interface is the `multiprocessing.dummy.Pool` object.

- Unfortunately, it is limited by the GIL to just a single active thread.

*thr\_pool.py*

```

1 #!/usr/bin/env python
2 import random
3 from multiprocessing.dummy import Pool
4
5
6 def main():
7     POOL_SIZE = 30 # Number of processes
8
9     # Read a line at a time from file into a list (removing new lines)
10    with open('../data/words.txt') as words_in:
11        word_list = [w.strip() for w in words_in]
12
13    random.shuffle(word_list) # Randomize word list
14
15    pool = Pool(POOL_SIZE) # Create pool with POOL_SIZE processes
16
17    # Pass word_list to pool and get results
18    # imap_unordered assigns values from list to process as needed
19    upper_words = pool imap_unordered(the_task, word_list)
20
21    for word, _ in zip(upper_words, range(20)): # Print just 20 the words
22        print(word)
23
24
25 def the_task(word): # Actual task to be completed
26     return word.upper()
27
28
29 if __name__ == '__main__':
30     main()
```

## Thread Queues

Multiprocessing requires a special kind of queue that can send and receive data across a process boundary.

A multithreaded queue can be much more lightweight, only needing locking operations to be safely used across multiple threads.

The base `queue` module's classes will work with multithreading.

- The queues it provides are thread-safe.
- It provides four different kinds of queues:
  - ▶ `Queue`:: A normal FIFO queue
  - ▶ `LifoQueue`:: A LIFO queue, or stack
  - ▶ `PriorityQueue`:: A queue that returns items in priority order, lowest to highest
  - ▶ `SimpleQueue`:: A lightweight queue that is missing some functionality

```
1 #!/usr/bin/env python
2 import random
3 import threading
4 import queue
5 import time
6
7
8 class LockList(list): # Extend list to be lockable (not required for append)
9     lock = threading.Lock()
10
11    def __enter__(self):
12        self.lock.acquire()
13        return self
14
15    def __exit__(self, exc, type, traceback):
16        self.lock.release()
17
18
19 class RandomWord(): # Define callable class to generate words
20     def __init__(self):
21         with open('../data/words.txt') as WORDS:
22             self._words = [word.strip() for word in WORDS]
23
24     def __call__(self):
25         return random.choice(self._words)
26
27
28 class Worker(threading.Thread): # Define worker thread
29     def __init__(self, name, q, results): # Initialize the thread
30         super().__init__()
31         self.name = name
32         self.q = q
33         self.results = results
34
35     def run(self): # Invoked when it is time to run thread
36         while True:
37             try:
38                 word = self.q.get(block=False) # Get next item from queue
39                 with self.results as out: # Acquire lock and release when done
40                     out.append(f'{self.name}-{word.upper()}')
41
42             except queue.Empty: # Raised when queue becomes empty
43                 break
```

*thr\_queue\_test.py*

```
1 #!/usr/bin/env python
2 import random
3 import queue
4 import time
5 from thr_queue import RandomWord, Worker, LockList
6
7
8 def main():
9     NUM_ITEMS = 25000
10    POOL_SIZE = 25
11
12    q = queue.Queue(0) # Initialize empty queue
13
14    words = RandomWord()
15    for i in range(NUM_ITEMS):
16        w = words()
17        q.put(w) # Fill the Queue
18
19    start_time = time.ctime()
20    output = LockList()
21
22    pool = []
23    for i in range(POOL_SIZE):
24        w = Worker(f'Worker {chr(i+65)}', q, output) # Add Thread to Pool
25        w.start() # Start the thread
26        pool.append(w)
27
28    for t in pool:
29        t.join() # Wait for thread to finish
30
31    end_time = time.ctime()
32
33    for _ in range(20):
34        print(random.choice(output))
35
36    print(start_time)
37    print(end_time)
38
39
40 if __name__ == '__main__':
41     main()
```

## Debugging threaded Programs

Debugging is always tough with parallel programs, including threaded programs.

- It's especially difficult with pre-emptive threads.
  - ▶ those accustomed to debugging non-threaded programs find it rather jarring to see sudden changes of context while single-stepping through code.

Tracking down the cause of deadlocks can be very hard.

- Often just getting a threaded program to end properly is a challenge.

Another problem which sometimes occurs is that issuing a `next` command may end up inside the internal thread's code.

- In such cases, a `continue` command can be used to resume userland debugging.

Unfortunately, threaded debugging is even more difficult in Python, at least with the basic PDB debugger.

- Invoking the debugger directly will fail to work correctly.
  - ▶ This is because the child threads will not inherit the PDB process from the main thread.

```
> # FAILS
> python3 -m pdb debug_threading.py
```

The correct way to invoke PDB is to manually call `pdb.set_trace()` inside the thread's target function.

- In essence, those points become breakpoints.

*debug\_threading.py*

```
1 #!/usr/bin/env python
2 import threading
3 import pdb
4
5
6 def main():
7     t = threading.Thread(target=childs_work, args=(1, 2))
8     t.start()
9     print("Inside Parent Thread\n")
10    t.join()
11
12
13 def childs_work(alpha, bravo):
14     pdb.set_trace()
15     print("Inside Child Thread:", alpha + bravo, "\n")
16
17
18 if __name__ == '__main__':
19     main()
```

The program should be run normally (**not** through PDB), and then it will automatically enter PDB on its own.

- Step-by-step debugging control then works normally (**n**, **s**, **c**, etc.).

## Exercises

For each exercise, ask the questions:

- Should this be multithreaded or multiprocessed?
- Distributed or local?

### Exercise 1 (pres\_thread.py)

Using a thread pool (`multiprocessing.dummy`), calculate the age at inauguration of the presidents.

- To do this, read the `presidents.txt` file into an array of tuples, and then pass that array to the mapping function of the thread pool.
- The result of the map function will be the array of ages.
- You will need to convert the date fields into actual dates, and then subtract them.

### Exercise 2 (wc.py)

Write a program that takes in a directory name on the command line, then traverses all the files in that directory tree and prints out a count of:

- how many files,
- how many lines,
- how many words, and
- how many characters.

### Exercise 3

Write a function that will take in two large arrays of integers and a target.

- It should return an array of tuple pairs
  - ▶ Each pair being one number from each input array, that sum to the target value.

# Chapter 11. Coroutines and asyncio

## Objectives

- Understand Asynchronous Programming
- Learn when to use `async` and `await`
- Easily parallelize I/O with coroutines

## Asynchronous programming with `asyncio`

The `asyncio` module adds easier concurrent programming to Python.

- Like threads, asynchronous code can take advantage of the time while I/O processes are blocking.
  - ▶ While the GIL does this transparently in multithreaded programs, using `asyncio` primitives allows the programmer a better understanding and control of program flow.

In synchronous (normal) programming, an I/O call (reading from a file, a web site, printing, etc.) will wait until the call returns, known as **blocking**.

In asynchronous programming, when the code is waiting for I/O or some other long-running task, it will manually return control to an event loop so other code can run.

Why not just use threads? \* Asynchronous programming, while complex to understand, can greatly simplify concurrent code.

# Key asynchronous vocabulary

An **event loop** is what controls the asynchronous functions.

- When a function needs something, it makes an I/O call.
- If the data are not ready, it returns control to the loop so that some other function can run.
  - ▶ The event loop is like a miniature multithreaded pool.

**Coroutine** is short for "cooperative routine".

- Coroutines are functions that can cooperate by sharing the event loop.
- A coroutine is a more general form of a function.
  - ▶ A normal function "blocks", in that it starts at the beginning, and runs until it returns.
  - ▶ A coroutine can "pause" its execution at various points and yield control to some **other** piece of code, rather than running to completion.
  - ▶ Generators in Python are a stepping stone between a basic function and a full-blown coroutine.

A **Future** is an object that will eventually (in the future) have a result.

- Futures are generally not explicit in application code; they are for lower-level library code.

A **Task** is a Future wrapping a coroutine, specifically.

- The event loop does not run coroutines directly—it runs tasks.

## Defining coroutines

A *coroutine* is a function defined with the `async` modifier keyword.

- Coroutines used to be defined as any function containing `yield from`.
  - ▶ That has changed in 3.7 with the addition of the `async` and `await` keywords that indicate coroutine-ness.

*basic\_coroutine.py*

```

1 #!/usr/bin/env python
2 import asyncio
3 import time
4
5 def main():
6     asyncio.run(lazy_print('Hello World'))
7
8
9 async def lazy_print(msg):
10    time.sleep(3)
11    print(msg)
12
13
14 if __name__ == '__main__':
15     main()

```

Any coroutine cannot be run by calling it directly.

- Instead, it must be run through the `asyncio` framework.
  - ▶ The `run()` function will handle running the coroutine, it fires up the event loop and places the code as a `Task` inside the event loop.

When a coroutine is called, it doesn't call the function, but creates and returns a *coroutine object*.

Compare this with creating a generator function.

- When a function contains `yield`, calling the function creates and returns a generator object; it does not execute the function.
  - ▶ Calling `next()` on the object is what actually runs code.

Similarly, using a coroutine object in an `await` expression actually runs the code.

The coroutine must be invoked by one of the event loop's methods.

- TIP: Use `asyncio.iscoroutine()` to check whether an object is a coroutine.
- TIP: Use `asyncio.iscoroutinefunction()` to check whether a function is a coroutine.

# The Event Loop

All coroutines must be executed within an *event loop*.

- By default, such an event loop is exposed in the `asyncio` module via `.get_event_loop()`.
  - ▶ While it is possible to create custom loop behavior or driver code, it is generally easiest to use the builtin event loop.
- Only one event loop can be running in a given thread.

*the\_event\_loop.py*

```

1 #!/usr/bin/env python
2 import asyncio
3 from basic_coroutine import lazy_print
4
5
6 def main():
7     loop = asyncio.get_event_loop()
8
9     loop.run_until_complete(lazy_print('Hello World'))
10
11
12 if __name__ == '__main__':
13     main()

```

Once the loop is created, coroutines may be added to it to run at some later time.

There are a number of such methods to facilitate this.

- `.create_task(coro)`
  - ▶ Add the coroutine to the loop, to be run later.
- `.run_forever()`
  - ▶ Run the event loop.
- `.run_until_complete(coro)`
  - ▶ Run the event loop immediately, suspending when the coroutine is complete.
- `.wait(coros)`
  - ▶ Create one coroutine that iterates through the argument list of coroutines, effectively combining them into a single coroutine object.

In addition to `loop.create_task()`, there is a module-level `asyncio.create_task()` function that can be used to create and schedule tasks on the current thread's event loop.

## Effective Coroutines

Truly effective coroutines are able to "give up" their ownership of a currently running thread and allow some other portion of work to progress.

- They do this by prefixing a function call with `await`.
  - ▶ While that `awaited` function executes in the background, a different coroutine in the event loop may make progress.
  - ▶ When the `awaited` function has finished, the coroutine will start again.

As such, this waiting is only effective when the function invoked is not CPU-bound.

- If the call is a `sleep`, or reading from a network socket, or publishing an event to an event queue, that call will be I/O-bound, no matter how fast the CPU is.
  - ▶ These are prime targets for the `await` invocation.
  - ▶ Only a coroutine declared with `async` may use `await` to control program flow.

`async_await.py`

```

1 #!/usr/bin/env python
2 import asyncio
3 from basic_coroutine import lazy_print
4
5 def main():
6     asyncio.run(serial_sleeper())
7
8
9 async def serial_sleeper():
10    print(1)
11    await asyncio.sleep(2)
12    await lazy_print('In Sleeper')
13    await asyncio.sleep(1)
14    print(2)
15
16
17 if __name__ == '__main__':
18     main()

```

Note that using `await` does not cause the current thread to execute out-of-order, it merely *allows* other asynchronous coroutines to start executing.

The previous code will print 1, sleep for two seconds, print '`In Sleeper`', sleep for one second, then print 2.

- It executes these in that order because no other coroutines are scheduled to run in the event loop.

A function declared as `async` **must** be `awaited`.

- Merely calling the function returns a special object, similar to how a generator call returns the generator object.
- That special object, a **future**, must be `awaited`.
  - ▶ Failing to do so results in a warning, letting the developer know that there is a programming error.

Scheduling multiple coroutines demonstrates much more clearly how asynchronous methods are scheduled by the event loop.

`async_tasks.py`

```

1 #!/usr/bin/env python
2 import asyncio
3 from random import randrange
4
5
6 def main():
7     loop = asyncio.get_event_loop()
8     loop.create_task(sleeper(1))
9     loop.create_task(sleeper(2))
10    loop.run_forever()
11
12
13 async def sleeper(n):
14     while True:
15         timeout = randrange(10)
16         print(f'[{n}] Sleeping for {timeout}s')
17         await asyncio.sleep(timeout)
18         print(f'[{n}] Done sleeping for {timeout}s')
19
20
21 if __name__ == '__main__':
22     main()

```

**Task**s are added to the loop to be run, and the loop switches between the tasks when possible.

While not being very interesting when it comes to mere `sleep` calls, the I/O multiplexing possibilities are limitless.

## Awaitable objects

An object is *awaitable* if it can be used with the `await` keyword; that is, if it can be called asynchronously.

- This generally means a coroutine, `Future`, or `Task`.

There are a number of interfaces that work with awaitables, the specific instance is generally not relevant.

- Due to duck typing, the specific instance should not be relevant in the majority of cases.

The vast majority of awaitables will be either coroutines or event loop-level `Task` objects.

## Futures

A `Future` is an object that will eventually hold the result of some code.

- That result may be an exception thrown by said code.

It is possible to query the `Future` object to discover its current status, the result, or the exception.

- `done()`
  - ▶ Indicates whether the `Future` has finished
- `result()`
  - ▶ Returns the result or raises the exception from the code
- `exception()`
  - ▶ Returns the exception from the code, or `None`
- `add_done_callback()`
  - ▶ Add a callback to be executed when the `Future` is done
- `cancel()`
  - ▶ Cancels the `Future`'s execution; starts callbacks
- `cancelled()`
  - ▶ Indicates whether the `Future` was cancelled

Unfortunately, there are two incompatible types of `Future` objects in Python:

- `concurrent.futures.Future` (thread-safe)
- `asyncio.futures.Future` (not thread-safe).

In general, these objects should not be created directly, but by various helper functions.

A `Future` is an awaitable object, so other coroutines and asynchronous code can `await` them for a result, exception, or cancellation.

- Futures are generally managed and manipulated by lower-level library code, rather than application code.

`async_future.py`

```

1 #!/usr/bin/env python
2 import asyncio
3 from random import randrange
4
5
6 def main():
7     loop = asyncio.get_event_loop()
8
9     # Not preferred, see next section
10    m1 = monitor(asyncio.ensure_future(long_process(1)))
11    m2 = monitor(asyncio.ensure_future(long_process(2)))
12
13    # Not preferred, see next section
14    asyncio.ensure_future(m1)
15    asyncio.ensure_future(m2)
16
17    loop.run_forever()
18
19
20 async def monitor(fut):
21     while not fut.done():
22         print(f'Waiting for {fut}\n')
23         await asyncio.sleep(1)
24         print(f'{fut} complete\n')
25
26
27 async def long_process(name):
28     print(f'Beginning {name}\n')
29     timeout = randrange(2, 10)
30
31     # Simulate a long asynchronous work item
32     await asyncio.sleep(timeout)
33     print(f'Ending {name}\n')
34
35
36 if __name__ == '__main__':
37     main()

```

The `ensure_future()` function will do two things:

- Make sure that the object passed to it conforms to the `Future` interface.
- Automatically add it to an event loop.
  - ▶ Once again, this is using the low-level `Future` object.

- 
- ▶ Most code interaction, if any, should be done with `Task` objects.

## Tasks

A **Task** is a subclass of **Future** that specifically wraps a coroutine.

Because a **Task** wraps a coroutine, it has additional methods to extract information about its execution that a **Future** object does not have:

- **get\_stack**
  - ▶ Returns a list of stack frames for the current **Task**
- **print\_stack**
  - ▶ Prints stack frames for the current **Task**

A **Future** object should never be created by end-user code, only **Task** objects.

- The **create\_task** method of the event loop can be used to wrap a coroutine in a task and add it to the loop.

```

1 #!/usr/bin/env python
2 import asyncio
3 from random import randrange
4
5
6 def main():
7     loop = asyncio.get_event_loop()
8
9     m1 = monitor(loop.create_task(long_process(1)))
10    m2 = monitor(loop.create_task(long_process(2)))
11    loop.create_task(m1)
12    loop.create_task(m2)
13
14    loop.run_forever()
15
16
17 async def monitor(fut):
18     while not fut.done():
19         print(f'Waiting for {fut}\n')
20         await asyncio.sleep(1)
21     print(f'{fut} complete\n')
22
23
24 async def long_process(name):
25     print(f'Beginning {name}\n')
26     timeout = randrange(2, 10)
27
28     # Simulate a long asynchronous work item
29     await asyncio.sleep(timeout)
30     print(f'Ending {name}\n')
31
32
33 if __name__ == '__main__':
34     main()

```

Without the loop having its own signal handler, it is not possible for normal signal handlers to affect the event loop.

Each task managed by the event loop is captured in a `Task` object, which can be queried by the `all_tasks()` function.

When it comes to managing awaitable objects, it can be useful to wait for a group of them to complete.

- It is possible to group multiple asynchronous pieces of work using the `gather()` function in `asyncio`.

- ▶ This function creates a new awaitable object that simply `awaits` all the passed awaitables concurrently.

```
1 #!/usr/bin/env python
2 import asyncio
3 from random import randrange
4
5
6 def main():
7     asyncio.run(roll_up(10))
8
9
10 async def long_process(name):
11     print(f'Beginning {name}')
12     timeout = randrange(2, 10)
13
14     # Simulate a long asynchronous work item
15     await asyncio.sleep(timeout)
16     print(f'Ending {name}')
17     return timeout * 1000
18
19
20 async def roll_up(times):
21     jobs = [long_process(n) for n in range(times)]
22     results = await asyncio.gather(*jobs)
23     print(f'Slept a total of {sum(results)}ms across {times} tasks')
24
25
26 if __name__ == '__main__':
27     main()
```

A `Task` object can be cancelled, and its results inspected.

- Doing so raises a `CancelledError` exception, which can be suppressed during normal cleanup.
- The `gather()` function also takes an optional keyword parameter `return_exceptions`, which will return `Exception` objects instead of raising them, which can be useful when dealing with multiple tasks.

Similar to `gather` is the `wait` function.

- The difference is that the `asyncio.wait()` function is capable of returning early, using its `return_when` keyword.

*async\_waiting.py*

```

1 #!/usr/bin/env python
2 import asyncio
3 from random import randrange
4
5
6 def main():
7     asyncio.run(upto_first_exception(10))
8
9
10 async def maybe_throw(n):
11     throws = randrange(10) < 1
12     timeout = randrange(2, 10)
13
14     print(f'{n} will throw in {timeout}s: {throws}')
15     await asyncio.sleep(timeout)
16
17     if throws:
18         raise Exception('Throwing!')
19     else:
20         return True
21
22
23 async def upto_first_exception(times):
24     jobs = [maybe_throw(n) for n in range(times)]
25
26     result = await asyncio.wait(jobs, return_when=asyncio.FIRST_EXCEPTION)
27     done, pending = result
28
29     print(f'Ran {len(done)} jobs, leaving {len(pending)}')
30
31
32 if __name__ == '__main__':
33     main()

```

Part of its utility is the ability to capture exceptions of awaitables as results rather than allowing the exception to continue bubbling up.

## Exceptions

Dealing with exceptions in asynchronous code is unintuitive at first.

- An unhandled exception can easily lead to a deadlock, yet the exception may be raised far from the code when it is scheduled.
- Further, the `KeyboardInterrupt` exception can produce very ugly tracebacks.

One difficult portion is making sure that all buffers are flushed, cleaned, and logged in an asynchronous application.

- Fortunately, the event loop itself has its own signal handlers.
- A specific signal can be used to control a graceful shutdown of the loop and therefore the application.

*async\_exceptions.py*

```
1 #!/usr/bin/env python
2 import asyncio
3 import signal
4 from random import randrange
5 from async_tasks import sleeper
6
7
8 def main():
9     loop = asyncio.get_event_loop()
10
11    loop.add_signal_handler(signal.SIGINT, asyncio.create_task,
12                           shutdown(loop))
13
14    loop.create_task(sleeper(1))
15    loop.create_task(sleeper(2))
16
17    loop.run_forever()
18
19
20 async def shutdown(loop):
21     print("Beginning graceful shutdown")
22     tasks = [t for t in asyncio.all_tasks(loop)
23              if t is not asyncio.current_task(loop)]
24
25     for task in tasks:
26         task.cancel()
27     # Allowing exceptions to bubble will cause the program to
28     # hang forever, as nothing handles the exceptions nor stops
29     # the loop
30     await asyncio.gather(*tasks, return_exceptions=True)
31     loop.stop()
32
33
34 if __name__ == '__main__':
35     main()
```

## Callbacks

It is possible to schedule functions to run at specific times, using the callback capability of the `asyncio` module. Note that callback functions are *not* coroutines! They execute synchronously!

`async_callbacks.py`

```
1 #!/usr/bin/env python
2 import asyncio
3
4
5 def main():
6     loop = asyncio.get_event_loop()
7
8     loop.call_later(6, print, 0, 'BOOM')
9     loop.call_later(5, print, 1)
10    loop.call_later(4, print, 2)
11    loop.call_later(3, print, 3)
12    loop.call_soon(print, 'Starting countdown!')
13
14    loop.run_forever()
15
16
17 if __name__ == '__main__':
18     main()
```

A callback may be invoked after a specific delay with `call_later` or as soon as possible with `call_soon`, or even at a specific time with `call_at`.

- The function in question will be invoked with the specified parameters.

Generally, global callbacks can and should be avoided in favor of awaitables.

Individual `Future` objects might have a few callbacks associated with them.

## How to run coroutines

There are multiple ways to run coroutines; generally involving methods from the `asyncio` module.

The simplest way is to use the `create_task()` method of the event loop to create one or more tasks.

- Then pass a task list to `asyncio.wait()` to schedule them.
- Finally, call `loop.run_until_complete()` with the object returned by `asyncio.wait()`.

`async_simple.py`

```
1 #!/usr/bin/env python
2 import asyncio
3
4
5 def main():
6     loop = asyncio.get_event_loop()                      # Get the event loop
7     loop.run_until_complete(say('hello, async world', 1)) # Run Coroutine
8     loop.close()                                         # Close the loop
9
10
11 async def say(what, when):    # Create coroutine with async
12     await asyncio.sleep(when)   # Simulate actual work with sleep
13     print(what)               # Business logic goes here
14
15
16 if __name__ == '__main__':
17     main()
```

---

# Exercises

## Exercise 1

Using callbacks, print a countdown from 10 to '**Blast off!**'

## Exercise 2

Using `asyncio` and `aiofiles`, write a program that takes in a directory name on the command line, then traverses all the files in that directory tree and prints out a count of:

1. how many files,
2. how many lines,
3. how many words, and
4. how many bytes.

## Exercise 3

Using `asyncio` and the `aiohttp` module, write a website-spider.

- Given a domain name, it should crawl the page at that domain, and any other URLs from that page with the same domain name.
- Limit the number of parallel requests to the webserver to no more than 4.

HINT: See `async_lcbo.py` for some examples of how to fetch JSON from a website.

# Chapter 12. Distributed Computing

## Objectives

- Understand what distributed computing is
- Learn terms used in parallel and distributed computing
- Run simple jobs with SCOOP
- Coordinate complicated jobs using Message Passing in `mpi4py`

## Parallel and distributed computing

**Parallel computing** is the concept of using more than one processor (or core) to solve a programming problem.

- This lets code take advantage of "down time" due to a process waiting for an I/O event.
- However, a program can only run as fast as the processor on the machine.

**Distributed computing** is the concept of using more than one *computer* to solve a programming problem.

- This allows code to be spread out over multiple machines, so that a task can be executed much faster than it could on a single machine.

TIP

A good source for distributed Python computing is "Distributed Computing with Python", from Packt Publishing

## Asynchronous programming

Asynchronous programming in Python can be achieved using threads, processes, or coroutines (and the `asyncio` subsystem). Another name for `async` programming is *event-driven*.

- The preferred approach nowadays is to use built-in `coroutines` (`async` and `await`) and the `asyncio` module.
  - ▶ See the chapter on coroutines and `asyncio` programming for full details.

## Multiprogramming and parallelism

As discussed earlier in the Parallelism and Concurrency chapter, threads and processes can be implemented to provide parallel programming in Python.

- As with `asyncio`, this only works with tasks that are I/O-bound, not those that require massive computation.

The `multiprocessing` module in the standard library provides both thread and process pool managers to easily parallelize a task.

# Distributed Computing

Writing Python code that can be distributed to multiple workers is a skill in and of itself.

- It requires the ability to program concurrently, avoiding deadlocks and starvation.
- As well as system administration, to ensure that the network of workers is healthy.

The simplest form of distributed computing will broadcast out code from a central machine to various other machines, which then run that code.

- Just like working with a large amount of data, it is best to test with a small amount of data and resources before attempting a larger run.

**ssh** is a perfectly serviceable tool for communicating between the various hosts.

- It does require that password-less key files are installed on each machine and used for authentication.

This content does not spend detail on configuring such an environment, only on using one after such a network of machines have been set up.

Generally, each worker node needs an identical environment (if not identical hardware) for the given process to execute in.

- This includes libraries, **ssh** keys, executables, packages, etc.

## SCOOP

SCOOP is the Scalable Concurrent Operations in Python library.

- It is designed for mass processing of data.

The general architecture is that a single source (the **broker**) divides up work to various workers or nodes.

- This is similar to how MapReduce divides up work.
- The data are divided among each worker, and the worker carries out the computation before feeding it back to the broker.

The actual implementation divides up the work into a binary tree, with the broker at the root node, and the leaf nodes are the resources carrying out the actual computation.

- Other internal nodes are reducing nodes that help manage the work.

While it is good practice in most Python code to have an `if __name__ == '__main__'` block, it is required in SCOOP.

- The Python script to be run through SCOOP is copied out to each machine, and run (but with a different value for `__name__`).
  - ▶ If the distributed script has any top-level code, it will be run on every machine!
  - ▶ This main-guard is vitally important when using SCOOP.

SCOOP is about parallel computation, using multiple resources.

- The most common application is to run a `map` job through the `futures` module of `scoop`.

scoop\_map.py

```
1 #!/usr/bin/env python
2 from random import randint
3 from scoop import futures
4
5
6 def main():
7     data = [randint(-1000, 1000) for _ in range(1000)]
8     serial = list(map(abs, data))
9     parallel = list(futures.map(abs, data))
10
11     assert serial == parallel
12
13
14 if __name__ == '__main__':
15     main()
```

The `futures.map` function will, like the builtin `map`, keep the ordering of any data iterated over.

- It processes each item from the input through the provided function.

The `futures.map_as_completed` function will instead return the data in the order it was processed, yielding results as soon as possible.

- Unless the order needs to be maintained, this is the more efficient function to use.

The `futures.reduce` function will, like the `functools.reduce` function, combine elements of the data pairwise using the provided function.

- Like the version in the standard library, it will process each pair sequentially from the input data.
- Unfortunately, there does not exist a `reduce` function that will apply work non-sequentially.

SCOOP will serialize both the function to execute and each item of data via `pickle`.

- This means that the function to be mapped must be a top-level function (no static class methods or local functions).
  - ▶ This is a limitation of the `pickle` module.

## Invoking SCOOP Code

SCOOP programs are invoked with the `-m scoop` library invoked on the command line to Python.

- Without this addition, the various `futures.*` functions decay to the builtin versions of `map` and `reduce`.
- A helpful warning is raised to this effect if the scoop library is not loaded at execution time.

Similarly to `mpi4py`, a number of other command-line flags can be specified to control how the work is distributed.

- `--host`
  - ▶ list of hostnames to use; specifying the same host multiple times will launch that many workers on it
- `--hostfile`
  - ▶ File of hosts to use; each line is of the form `<hostname>[ <number-of-workers>]`, default of 1 worker
- `-n`
  - ▶ Number of workers (be careful overriding the default of one worker per core!)

## SCOOP Globals

For large objects or data, it is better to share them via SCOOP's globals than letting them be pickled by default.

- Note that these must be constant!

*scoop\_globals.py*

```
1 #!/usr/bin/env python
2 from scoop import futures, shared
3
4
5 def main():
6     shared.setConst('value', 17)
7     print(sum(futures.map(add_to, range(10))))
8
9
10 def add_to(x):
11     val = shared.getConst('value')
12     return x + val
13
14
15 if __name__ == '__main__':
16     main()
```

As SCOOP is across multiple pieces of hardware, it may be that some parts encounter network or other communication hiccups.

- As such, especially for longer jobs, an optional timeout can be specified for retrieving a constant.

*scoop\_timeouts.py*

```
1 #!/usr/bin/env python
2 from scoop import futures, shared
3
4
5 def main():
6     shared.setConst('value', 17)
7     print(sum(futures.map(add_to, range(1000))))
8
9
10 def add_to(x):
11     try:
12         val = shared.getConst('value', timeout=5)
13         return x + val
14     except TimeoutError:
15         return 0
16
17
18 if __name__ == '__main__':
19     main()
```

# Message Passing Interface Using `mpi4py`

`mpi4py` is a Python module that follows the **Message Passing Interface** protocol.

- This requires some additional driver software, and python becomes just one of many possible distributed application platforms.
  - ▶ However, it allows for much finer-grained control than SCOOP.

A different executable is used to interact with the MPI framework.

- Python is merely a command to that framework, to be used across the various nodes.

```
> mpiexec python script.py
```

Unfortunately, writing Pythonic code is difficult with `mpi4py`.

- The MPI interface hews closely to MPI's C interface.
  - ▶ This does mean that developers familiar with that C API should be able to transition easily to `mpi4py`.

A program run under MPI generally needs to know its "rank".

- This is a numeric identifier as to the particular process being run.
  - ▶ So, a given MPI program is usually split out into logic for specific ranks (processes).

*messaging.py*

```
1 #!/usr/bin/env python
2 from mpi4py import MPI
3
4
5 def main():
6     comm = MPI.COMM_WORLD
7     print(f'Inside rank {comm.Get_rank()')
8
9
10 if __name__ == '__main__':
11     main()
```

The 0-rank process is generally treated as the root or broker.

- So, any aggregation of results is usually performed inside such a conditional block.

```

1 #!/usr/bin/env python
2 from mpi4py import MPI
3
4
5 def main():
6     comm = MPI.COMM_WORLD
7     rank = comm.Get_rank()
8     workers = comm.Get_size()
9
10    if rank == 0:
11        data = list(range(1000000))
12
13        for n in range(1, workers):
14            comm.send(data[n::workers], dest=n)
15
16        result = 0
17        for n in range(1, workers):
18            result += comm.recv(source=n)
19        print(result)
20    else:
21        data = comm.recv(source=0)
22        total = sum(data)
23        comm.send(total, dest=0)
24
25
26 if __name__ == '__main__':
27     main()

```

Like SCOOP, `mpi4py` allows the sending and receiving of any `pickle`-able object.

Unlike SCOOP, all the sends and the receives must be executed manually, rather than automatically with SCOOP's use of `futures`.

On the plus side, communicating between the different workers is controlled much more easily in `mpi4py`.

There are a few main patterns of such communications:

- `bcast`
  - ▶ sent from the root worker to all others
- `scatter`
  - ▶ divided up from the root worker to all others
- `gather`

- ▶ send from all others to the root worker
- **reduce**
  - ▶ combine from all others and return to the root worker

Thus, the previous program would more properly be written as:

*divided.messaging.py*

```

1 #!/usr/bin/env python
2 from mpi4py import MPI
3
4
5 def main():
6     comm = MPI.COMM_WORLD
7     rank = comm.Get_rank()
8     workers = comm.Get_size()
9
10    data = list(range(1000000))
11
12    slice = comm.scatter((data[n::workers] for n in range(workers)), root=0)
13    total = sum(slice)
14    result = comm.reduce(total, root=0)
15
16    if rank == 0:
17        print(result)
18
19
20 if __name__ == '__main__':
21    main()
```

The **reduce** method may take a named parameter of **op**, specifying the operation to reduce upon.

MPI provides some sample reducing functions, but any top-level python function (**pickleable**) may be used.

- **MPI.SUM**
  - ▶ Sum
- **MPI.PROD**
  - ▶ Product
- **MPI.MIN**
  - ▶ Minimum

- **MPI.MAX**
  - ▶ Maximum
- **MPI.LAND**
  - ▶ Logical AND
- **MPI.LOR**
  - ▶ Logical OR
- **MPI.BAND**
  - ▶ Bitwise AND
- **MPI.BOR**
  - ▶ Bitwise OR

The place where MPI really shines is the ability to provide a "map" of processes' connectedness.

- Many calculations are performed on some kind of coordinate plane, cube, or other dimensional coordinate system.
- MPI supports organizing the processes along those line (limited by the number of processes available to use).

*message\_grid.py*

```

1 #!/usr/bin/env python
2 from mpi4py import MPI
3
4
5 def main():
6     comm = MPI.COMM_WORLD
7     rank = comm.Get_rank()
8
9     # Create a 4x4 "grid" of processes
10    cartesian = comm.Create_cart(dims=(4, 4), periods=(False, False),
11                                reorder=True)
12    coords = cartesian.Get_coords(rank)
13
14    print(f'In 2D topology, {rank} handles {coords}')
15
16
17 if __name__ == '__main__':
18     main()
```

This approach is very useful for applications that can be parallelized easily.

- The given coordinate can access nearby coordinates via the `.Shift()` method.
  - ▶ This allows access to the rank-numbers of "adjacent" work centers, and then data may be shared or manipulated.

Finally, in order to better synchronize communications, it may be useful for all processes in an MPI job to hit a checkpoint before continuing.

- This can be accomplished with the `.Barrier()` method, which synchronizes all processes to that call.

---

# Exercises

## Exercise 1

Asymptotically calculate **pi** by randomly picking two numbers in the range [0, 1], squaring them, and seeing if the sum is less than or equal to 1, and keep a running ratio of the results.

- Do so for approximately 1,000,000 points.

## Exercise 2

An alphanumeric password known to be 5 characters long has the MD5 digest of: \*  
**7a3e645ce1b44e38852fba8561dd07b8** \*\* Find the original password.

## Exercise 3

The following page shows a simple implementation of the Game of Life.

- Modify the class so that it runs in a reasonable amount of time for sizes of at least 1,000.

## life.py

```

1 #!/usr/bin/env python
2
3
4 class Board:
5     def __init__(self, size):
6         self.data = [[False] * size for _ in range(size)]
7
8     def tick(self):
9         future = [x[:] for x in self.data]
10        for x in range(len(self.data)):
11            for y in range(len(self.data)):
12                neighbors = self.count_neighbors(x, y)
13                if neighbors < 2 or 4 < neighbors:
14                    future[x][y] = False
15                elif neighbors == 3:
16                    future[x][y] = True
17        self.data = future
18
19    def count_neighbors(self, x, y):
20        total = 0
21        for dx in range(max(0, x-1), min(x+2, len(self.data))):
22            for dy in range(max(0, y-1), min(y+2, len(self.data))):
23                # Ignore the square itself
24                if dx == x and dy == y:
25                    continue
26
27                total += 1 if self.data[dx][dy] else 0
28        return total
29
30    def __str__(self):
31        return '\n'.join(
32            ''.join(map(lambda y: '#' if y else '.', x))
33            for x in self.data)

```