Python Advanced

Building on the foundation

```python
course = {
    'title':
        'Python Advanced',
    'chapters': [
        'Introduction',
        'Special Methods & Attributes',
        'Unit Testing',
        'Function, Object & Class Customisation',
        'Object Thinking',
        'Containers',
        'Iterators & Generators',
        'Functional Thinking',
        'Concurrency',
        'Modules & Packages',
        'Outroduction',
        'Labs & Homework'
    ],
    ...
}
```

```
course = {
    ...
    'licence':
        ('Creative Commons Attribution 4.0, '
         'https://creativecommons.org/licenses/by/4.0/'),
    'contributors': [
        'Kevlin Henney, kevlin@curbralan.com',
    ],
    'date': '2015-11-05'
}
```

Introduction

Everything has a beginning…

Course goals

- **Increase knowledge of Python 3**
  - **This course builds on the themes and concepts of the *Python Foundation***
  - **Attendance on the *Python Foundation* course is assumed**
- **The course focus is on the language, the standard library and techniques**
  - **Using slides, discussion and hands-on experiments and labs**

Coding experiments

- **Small (or large) coding experiments to explore concepts in practice**
  - **Use the IDLE environment, the default *python* shell or any Python environment and editor combination of your choice!**
  - **Some experiments are described in the slides, but others may be suggested**
- **You can carry out experiments individually or in pairs**

Programming labs

- **Labs implemented using Cyber-Dojo**
  - ***http://cyber-dojo.org/***
  - **Labs involve writing both code and tests**
- **Labs are undertaken in pairs**
  - **But pairing is not marriage — you will move around**
- **Oh, and there's homework too**
  - **Individual rather than paired, and not in Cyber-Dojo**

Questions, answers & intros

- **The best way to have your questions answered is to ask them**
  - **Mindreading is not (yet) supported**
- **And now, please introduce yourself!**
  - **Briefly…**

Special Methods & Attributes

Integrating new types with operators & built-in functions

Facts at a glance

- **Special methods take the place of operator overloading in Python**
- **Built-in functions also build on special methods and special attributes**
- **The set of special names Python is aware of is predefined**
- **Context managers allow new classes to take advantage of *with* statements**

Operator overloading

- **It is not possible to overload operators in Python directly**
  - **There is no syntax for this**
- **But classes can support operators and global built-ins using special methods**
  - **They have reserved names, meanings and predefined relationships, and some cannot be redefined, e.g., *is* and *id***
  - **Often referred to as *magic methods***

Comparisons

Equality operations

If not defined for a class, == supports reference equality

```
__eq__(self, other)      self == other
__ne__(self, other)      self != other
```

If not defined for a class, *!=* is defined as *not self == other*

Ordering operations

```
__lt__(self, other)      self < other
__gt__(self, other)      self > other
__le__(self, other)      self <= other
__ge__(self, other)      self >= other
```

Note that the *__cmp__* special method is not supported in Python 3, so consider using the *functools.total_ordering* decorator to help define the rich comparisons

These methods return *NotImplemented* if not provided for a class, but whether *TypeError* is raised when an operator form is used depends on whether a reversed comparison is also defined, e.g., if *__lt__* is defined but *__gt__* is not, *a > b* will be executed as *b < a*

Arithmetic operations

Unary operators and built-in functions

`__neg__(self)`	`-self`
`__pos__(self)`	`+self`
`__invert__(self)`	`~self`
`__abs__(self)`	`abs(self)`

Binary arithmetic operators

Reflected forms also exist, e.g., *__radd__*, where method dispatch should be on the right-hand rather than left-hand operand

`__add__(self, other)`	`self + other`
`__sub__(self, other)`	`self - other`
`__mul__(self, other)`	`self * other`
`__truediv__(self, other)`	`self / other`
`__floordiv__(self, other)`	`self // other`
`__mod__(self, other)`	`self % other`
`__pow__(self, other)`	`self**other`
`...`	

Augmented assignment

Note that if these special methods are not defined, but the corresponding binary operations are, these assignment forms are still valid, e.g., if *__add__* is defined, then *lhs += rhs* becomes *lhs = lhs.__add__(rhs)*

Augmented assignment operators

```
__iadd__(self, other)         self += other
__isub__(self, other)         self -= other
__imul__(self, other)         self *= other
__itruediv__(self, other)     self /= other
__ifloordiv__(self, other)    self //= other
__imod__(self, other)         self %= other
__ipow__(self, other)         self **= other
...
```

Each method should return the result of the in-place operation (usually *self*) as the result is used for the actual assignment, i.e., *lhs += rhs* is equivalent to *lhs = lhs.__iadd__(rhs)*

Conversions & representation

Numeric type conversions

```
__int__(self)
__float__(self)
__round__(self)
__complex__(self)
__bool__(self)
__hash__(self)
```

And also *__round__(self, n)* to round to *n* digits after the decimal point (or before if negative)

Should only define if *__eq__* is also defined, such that objects comparing equal have the same hash code

String type conversions

If *__str__* is not defined, *__repr__* is used by *str* if present

Official string representation, that should ideally be a valid Python expression that constructs the value

```
__str__(self)
__format__(self, format)
__repr__(self)
__bytes__(self)
```

Containers & iteration

Content query

```
__len__(self)                  len(self)
__contains__(self, item)       item in self
```

If *__contains__* not defined, *in* uses linear search based on *__iter__* or indexing from 0 of *__getitem__*

Iteration

Most commonly invoked indirectly in the context of a *for* loop

```
__iter__(self)    iter(self)
__next__(self)    next(self)
```

Subscripting

```
__getitem__(self, key)          self[key]
__setitem__(self, key, item)    self[key] = item
__delitem__(self, key)          del self[key]
__missing__(self, key)
```

Hook method used by *dict* for its subclasses

Interception & lifecycle

Support for function calls and attribute access

```
__call__(self, ...)
__getattr__(self, name)
__getattribute__(self, name)
__setattr__(self, name, value)
__delattr__(self, name)
__dir__(self)
```

Object lifecycle

```
__init__(self, ...)
__del__(self)
```

Context management

```
__enter__(self)
__exit__(self, exception, value, traceback)
```

with & context managers

- The *with* statement provides exception safety for paired actions
  - I.e., where an initial acquire/open/lock action must be paired with a final release/close/unlock action, regardless of exceptions
- A context manager is an object that can be used in this way in a *with*
  - It is defined by a simple protocol

Context manager anatomy

Called on entry to *with*

Result bound to the target of the *with*

```
class Resource:
    def __enter__(self):
        ...
        return self
    def __exit__(self, exception, value, trace):
        ...
```

Exception string or *None* if no exception

Traceback or *None* if no exception

Exception class or *None* if no exception

```
with Resource() as resource:
    ...
```

Note that *resource* still accessible after the *with* statement

Experiment

Write a context manager to time the execution of code within a *with* statement

```
with Timing():
    task_to_time()
```

Code to time the execution of

```
from time import time

class Timing:
    def __enter__(self):
        ...
    def __exit__(self, exception, value, trace):
        ...
```

Record current time using *time*

Print elapsed time, optionally indicating whether an exception was thrown

Special attributes

- **A number of special attributes can also be defined, including...**
  - **__*doc*__ is the documentation string and is used by the built-in *help* function**
  - **__*name*__ is used to hold the name for functions and classes**
  - **__*dict*__ refers to the dictionary of values held by any object**
  - **__*module*__ names the defining module**

Unit Testing

Programming *unittest* with GUTs

Facts at a glance

- **Python has many testing frameworks available, including *unittest***
- ***unittest* is derived from JUnit and has a rich vocabulary of assertions**
- **Good Unit Tests (GUTs) are structured around explanation and intention**
- **The outcome of a unit test depends solely on the code and the tests**

unittest

- **The *unittest* framework is a standard JUnit-inspired framework**
  - **It is organised around test case classes, which contain test case methods**
  - **Naming follows Java camelCase**
  - **Runnable in other frameworks, e.g., *pytest***
- **Python has no shortage of available testing frameworks!**
  - **E.g., *doctest*, *nose*, *pytest***

Good Unit Tests (GUTs)

Very many people say "TDD" when they really mean, "I have good unit tests" ("I have GUTs"?). Ron Jeffries tried for years to explain what this was, but we never got a catch-phrase for it, and now TDD is being watered down to mean GUTs.

Alistair Cockburn

http://alistair.cockburn.us/The+modern+programming+professional+has+GUTs

GUTs in practice

- Good unit tests (GUTs)...
  - Are fully automated, i.e., write code to test code
  - Offer good coverage of the code under test, including boundary cases and error-handling paths
  - Are easy to read and to maintain
  - Express the intent of the code under test — they do more than just check it

Not-so-good unit tests

- **Problematic test styles include...**
  - ***Monolithic tests*: all test cases in a single function, e.g., *test***
  - ***Ad hoc tests*: test cases arbitrarily scattered across test functions, e.g., *test1*, *test2*, ...**
  - ***Procedural tests*: test cases bundled into a test method that correspond to target method, e.g., *test_foo* tests *foo***

A *unittest* example

Test case method names must start with *test*

Derivation from base class is necessary

```
from leap_year import is_leap_year
import unittest

class LeapYearTests(unittest.TestCase):

    def test_that_years_not_divisible_by_4_are_not_leap_years(self):
        self.assertFalse(is_leap_year(2015))

    def test_that_years_divisible_by_4_but_not_by_100_are_leap_years(self):
        self.assertTrue(is_leap_year(2016))

    def test_that_years_divisible_by_100_but_not_by_400_are_not_leap_years(self):
        self.assertFalse(is_leap_year(1900))

    def test_that_years_divisible_by_400_are_leap_years(self):
        self.assertTrue(is_leap_year(2000))


if __name__ == '__main__':
    unittest.main()
```

Assertions

```
assertEqual(lhs, rhs)
assertNotEqual(lhs, rhs)
assertTrue(result)
assertFalse(result)
assertIs(lhs, rhs)
assertIsNot(lhs, rhs)
assertIsNone(result)
assertIsNotNone(result)
assertIn(lhs, rhs)
assertNotIn(lhs, rhs)
assertIsInstance(lhs, rhs)
assertNotIsInstance(lhs, rhs)
...
```

All assertions also take an optional *msg* argument

More assertions

```
assertLess(lhs, rhs)
assertLessEqual(lhs, rhs)
assertGreater(lhs, rhs)
assertGreaterEqual(lhs, rhs)
assertRegex(result)
assertNotRegex(result)
assertRaises(exception)
assertRaises(exception, callable, *args, *kwargs)
fail()
assertAlmostEqual(lhs, rhs)
assertAlmostNotEqual(lhs, rhs)
...
```

By default, approximate equality is established to within 7 decimal places — this can be changed using the *places* keyword argument or by providing a *delta* keyword argument

assertRaises

- ***assertRaises*** **can be used as an ordinary assertion call or with *with***
  - **If a callable argument is not provided, *assertRaises* returns a context manager**
  - **As a context manager, *assertRaises* fails if associated *with* body does not *raise***

```
self.assertRaises(ValueError, lambda: is_leap_year(-1))
```

```
with self.assertRaises(ValueError) as context:
    is_leap_year(-1)
```

Optional, but can be used to access raised exception details

assertRaisesRegex

- ***assertRaisesRegex*** **also matches string representation of raised exception**
  - **Equivalent to regex *search***
  - **Like *assertRaises*, *assertRaisesRegex* can be used as function or context manager**

```
self.assertRaisesRegex(
    ValueError, 'invalid', lambda: is_leap_year(-1))
```

```
with self.assertRaisesRegex(ValueError, 'invalid') :
    is_leap_year(-1)
```

Tests as explanation

So who should you be writing the tests for? For the person trying to understand your code.

Good tests act as documentation for the code they are testing. They describe how the code works. For each usage scenario, the test(s):

- Describe the context, starting point, or preconditions that must be satisfied
- Illustrate how the software is invoked
- Describe the expected results or postconditions to be verified

Different usage scenarios will have slightly different versions of each of these.

Gerard Meszaros
"Write Tests for People"

Test anatomy

- **Example-based tests ideally have a simple linear flow: arrange, act, assert**
  - ***Given*: set up data**
  - ***When*: perform the action to be tested**
  - ***Then*: assert desired outcome**
- **Tests should be short and focused**
  - **A single objective or outcome — but not necessarily a single assertion — that is reflected in the name**

Common fixture code

- **Common test fixture code can be factored out...**
  - **By defining *setUp* and *tearDown* methods that are called automatically before and after each test case execution**
  - **By factoring out common initialisation code, housekeeping code or assertion support code into its own methods, local to the test class**

Tests as specifications

Tests that are not written with their role as specifications in mind can be very confusing to read. The difficulty in understanding what they are testing can greatly reduce the velocity at which a codebase can be changed.

Nat Pryce & Steve Freeman
"Are your tests really driving your development"

What *is* a unit test?

A test is not a unit test if:

- It talks to the database
- It communicates across the network
- It touches the file system
- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment (such as editing config files) to run it.

Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness. However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes.

Michael Feathers

http://www.artima.com/weblogs/viewpost.jsp?thread=126923

External dependencies

- External resources are a common source of non-unit dependencies
  - Registries and environment variables
  - Network connections and databases
  - Files and directories
  - Current date and time
  - Hardware devices
- Externals can often be replaced by test doubles or pre-evaluated objects

A taxonomy of test doubles

- **It's not just about mocks...**
  - **A *test stub* is used to substitute input and can be used for fault injection**
  - **A *test spy* offers input and captures output behaviour**
  - **A *mock object* validates expectations**
  - **A *fake object* offers a usable alternative to a real dependency**
  - **A *dummy object* fulfils a dependency**

Doubles in practice

- **Mocking and related techniques are easier in Python than many languages**
  - **Duck typing means dependencies can be substituted without changing class design**
  - **Passing functions as objects allows for simple pluggability**
  - **It is easy to write interception code for new classes and to add interception code on existing objects or use existing libraries, e.g., *unittest.mock***

Function, Object & Class Customisation

Ad hoc objects, metaclasses, monkey patching & decorators

Facts at a glance

- **Functions are objects with state**
- **Classes are mutable after definition, so methods can be added and removed**
- **Objects can have attributes added and removed after creation**
- **Decorators wrap functions on definition**
- **A metaclass is the class of a class, defining how the class behaves**

It's objects all the way down

- **Everything in Python is expressed as objects, from functions to modules**
  - **Functions are *callable* objects — with associated state and metadata — and are instantiated with a *def* statement**
- **And Python is a very dynamic language, not just dynamically typed**
  - **Python's object model can be fully accessed at runtime**

Object creation

- **Objects are normally created based on calling their class name**
- **But the class name does not have to be present, only the class object**
  - **E.g., *type(parrot)()* creates a new object with the same type as *parrot***
  - **This eliminates the need for some of the factory patterns and convoluted tricks employed in other languages**

Ad hoc data structures

- **Keyword arguments make *dicts* easy to use as ad hoc data structures**
  - **Keyword arguments are passed as *dicts*, with keywords becoming keys**
  - **See also *collections.namedtuple***

```
sputnik_1 = dict(year=1957, month=10, day=4)
         {'day': 4, 'month': 10, 'year': 1957}
origin = dict(x=0, y=0)
                             {'y': 0, 'x': 0}
```

Dynamic object attributes

- **It is possible to add attributes to an object after it has been created**
  - **Object attributes are dynamic and implemented as a dictionary**
  - **An attribute can be removed using *del***

Ad hoc data structures with named attributes can be created easily

```
class Spam:
    pass
meal = Spam()
meal.egg = True
meal.bacon = False
```

Monkey patching

A **monkey patch** is a way for a program to extend or modify supporting system software locally (affecting only the running instance of the program). This process has also been termed **duck punching**.

The definition of the term varies depending upon the community using it. In Ruby, Python, and many other dynamic programming languages, the term *monkey patch* only refers to dynamic modifications of a class or module at runtime.

http://en.wikipedia.org/wiki/Monkey_patch

Modifying existing classes

- **Methods and attributes can be attached to existing classes**
  - **But be careful doing so!**

```
class Menu:
    pass

menu = Menu()

Menu.options = {'spam', 'egg', 'bacon', 'sausage'}
def order(self, *choice):
    return set(choice) <= self.options
Menu.order = order

assert menu.order('egg')
assert not menu.order('toast')
```

Exploring the object model

- **A number of functions help to explore the runtime object model**
  - ***dir*: a sorted list of strings of names in the current or specified scope**
  - ***locals*: *dict* of current scope's variables**
  - ***globals*: dict of current scope's global variables**
  - ***vars*: *dict* of an object's state, i.e., *__dict__*, or *locals* if no object specified**

Object attribute operations

- **Object attributes can be accessed using *hasattr*, *getattr* and *setattr***
  - **These global functions are preferred to manipulating an object's *__dict__* directly**
- **An object can provide special methods to intercept attribute access**
  - **Query is via *__getattr__* (if not present) and *__getattribute__* (unconditional)**
  - ***Modify* via *__setattr__* and *__delattr__***

Decorators

- A function definition may be wrapped in decorator expressions
  - A decorator is a callable object that results in a callable object given a callable object or specified arguments
  - On definition the function is passed to the decorator and the decorator's result is used in place of the function
  - This forms a chain of wrapping if more than one decorator is specified

Basic decoration

```
def non_zero_args(wrapped):
    def wrapper(*args):
        if all(args):
            return wrapped(*args)
        else:
            raise ValueError
    return wrapper
```

A nested function that performs the validation, executing the wrapped function with arguments if successful *(Note: for brevity, keyword arguments are not handled)*

```
@non_zero_args
def date(year, month, day):
    return year, month, day
```

Decorator

```
date(1957, 10, 4)
date(2000, 0, 0)
```

Returns *(1957, 10, 4)*

Raises *ValueError*

Refined decoration

```
from functools import wraps
def non_zero_args(wrapped):
    @wraps(wrapped)
    def wrapper(*args):
        if all(args):
            return wrapped(*args)
        else:
            raise ValueError
    return wrapper
```

Ensures the resulting wrapper has the same principal attributes as the wrapped function, e.g., __*name*__ and __*doc*__

```
@non_zero_args
def date(year, month, day):
    return year, month, day
```

Parameterised decoration

```
def check_args(checker):
    def decorator(wrapped):
        def wrapper(*args):
            if checker(args):
                return wrapped(*args)
            else:
                raise ValueError
        return wrapper
    return decorator
```

Nested function that curries the *checker* argument

Nested nested function that performs the validation

```
@check_args(all)
def date(year, month, day):
    return year, month, day
```

```
date(1957, 10, 4)
date(2000, 0, 0)
```

Returns *(1957, 10, 4)*

Raises *ValueError*

Metaclasses

- A metaclass can be considered the class of class
  - A class defines how objects behaves; a metaclass defines how classes behaves
- Metaclasses are most commonly used as class factories
  - Customise class creation and execution
  - A class is created from a triple of name, base classes and a dictionary of attributes

abc.ABCMeta

Use of the *ABCMeta* metaclass to define abstract classes is one of the more common metaclass examples

Instantiation for *Visitor* is disallowed because of *ABCMeta* and the presence of at least one *@abstractmethod*

@abstractmethod only has effect if *ABCMeta* is the metaclass

```
from abc import ABCMeta
from abc import abstractmethod

class Visitor(metaclass=ABCMeta):
    @abstractmethod
    def enter(self, visited):
        pass
    @abstractmethod
    def exit(self, visited):
        pass
    @abstractmethod
    def visit(self, value):
        pass
```

type

- *type* is the default metaclass in Python
  - And *type* is its own class, i.e., *type*(*type*) is an identity operation
  - *type* is most often used as a query, but it can also be used to create a new type

```
def init(self, options):
    self.__options = options
def options(self):
    return self.__options
Menu = type('Menu', (),
            {'__init__': init, 'options': options})
menu = Menu({'spam', 'egg', 'bacon', 'sausage'})
```

The base defaults to *object*

Adding new code at runtime

- **New code, from source, can be added to the Python runtime**
  - **The *compile* function returns a code object from a module, statement or expression**
  - **The *exec* function executes a code object**
  - **The *eval* function evaluates source code and returns the resulting value**
- **Here be dragons!**

Object Thinking

Protocols, polymorphism, patterns & practical advice

Facts at a glance

- **Object usage defined more by protocols than by class relationships**
- **Substitutability is strong conformance**
- **Polymorphism is a fundamental consideration (but inheritance is not)**
- **Values can be expressed as objects following particular conventions**
- **Enumeration types have library support**

Not all objects are equal

- **Be careful how you generalise your experience!**
  - **Patterns of practice are context sensitive**
- **Python's type system means that...**
  - **Some OO practices from other languages and design approaches travel well**
  - **Some do not translate easily or well — or even at all**
  - **And some practices are specific to Python**

Protocols and conformance

- **An expected set of method calls can be considered to form a *protocol***
  - **Protocols may be named informally, but they are not part of the language**
- **Conformance to a protocol does not depend on inheritance**
  - **Dynamic binding means polymorphism and subclassing are orthogonal**
  - **Think *duck types* not *declared types***

Liskov Substitution Principle

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov
"Data Abstraction and Hierarchy"

Contracts & consequences

- **Substitutability is the strongest form of conformance to a protocol**
  - **A contract of behavioural equivalence**
- **Following the contract means...**
  - **An overridden method cannot result in a wider range or cover a narrower domain**
  - **Concrete classes should not be bases**
  - **A derived class should support the same initialisation protocol as its base**

Inheritance & composition

- A class can respect logical invariants
  - Assertions true of its methods and state
- A derived class should respect the invariants of its base classes
  - And may strengthen them
- Consider composition and forwarding
  - Especially if derivation would break invariants and lead to method spam from the base classes

Object structure & traversal

- **The following patterns combine and complement one another:**
  - ***Composite*: recursive–whole part structure**
  - ***Visitor*: dispatch based on argument type**
  - ***Enumeration Method*: iteration based on inversion of control flow**
  - ***Lifecycle Callback*: define actions for object lifecycle events as callbacks**

Composite pattern

```
class Node(metaclass=ABCMeta):
    @abstractmethod
    def children(self):
        pass
    def name(self):
        return self._name
    ...
```

```
class Primitive(Node):
    def __init__(self, name):
        self._name = name
    def children(self):
        return ()
    ...
```

```
class Group(Node):
    def __init__(self, name, children):
        self._name = name
        self._children = tuple(children)
    def children(self):
        return self._children
    ...
```

Visitor pattern

```
class Node(metaclass=ABCMeta):
    ...
    @abstractmethod
    def accept(self, visitor):
        pass
    ...
```

```
class Primitive(Node):
    ...
    def accept(self, visitor):
        visitor.visit_primitive(self)
    ...
```

```
class Group(Node):
    ...
    def accept(self, visitor):
        visitor.visit_group(self)
    ...
```

```
class Visitor:
    def visit_primitive(self, visited):
        ...
    def visit_group(self, visited):
        ...
```

Enumeration Method pattern

```
class Node(metaclass=ABCMeta):
    @abstractmethod
    def map(self, callback):
        pass
    ...
```

*

```
class Primitive(Node):
    ...
    def map(self, callback):
        pass
    ...
```

```
class Group(Node):
    def __init__(self, name, children):
        self._name = name
        self._children = tuple(children)
    def map(self, callback):
        for child in self._children:
            callback(child)
    ...
```

Lifecycle Callback pattern

```
class Visitor:
    def enter(self, group):
        ...
    def leave(self, group):
        ...
    def visit(self, primitive):
        ...
```

```
class Printer:
    def __init__(self):
        self.__depth = 0

    def enter(self, group):
        print(self.__depth * ' ' + '<group>')
        self.__depth += 1

    def leave(self, group):
        self.__depth -= 1
        print(self.__depth * ' ' + '</group>')

    def visit(self, primitive):
        print(self.__depth * ' ' + '<primitive/>')
```

Patterns in combination

```
class Node(metaclass=ABCMeta):
    @abstractmethod
    def for_all(self, visitor):
        pass
    ...
```

*

```
class Primitive(Node):
    ...
    def for_all(self, visitor):
        visitor.visit(self)
    ...
```

```
class Group(Node):
    ...
    def for_all(self, visitor):
        visitor.enter(self)
        for child in self._children:
            child.for_all(visitor)
        visitor.leave(self)
    ...
```

Object forwarding

- **Many object forwarding patterns have a simple and general form in Python**
  - ***Proxy*: offer transparent forwarding by one object to another, supporting the target object's protocol**
  - ***Null Object*: represent absence of an object with an object that realises the object protocol with do-nothing or return-default implementations for its methods**

Proxy pattern

```
class TimingProxy:
    def __init__(self, target, report=print):
        self.__target = target
        self.__report = report
    def __getattr__(self, name):
        attr = getattr(self.__target, name)
        def wrapper(*args, **kwargs):
            start = time()
            try:
                return attr(*args, **kwargs)
            finally:
                end = time()
                self.__report(name, end - start)
        return wrapper if callable(attr) else attr
```

Null Object pattern

A generic Null Object implementation useful, for example, as for test dummy objects, particularly if no specific return values are expected or tested

```
class NullObject:
    def __call__(self, *args, **kwargs):
        return self
    def __getattr__(self, name):
        return self
```

Method calls on *NullObject* are chainable as *NullObject* is callable

Realising values as objects

- **It is worth differentiating between objects that represent mechanisms...**
  - **And may therefore have changing state**
- **And objects that represent values**
  - **Their focus is information (e.g., quantities), rather than being strongly behavioural (e.g., tasks), or entity-like (e.g., users)**
  - **They should be easily shared and consistent, hence immutable (i.e., like *str*)**

Immutable Value pattern

- **For a class of value objects...**
  - **Provide for rich construction to ensure well-formed objects are easy to create**
  - **Provide query but not modifier methods**
  - **Consider *@property* for zero-parameter query methods so they look like attributes**
  - **Define *__eq__* and *__hash__* methods**
  - **Provide relational operators if values are ordered (see *functools.total_ordering*)**

In the money

```
@total_ordering
class Money:
    def __init__(self, units, hundredths):
        self.__units = units
        self.__hundredths = hundredths

    @property
    def units(self):
        return self.__units

    @property
    def hundredths(self):
        return self.__hundredths

    def __eq__(self, other):
        return (self.units == other.units and
                self.hundredths == other.hundredths)

    def __lt__(self, other):
        return ((self.units, self.hundredths) <
                (other.units, other.hundredths))
    ...
```

On the money

```
@total_ordering
class Money:
    def __init__(self, units, hundredths):
        self.__total_hundredths = units * 100 + hundredths

    @property
    def units(self):
        return self.__total_hundredths // 100

    @property
    def hundredths(self):
        return self.__total_hundredths % 100

    def __eq__(self, other):
        return (self.units == other.units and
                self.hundredths == other.hundredths)

    def __lt__(self, other):
        return ((self.units, self.hundredths) <
                (other.units, other.hundredths))
    ...
```

Enumeration types

- **Python (as of 3.4) supports enum types**
  - **They are similar — but also quite different — to enum types in other languages**
  - **Support comes from the *enum* module**
- **An enumeration type must inherit from either *Enum* or *IntEnum***
  - ***Enum* is the base for pure enumerations**
  - ***IntEnum* is the base class for integer-comparable enumerations**

Enumerated

```
from enum import Enum
class Suit(Enum):
    spades = 1
    hearts = 2
    diamonds = 3
    clubs = 4
```

Use *IntEnum* if easily comparable enumerations are needed

Enumeration names are accessible as strings are accessible via the *name* property and the integer via *value*

```
values = ['A'] + list(range(2, 11)) + ['J', 'Q', 'K']
[(value, suit.name) for suit in Suit for value in values]
```

Enumeration types are iterable

Ad hoc enums

- ***Enum*** **and** ***IntEnum*** **can be used to create enums on the fly**
  - **They are both** ***callable***

```
Colour = Enum('Colour', 'red green blue')
```

```
Colour = Enum('Colour', ('red', 'green', 'blue'))
```

```
Colour = Enum('Colour', {'red': 1, 'green': 2, 'blue': 3})
```

```
class Colour(Enum):
    red = 1
    green = 2
    blue = 3
```

Containers

Built-in containers, library collections, tips & tricks

Facts at a glance

- **Built-in containers address most needs**
- **There are some common iteration patterns to follow (and avoid)**
- **Comprehensions address many common container iteration needs**
- ***collections* module offers variations on standard sequence and lookup types**
- ***collections.abc* supports container usage and definition**

Container guidance

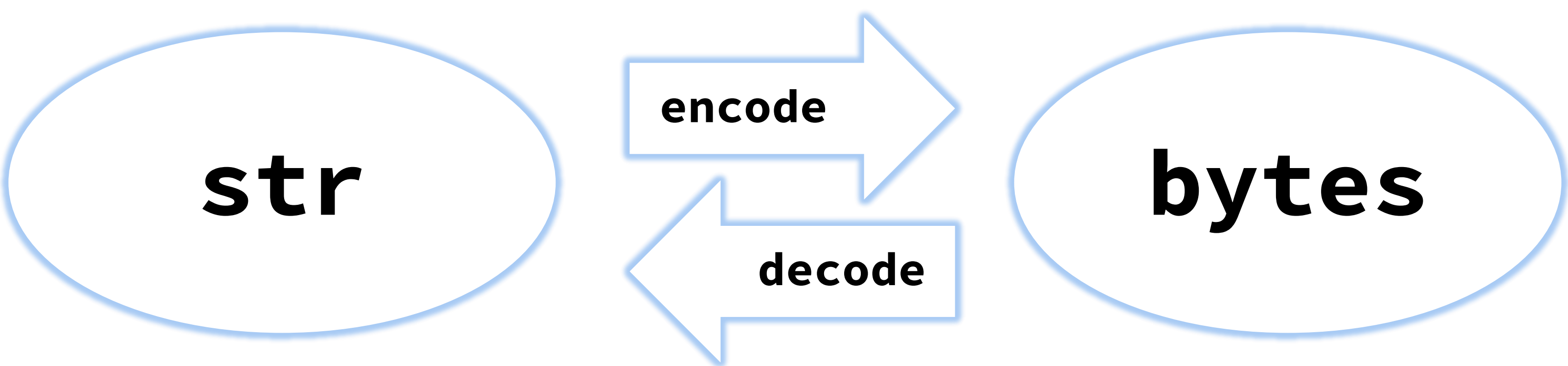
- **Although everything is, at one level, a *dict*, *dict* is not always the best choice**
  - **Nor is *list***
- **Choose containers based on usage patterns and mutability**
  - **E.g., *tuple* is immutable whereas *list* is not**
  - **Iteration is the most common container activity, so favour the most direct and efficient iteration style**

Built-in sequence types

list	`list()`
	`[]`
	`[0, 0x33, 0xCC]`
	`['Albert', 'Einstein', [1879, 3, 14]]`
	`[random() for _ in range(42)]`
tuple	`tuple()`
	`()`
	`(42,)`
	`('red', 'green', 'blue')`
	`((0, 0), (3, 4))`
range	`range(42)`
	`range(1, 100)`
	`range(100, 0, -1)`

From *str* to *bytes* & back

- ***bytes*** **and** ***str*** **are immutable sequence types used for holding strings**
  - ***str*** **is Unicode and** ***bytes*** **holds, well, bytes**
  - **By default, encoding and decoding between** ***str*** **and** ***bytes*** **is based on UTF-8**



Built-in lookup types

```
dict       dict()
           {}
           {'Bohr': True, 'Einstein': False}
           {n: n**2 for n in range(0, 100)}
set        set()
           {'1st', '2nd', '3rd'}
           {n**2 for n in range(1, 100)}
frozenset  frozenset()
           frozenset({15, 30, 40})
           frozenset(n**2 for n in range(0, 100))
```

How not to iterate

```
currencies = {
    'EUR': 'Euro',
    'GBP': 'British pound',
    'NOK': 'Norwegian krone',
}
```

```
for code in currencies:
    print(code, currencies[code])
```

```
ordinals = ['first', 'second', 'third']
```

```
for index in range(0, len(ordinals)):
    print(ordinals[index])
```

```
for index in range(0, len(ordinals)):
    print(index + 1, ordinals[index])
```

How to iterate

```
currencies = {
    'EUR': 'Euro',
    'GBP': 'British pound',
    'NOK': 'Norwegian krone',
}
```

```
for code, name in currencies.items():
    print(code, name)
```

```
ordinals = ['first', 'second', 'third']
```

```
for ordinal in ordinals:
    print(ordinal)
```

```
for index, ordinal in enumerate(ordinals, 1):
    print(index, ordinal)
```

Comprehensive containers

List comprehension

Set comprehension

```
[n for n in range(2, 100)
    if n not in
        {m for l in range(2, 10)
            for m in range(l*2, 100, l)}]
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
                         79, 83, 89, 97]
```

collections

- **Built-in container types meet many common and initial needs**
  - **But sometimes a different data structure will cut down the amount of code written or the runtime resources used**
- **Built-in container types supplemented by the standard *collections* module**
  - **Container types in *collections* are not built in, so they do not have display forms**

defaultdict

- ***defaultdict*** **offers on-demand creation for keys that are not already present**
  - **Derives from *dict***
  - **Uses a given factory function, such as a class name, to create default values**

```
def histogram(data):
    result = defaultdict(str)
    for item in data:
        result[word] += '#'
    return result
```

Inserts a value of *str()* for each key looked up that is not already present

Counter

- ***Counter*** **tracks occurrences of a key**
  - **It derives from *dict* but behaves a lot like a bag or a multiset**
  - **Counts can be looked up for a key**
  - **Each key 'occurrence' can be iterated**

```
with open(filename) as input:
    words = input.read().split()
counted = Counter(votes)
most_common = counted.most_common(1)
most_to_least_common = counted.most_common()
```

OrderedDict

- ***OrderedDict* preserves the order in which keys were inserted**
  - **Derives from *dict* (with some LSP issues...)**
  - **One of the most useful applications is to create sorted dictionaries, i.e., initialise from result of *sorted* on a *dict***

```
def word_counts(words):
    counts = OrderedDict()
    for word in words:
        counts[word] = 1+ counts.setdefault(word, 0)
return counts
```

ChainMap

- *ChainMap* provides a view over a number of mapping objects
  - Lookups are in sequence along the chain
  - Updates apply only to the first mapping — avoid side effects by supplying *{}*

```
defaults = {'drink': 'tea', 'snack': 'biscuit'}
preferences = {'drink': 'coffee'}
options = ChainMap(preferences, defaults)
```

```
for key, value in options.items():
    print(key, value, sep=': ')
```

```
drink: coffee
snack: biscuit
```

deque

- **A *deque* is a double-ended queue**
  - **Supports efficient *append* and *pop* and *appendleft* and *popleft* operations**
  - **Can be bounded, with overflow causing a pop from the opposite end to the append**

```
def tail(filename, lines=10):
    with open(filename) as source:
        return deque(source, lines)
```

namedtuple

- ***namedtuple*** **allows the creation of a tuple type with named attributes**
  - **Can still be indexed and iterated**
  - ***namedtuple*** **is subclass of** ***tuple***

```
Date = namedtuple('Date', 'year month day')
```

```
Date = namedtuple('Date', ('year', 'month', 'day'))
```

```
sputnik_1 = Date(1957, 10, 4)
sputnik_1 = Date(year=1957, month=10, day=4)
year, month, day = sputnik_1
year = sputnik_1.year
month = sputnik_1[1]
```

collections.abc

- **The *collections.abc* module provides abstract container classes**
  - **Using *isinstance*, these can be used to check whether an object conforms to a protocol, e.g., *Container*, *Hashable*, *Iterable*, *MutableSequence*, *Set***
  - **These abstract classes can also be used as mixin base classes for new container classes**

Iterators & Generators

Iteration abstractions, iterators, generators & wrapped control

Facts at a glance

- **Iterators enjoy direct protocol and control structure support in Python**
- **Iterators can be defined as classes**
- **Generators are functions that automatically create iterators**
- **Generator expressions support a simple way of generating generators**
- **Generators can abstract *with* logic**

Iteration

- **An iterator is an object that iterates, following the iterator protocol**
  - **An iterable object can be used with *for***
- **Iterators and generators are lazy, returning values on demand**
  - **You don't need to resolve everything into a list in order to use a series of values**
  - **Yield values in sequence, so can linearise complex traversals, e.g., tree structures**

Iterables & iterators

An object is iterable if it supports the __*iter*__ special method, which is called by the *iter* function

```
class Iterable:
    ...
    def __iter__(self):
        return Iterator(...)
    ...
```

All iterators are iterable, so the __*iter*__ method is an identity operation

The __*next*__ special method, called by *next*, advances the iterator

Iteration is terminated by raising *StopIteration*

```
class Iterator:
    ...
    def __iter__(self):
        return self
    def __next__(self):
        ...
            raise StopIteration
        ...
    ...
```

Defining iterators

- **There are many ways to provide an iterator...**
  - **Define a class that supports the iterator protocol directly**
  - **Return an iterator from another object**
  - **Compose an iterator with *iter*, using an action and a termination value**
  - **Define a generator function**
  - **Write a generator expression**

iter

- **Use *iter* to create an iterator from a callable object and a sentinel value**
  - **Or to create an iterator from an iterable**

```
def pop_until(stack, end):
    return iter(stack.pop, end)

for popped in pop_until(history, None):
    print(popped)
```

```
def repl():
    for line in iter(lambda: input('> '), 'exit'):
        print(evaluate(line))
```

next

- **Iterators can be advanced manually using *next***
  - **Calls the __*next*__ method**
  - **Watch out for *StopIteration* at the end...**

```
def repl():
    try:
        lines = iter(lambda: input('> '), 'exit')
        while True:
            line = next(lines)
            print(evaluate(line))
    except StopIteration:
        pass
```

Generator expressions

- A comprehension-based expression that results in an iterator object
  - Does not result in a container of values
  - Must be surrounded by parentheses unless it is the sole argument of a function
  - May be returned as the result of a function

```
numbers = (random() for _ in range(42))
sum(numbers)
```

```
sum(random() for _ in range(42))
```

Generator functions

- **A generator is an ordinary function that returns an iterator as its result**
  - **The presence of a *yield* or *yield from* makes a function a generator, and can only be used within a function**
  - ***yield* returns a single value**
  - ***yield from* takes values from another iterator, advancing by one on each call**
  - ***return* in a generator raises *StopIteration*, passing it any return value specified**

Yielding a winner

```
for medal in medals():
    print(medal)
```

```
def medals():
    yield 'Gold'
    yield 'Silver'
    yield 'Bronze'
```

```
def medals():
    for result in 'Gold', 'Silver', 'Bronze':
        yield result
```

```
def medals():
    yield from ['Gold', 'Silver', 'Bronze']
```

@contextmanager

- **The *contextmanager* decorator offers simple definition of context managers**
  - **Defined in the standard *contextlib* module**
- **Decorate a generator that holds a single *yield* statement**
  - **The entry action comes before the *yield***
  - **The exit action follows the *yield***
  - **If *yield* has a value it will be bound to the target of the *with***

It's all about timing

```
class timing:
    def __init__(self, report=print):
        self.__report = report
    def __enter__(self):
        self.__start = time()
    def __exit__(self, *args):
        self.__report(time() - self.__start)
```

```
with timing():
    task_to_time()
```

```
@contextmanager
def timing(report=print):
    start = time()
    yield
    report(time() - start)
```

Trading places

```
print(os.getcwd())
with pushd('/tmp'):
    print(os.getcwd())
    task()
print(os.getcwd())
```

Should print '*/tmp*'

Should print the same directory before and after the *with*

```
@contextmanager
def pushd(new):
    old = os.getcwd()
    os.chdir(new)
    try:
        yield
    finally:
        os.chdir(old)
```

Without the *finally*, any exception would propagate out from the *yield*, terminating the generator and bypassing the directory restore action

Experiment

- **Define a generator that returns successive days from a start date**
  - **Use *date* and *timedelta* from *datetime***
  - **Consider using *date.today()* as default**
- **Use it to explore different ways of selecting dates that fall on Friday 13th**
  - **Use a generator expression**
  - **Use a container comprehension**
  - **Use *filter***

Functional Thinking

Functional programming techniques, tips & tricks

Facts at a glance

- **Functions are the principal units of composition**
- **Functions are pure, i.e., no side effects**
- **Immutability, i.e., query and create state rather than modify it**
- **Declarative over imperative style, e.g., use of comprehensions over loops**
- **Deferred evaluation, i.e., be lazy**

Functional guidance

- **Immutability...**
  - **Prefer to return new state rather than modifying arguments and attributes**
  - **Resist the temptation to match every query or *get* method with a modifier or *set***
- **Expressiveness...**
  - **Consider where loops and intermediate variables can be replaced with comprehensions and existing functions**

Mutability pitfalls

- **Python is reference-based language, so objects are shared by default**
  - **Easily accessible data or state-modifying methods can give aliasing surprises**
  - **Note that true and full object immutability is not possible in Python**
- **Default arguments should be of immutable type, e.g., use *tuple* not *list***
  - **Changes persist between function calls**

Fewer explicit loops

- **A common feature of functional programming is fewer explicit loops**
  - ***Recursion*, but note that Python does not support tail recursion optimisation**
  - ***Comprehensions* — very declarative, very Pythonic**
  - ***Higher-order functions*, e.g., *map* applies a function over an iterable sequence**
  - ***Existing algorithmic functions*, e.g., *min*, *str.split*, *str.join***

Recursion

- Recursion may be a consequence of data structure traversal or algorithm
  - E.g., iterating over a tree structure
  - E.g., quicksort
- But it can be used as an alternative to looping in many simple situations
  - But beware of efficiency concerns and Python's limits (see *sys.getrecursionlimit*)

Reducing modifiable state

```
def factorial(n):
    result = 1
    while n > 0:
        result *= n
        n -= 1
    return result
```

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

```
def factorial(n):
    if n > 0:
        return n * factorial(n - 1)
    else:
        return 1
```

Two variables being modified explicitly

No variables modified

Expressions versus statements

There is a tendency in functional programming to favour expressions...

```
def factorial(n):
    if n > 0:
        return n * factorial(n - 1)
    else:
        return 1
```

```
def factorial(n):
    return n * factorial(n - 1) if n > 0 else 1
```

```
factorial = lambda n: n * factorial(n - 1) if n > 0 else 1
```

But using a lambda bound to a variable instead of a single-statement function is not considered Pythonic and means *factorial* lacks some metadata of a function, e.g., a good __*name*__.

Comprehensions

- **A comprehension is used to define a sequence of values declaratively**
  - **Sequence of values defined by intension as an expression, rather than procedurally in terms of loops and modifiable state**
  - **They have a *select...from...where* structure**
  - **Many common container-based looping patterns are captured in the form of container comprehensions**

Imperative versus declarative

```
def is_leap_year(year):
    return year % 4 == 0 and year % 100 != 0 or year % 400 == 0
```

```
leap_years = []
for year in range(2000, 2030):
    if is_leap_year(year):
        leap_years.append(year)

        [2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028]
```

Imperative list initialisation

```
leap_years = [year for year in range(2000, 2030) if is_leap_year(year)]

        [2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028]
```

List comprehension

Higher-order functions

- A higher-order function...
  - Takes one or more functions as arguments
  - Returns one or more function as its result
  - Takes and returns functions
- Higher-order functions often used to abstract common iteration operations
  - Hides the mechanics of repetition
  - Comprehensions are often an alternative to such higher-order functions

map

- ***map*** **applies a function over an iterable to produce a new iterable**
  - **Can sometimes be replaced with a comprehension or generator expression that has no predicate**
  - **Often shorter if no lambdas are involved**

```
map(len, 'The cat sat on the mat'.split())
```

```
(len(word) for word in 'The cat sat on the mat'.split())
```

filter

- ***filter*** **includes only values that satisfy a given predicate in its generated result**
  - **Can sometimes be replaced with a comprehension or generator expression that has a predicate**
  - **Often shorter if no lambdas are involved**

```
filter(lambda score: score > 50, scores)
```

```
(score for score in scores if score > 50)
```

reduce

- *functools.reduce* implements what is know as a *fold left* operation
  - Reduces a sequence of values to a single value, left to right, with the accumulated value on the left and the other on the right

```
def factorial(n):
    return reduce(lambda l, r: l*r, range(1, n+1), 1)
```

```
def factorial(n):
    return reduce(operator.mul, range(1, n+1), 1)
```

int.__mul__ would be a less general alternative

The *operator* module

operator exports named functions corresponding to operators

Comparison operations

```
eq(a, b)        a == b
ne(a, b)        a != b
lt(a, b)        a < b
le(a, b)        a <= b
gt(a, b)        a > b
ge(a, b)        a >= n
is_(a, b)       a is b
is_not(a, b)    a is not b
contains(a, b)  a in b
```

Binary arithmetic operations

```
add(a, b)       a + b
sub(a, b)       a - b
mul(a, b)       a * b
truediv(a, b)   a / b
floordiv(a, b)  a // b
mod(a, b)       a % b
pow(a, b)       a**b
```

Unary operations

```
pos(a)     +a
neg(a)     -a
invert(a)  ~a
```

Experiment

- **Use *reduce* to implement your own version of *map***
  - **Can be written as a single-line method**
  - **Return the result as a list**

```
def my_map(func, iterable):
    ...
```

```
my_map(lambda n: n*2, [3, 1, 4, 1, 5, 9])

                   [6, 2, 8, 2, 10, 18]
```

Currying

In mathematics and computer science, **currying** is the technique of translating the evaluation of a function that takes multiple arguments (or a tuple of arguments) into evaluating a sequence of functions, each with a single argument (partial application). It was introduced by Moses Schönfinkel and later developed by Haskell Curry.

http://en.wikipedia.org/wiki/Currying

Nested functions & lambdas

- **Nested functions and lambdas bind to their surrounding scope**
  - **I.e., they are closures**

```
def curry(function, first):
    def curried(second):
        return function(first, second)
    return curried
```

```
def curry(function, first):
    return lambda second: function(first, second)
```

```
hello = curry(print, 'Hello')
hello('World')
```

Function wrapping

Force non-positional use of
subsequent parameters

```
def timed_function(function, *, report=print):
    from time import time
    def wrapper(*args):
        start = time()
        try:
            function(*args)
        finally:
            report(time() - start)
    return wrapper
```

```
wrapped = timed_function(long_running)
wrapped(arg_for_long_running)
```

partial application

- **Values can be bound to a function's parameters using *functools.partial***
  - **Bound positional and keyword arguments are supplied on calling the resulting callable, other arguments are appended**

```
quoted = partial(print, '>')

quoted()
                                        print('>')
quoted('Hello, World!')
                      print('>', 'Hello, World!')
quoted('Hello, World!', end=' <\n')
          print('>', 'Hello, World!', end=' <\n')
```

Built-in algorithmic functions

```
min(iterable)
min(iterable, default=value)
min(iterable, key=function)
max(iterable)
max(iterable, default=value)
max(iterable, key=function)
sum(iterable)
sum(iterable, start)
any(iterable)
all(iterable)
sorted(iterable)
sorted(iterable, key=function)
sorted(iterable, reverse=True)
zip(iterable, ...)
...
```

Default used if *iterable* is empty

The function to transform the values before determining the lowest one

One or more iterables can be zipped together as tuples, i.e., effectively converting rows to columns, but zipping two iterables is most common

Functions from *itertools*

```
chain(iterable, ...)
compress(iterable, selection)
dropwhile(predicate, iterable)
takewhile(predicate, iterable)
count()
count(start)
count(start, step)
islice(iterable, stop)
islice(iterable, start, stop)
islice(iterable, start, stop, step)
cycle(iterable)
repeat(value)
repeat(value, times)
zip_longest(iterable, ...)
zip_longest(iterable, ..., fillvalue=fill)
...
```

Concurrency

Coroutines, threading, processes & futures

Facts at a glance

- Generators can be used as coroutines
- Threading is based on native threads
- Python's threading is subject to GIL constraints on many implementations
- Multiprocessing has API-compatible support plus IPC mechanisms
- Executors and futures offer a simple task-based concurrency model

Concurrency

- **Concurrency can...**
  - **Be implicit in the problem domain**
  - **Be a by-product of the implementation**
  - **Improve performance**
  - **Worsen performance**
  - **Simplify and decouple code**
  - **Complicate and couple code**
- **It's all a question of implementation mechanisms and choices!**

Coroutines

Coroutines are computer program components that generalize subroutines for nonpreemptive multitasking, by allowing multiple entry points for suspending and resuming execution at certain locations. Coroutines are well-suited for implementing more familiar program components such as cooperative tasks, exceptions, event loop, iterators, infinite lists and pipes.

https://en.wikipedia.org/wiki/Coroutine

Generators as coroutines

- ***yield* and *yield from* cause suspension of the current function execution**
  - **On resumption, e.g., via *next*, execution continues where it left off, so that local variable state is retained**
- **The alternative to using coroutines is often more complex**
  - **E.g., a state machine to remember where a task was last time it was executed**

Reducing complexity

Symmetry is a complexity-reducing concept (co-routines include subroutines); seek it everywhere.

Alan Perlis

"Epigrams in Programming"

Coroutine communication

- **Can communicate with a generator**
  - ***send*** **passes a value to it**
  - ***throw*** **raises an exception within it**
  - ***close*** **terminates it**
  - ***send*** **and** ***throw*** **return next yielded value**
- **Although often used as statements,** ***yield*** **and** ***yield from*** **are expressions**
  - **When invoked by** ***next*****, they return** ***None***
  - **When** ***send*** **is used, value sent is returned**

Coroutine action

```
def evaluate():
    input = ''
    for n in count():
        input = (yield str(n) + ' ' + evaluated(input))
```

```
def repl():
    evaluator = evaluate()
    next(evaluator)
    try:
        for line in iter(lambda: input('> '), 'exit'):
            print(evaluator.send(line))
    finally:
        evaluator.close()
```

Multithreading

Multithreading is just one damn thing before, after, or simultaneous with another.

Andrei Alexandrescu

threading

- The *threading* module offers constructs for creating threads and locking
  - Threads are built on native OS threads
  - Locking primitives are based on binary semaphores, which are in turn based on OS mutexes
  - Other synchronisation primitives, such as events, and thread-based constructs, such as timers and thread-local data, are also defined

Thread definition

Inheritance-based approach

```
task = Task()
task.start()
...
task.join()
```

```
class Task(Thread):
    def run():
        ...
```

Composition-based approach

```
task = Thread(target=run)
task.start()
...
task.join()
```

```
def run():
    ...
```

Global Interpreter Lock (GIL)

- **In CPython, the GIL introduces a concurrency bottleneck**
  - **It is a mutex that restricts system threads from concurrently executing Python code**
  - **The thread-unsafe reference-counted memory management model is the primary reason for the GIL's existence**
  - **A feature of CPython but not necessarily other implementations, e.g., Jython relies on JVM GC and IronPython relies on .NET**

Synchronisation

- **Python offers a number of different types of synchronisation primitive**
  - **E.g., *Lock*, *RLock*, *Condition*, *Semaphore*, *BoundedSemaphore*, *Event*, *Barrier***
- **All the locking-related primitives are context managers**
  - **Use *with* rather than *acquire* and *release* unless for non-blocked acquisition or a timeout is required**

Sharing & synchronisation

> Shared memory is like a canvas where threads collaborate in painting images, except that they stand on the opposite sides of the canvas and use guns rather than brushes. The only way they can avoid killing each other is if they shout "duck!" before opening fire.
>
> *Bartosz Milewski*

http://bartoszmilewski.com/2013/12/10/functional-data-structures-and-concurrency-in-c/

Multiprocessing

- An alternative to threads is to use OS processes to express concurrency
  - A standard workaround for GIL issues
- Processes naturally give better isolation and long-term concurrency
  - But may not be effective for fine-grained concurrency
  - Any values exchanged between processes must be picklable

multiprocessing

- The *multiprocessing* module has similar API to *threading*
  - Including locking primitives
- Also includes IPC mechanisms
  - Different types of inter-process queues and *Pipe*, which results in a pair of *Connection* objects
  - *Value* and *Array* can be used to create and access objects in shared memory

Process action

```
def evaluate(client):
    try:
        for n in count(1):
            input = client.recv()
            client.send(str(n) + ' ' + evaluated(input))
    except EOFError:
        pass
```

```
def repl():
    here, there = Pipe()
    evaluator = Process(target=evaluate, args=(there,))
    evaluator.start()
    for line in iter(lambda: input('> '), 'exit'):
        here.send(line)
        print(here.recv())
    here.close()
    evaluator.join()
```

Asynchronous functions

- **Asynchronous function approaches simplify use of threads and processes**
  - **Instead of blocking on a function call, a function is executed concurrently**
  - **A *future* — a virtual proxy also known as an *IOU* or a *deferred* — is a queryable place holder for the function's result**
- ***asyncio* and *concurrent.futures* are based on this model**

Executors & futures

- **Executors simplify use of concurrency execution resources**
  - **One or more tasks can be submitted to an executor for execution**
  - **Execution mechanics and management are hidden from the submitter**
- **The caller uses a future to synchronise with and collect execution results**
  - **Reduces need for other synchronisation**

concurrent.futures

- **An executor is a pools of workers**
  - **Either threads, in the case of *ThreadPoolExecutor*, or processes, in the case of *ProcessPoolExecutor***
  - **Executors are context managers**
- **To submit work...**
  - **Call *submit* to submit an individual item**
  - **Call *map* to submit multiple items — a returned generator plays the future role**

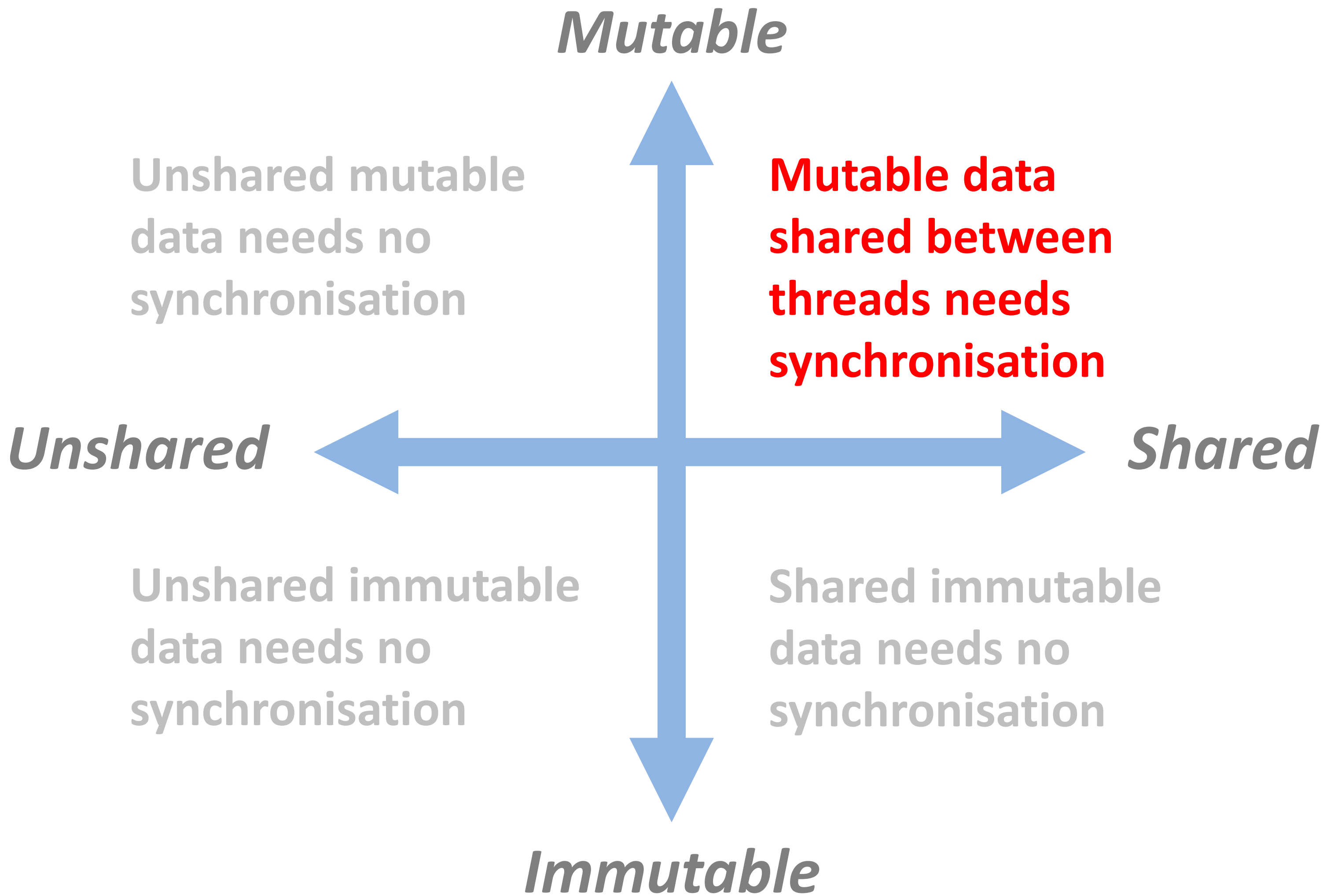
In the future...

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(task, data)
    print(future.result())
```

```
with ThreadPoolExecutor(max_workers=4) as executor:
    futures = [
        executor.submit(task, data) for data in work
    ]
    for future in futures:
        print(future.result())
```

```
with ThreadPoolExecutor(max_workers=4) as executor:
    for result in executor.map(task, work):
        print(result)
```

Synchronisation quadrant



Modules & Packages

The mechanics of organising source code

Facts at a glance

- A package hierarchically organises modules and other packages
- Where a module corresponds to a file, a regular package corresponds to a directory
- Modules and packages define global namespaces
- Extension modules allow extension of Python itself

Modules

- A module in Python corresponds to a file with a *.py* extension
  - *import* is used to access features in another module
- A module's *__name__* corresponds to its file name without the extension
  - When run as a script (e.g., using the *–m* option), the root module has '*__main__*' as its *__name__*

Extensions & embedding

- **The Python interpreter can be extended using extension modules**
  - **Python has a C API that allows programmatic access in C (or C++) and, therefore, any language callable from C**
- **Python can also be embedded in an application as a scripting language**
  - **E.g., allow an application's features to be scripted in Python**

Packages

- **Packages are a way of structuring the module namespace**
  - **A submodule *bar* within a package *foo* represents the module *foo.bar***
- **A regular package corresponds to a directory of modules (and packages)**
  - **A regular package directory must have an *__init__.py* file (even if it's empty)**
  - **The package name is the directory name**

Namespaces

- A namespace is a mapping from names to objects, e.g., variables
  - A namespace is implemented as a *dict*
  - Global, local, built-in and nested
- Each module gets its own global namespace, as does each package
  - *import* pulls names into a namespace
  - Within a scope, names in a namespace can be accessed without qualification

Packages & *import*

- **A submodule can be imported with respect to its package**
  - **E.g., *import package.submodule***
  - **Relative naming can be used to navigate within a package**
  - **Main modules must use absolute imports**
  - **Submodules seen by wildcard import can be specified by assigning a list of module names (as strings) to *__all__* in *__init__.py***

Packages, modules & imports

Top-level regular package

Subpackage

```
root/
    __init__.py
    first/
        __init__.py
        foo.py
        bar.py
    second/
        __init__.py
        foo.py
        bar.py
```

```
# empty, but necessary
```

```
__all__ = ['foo']
```

```
from . import bar
from .. import first
from ..first import foo
```

Experiment

- **Write a simple script that lists the package and module hierarchy**
  - **Use current directory if no path supplied**
  - **Use *os.listdir*, *os.walk* or *pathlib* for traversal**
  - **No need to open and load modules, just detect packages by matching *__init__.py* and modules by matching **.py***
  - **List the results using indentation and/or in some XML-like form**

Outroduction

All good things…

The end?

- **The foundation is all there**
  - ***Python Foundation*** **course and this course**
- **But a foundation means little unless you build on it**
  - **There is more to the language...**
  - **There is more to the library...**
  - **There are more libraries...**
- **Enjoy!**

Labs & Homework

To do...

Guidance

- **Labs should be undertaken in pairs using Cyber-Dojo**
  - **Swap between driving and navigator roles within the same pair**
- **Homework is carried out individually in your own environment**
  - **Do not spend too much time on this — an hour or so — and try not to overengineer the solution!**

Lab: Day 1

Money

Write a class (and unit tests) for a class that represents a monetary value:

- It should hold an amount that can be queried as units and hundredths (e.g., pounds and pence, euros and cents, dollars and cents).
- It should be possible to compare money values for equality and for ordering using relational operators.
- It should support non-lossy addition and subtraction, i.e., don't use *float* to hold the amount.
- Negative money is allowed.
- Money values should not offer modifier methods.
- Don't worry about different currencies.

Lab: Day 1

Option 1: Incompatible currencies

Refine your money class and tests so that instances have a currency code associated with them, e.g., '*NOK*' for Norwegian krone, '*EUR*' for Euro:

- Only money values with the same currency code can be added or subtracted from one another.
- Money values of different currencies can be compared for equality, but always return *False*.
- Money values of different currencies cannot be ordered.

Lab: Day 1

Option 2: Compatible currencies

Allow money values of different currencies to be added together, but so that they result in composite type, i.e., a money bag:

- They support addition, but do not resolve down to a single currency, e.g., 100 NOK + 10 EUR + 200 NOK + 5 EUR results in a money bag of 300 NOK + 15 EUR.
- Money bags can be compared for equality, but cannot be ordered.

Homework

A simple testing framework

Write a small testing framework that executes all the tests defined within a given class:

- Provide a *run* function that takes a class of test cases as its argument.
- For each test method in the given class, create an instance of the class and execute the test method against it, reporting back the result to the console, and then reporting a summary of the result at the end of the test run.
- A test method is one whose name begins with *test*.
- Use the built-in *assert* statement for assertions within test methods.

Homework

An example of use

For the following function, which determines whether a year is a leap year or not:

```
def is_leap_year(year): ...
```

The test framework should support the following test code:

```
class LeapYearTests:
    def test_years_not_divisible_by_4_are_not_leap_years(self):
        assert not is_leap_year(2015)
    def test_years_divisible_by_4_but_not_by_100_are_leap_years(self):
        assert is_leap_year(2016)
    def test_years_divisible_by_100_but_not_by_400_are_not_leap_years(self):
        assert not is_leap_year(1900)
    def test_years_divisible_by_400_are_leap_years(self):
        assert is_leap_year(2000)
```

Executed as follows:

```
run(LeapYearTests)
```

Homework

Option 1: Using decorators to mark test methods

If you have enough time, consider changing the test runner so that instead of relying on a method prefix, a decorator is used to indicate that a method is a test method:

```
class LeapYearTests:

    @test
    def years_not_divisible_by_4_are_not_leap_years(self):
        assert not is_leap_year(2015)

    @test
    def years_divisible_by_4_but_not_by_100_are_leap_years(self):
        assert is_leap_year(2016)

    ...
```

This is syntactically similar to the annotations used in JUnit, but in Python it will work by a different effect. Define a *test* decorator that adds an attribute to the test method; it is this attribute that the *run* function will use to select the test methods for execution.

Homework

Option 2: Using decorators to pass data to a test method

If you have enough time, consider refining your decorator design so that data can optionally be passed into the test case:

```
class LeapYearTests:

    @test
    @data(2015, 1999, 1)
    def years_not_divisible_by_4_are_not_leap_years(self, year):
        assert not is_leap_year(year)

    @test
    @data(2016)
    @data(1984)
    @data(4)
    def years_divisible_by_4_but_not_by_100_are_leap_years(self, year):
        assert is_leap_year(year)

    @test
    def years_divisible_by_100_but_not_by_400_are_not_leap_years(self):
        assert not is_leap_year(1900)

    @test
    @data(*range(400, 2401, 400))
    def years_divisible_by_400_are_leap_years(self, year):
        assert is_leap_year(year)
```

Homework

Further variations

If you still have enough time, or would prefer something different to the previous two options, here are some more ideas to play with:

- Use a keyword argument to specify the output for test results, with the *sys.stdout* as default.
- Use an Enumeration Method callback from the test runner function to communicate test execution results (default to *print*).
- Use multiple callbacks to indicate different events in a test run's lifecycle, i.e., the Lifecycle Callback pattern. Either pass in callables via keyword arguments or use the Visitor pattern.
- Instead of instantiating a test object, consider using module-level functions or static methods as the basis for test cases.
- Define a decorator that checks that an exception has been thrown from a test case, failing if no or the wrong type is thrown.
- Execute test cases concurrently instead of sequentially.

Lab: Day 2

Recently used list

Write a class (and unit tests) for a container that represents a *recently used list*, i.e., a sequence ordered by reverse insertion order and restricted by uniqueness:

- A recently used list is initially empty.
- Strings can be added to the list, but non-string values (including *None*) are disallowed.
- Items can be indexed, which counts from zero.
- Additions are retained in stack ordering (i.e., LIFO), so the most recently added item is first, the least recently added item is last, and so on.
- Items in the list are unique, so duplicate insertions are moved to the head rather than added, i.e., the length will not increase.

Lab: Day 2

Option 1: Support for non-string items

Generalise the implementation so that addition of non-string values is supported. This is not, however, simply a case of removing the type restriction in *add*:

- Because the list is set-like — i.e., items are unique — items also play the role of keys, which means they should be immutable.
- There is no guarantee of a type's immutability, but a common convention is that hashable types are immutable and, therefore, non-hashable types are not.
- It should therefore be possible to add tuples and integers to a recently used list, but not lists and sets.
- But note that addition of *None* is still disallowed.

Lab: Day 2

Option 2: Additional container operations

Add methods to improve support for a more typical container protocol:

- A *clear* method to empty the list.
- A *pop* method to remove the oldest item in the list.
- A *remove* method to remove an item by value.
- Support common lookup operations such as *in*, *not in* and *index*.
- Support comparison for equality and inequality.
- Support the iterator protocol.
- Allow items to be deleted by index.