Further Hands-On Python Programming

1 Introduction

This course has three main organizational components

- Developing and enriching a single code base, throughout the duration of the course.
- Gaining a profound understanding of Python through study of concepts which arise in the process of developing the above program.
- For those who need to be stretched a bit more, a set of more challenging exercises is provided.

Most of the value in this course comes from attending the sessions, doing the exercises and having **your own** questions (as well as those of others) answered.

What actually happens in any edition of this course mostly depends on the people who attend, and the questions they ask. These notes cannot possibly be an accurate representation of any particular edition of the course.

2 Suggested path

2.1 Version control

Create a Mercurial repository for the code we will develop in this course.

2.2 Getting started

Make sure that you can run `tkinterversion.py` and `pygletversion.py`. These programs are far from a shining example of good design and programming practice. We will identify many of their flaws and iron them out, during the course.

The former depends on Python's standard module `Tkinter`, the latter depends on `pyglet`, a third-party library. These should be pre-installed on the machines provided, but you may need to install them yourself if you are working on your machine.

Even though `Tkinter` is a standard Python module, some Linux distributions provide Python packages in which `Tkinter` will not work unless you explicitly install `Tk`.

2.3 Remove magic numbers

> Remove any magic numbers from the code. Commit the changes to your Mercurial repository.

2.4 Ask questions about the code

Read through the code and ask questions about anything that you don't understand, any syntax or constructs with which you are not familiar.

2.5 Testing

Test-driven deveopment (TDD) is the practice of

- Writing an automated test for any functionality in your code, just **before** you implement the functionality
- Not writing any production code unless there is a failing test requiring you to do so.

This might seem like it slows progress, because it doubles the amount of work that needs to be done, but, in practice, many people find that this extra effort pays back handsomely, because

- You get an almost complete test suite for your projects.
- Attempting to pass the currently failing test helps you concentrate on the task at hand, thereby avoiding feature creep and overgeneralization.
- Writing tests and matching implementation code makes you write code which is testable in isolation of the rest of your program, which is usually more loosely coupled, less monolithic, more flexible: in short *better*.
- Refactoring becomes much cheaper: you see immediately that everything that was expected to work still works, or you see immediately where things have gone wrong. Reducing relucance to refactor, helps to keep down the technical debt of our project.
- The tests document expected usage and behaviour.

When practising TDD one should aim to do the simplest thing that makes the currently failing test pass. It's fine to commit hienous coding crimes in order to pass the test (with the exception of deleting or breaking the test itself), as long as

- It doesn't cause your code to fail any other tests
- You immediately follow this up by writing another test which makes you atone for the crime.

The point is to keep the scope of your current problem manageably small: don't try to do too much at once.

As a simplistic example of this principle, consider the test

```
def test_addfunction_should_give_correct_answer_for_1_and_2():
    assert addfunction(1,2) == 3
```

It's easy to cheat on this test:

```
def addfunction(a,b):
    return 3
```

but the implementation is clearly unsatisfactory. The solution is to strengthen the test suite. One way to do this would be to add a second test:

```
def test_addfunction_should_give_correct_answer_for_4_and_9():
    assert addfunction(4,9) == 13
```

Another way would be to strengthen the original test:

```
def test_addfunction_should_give_correct_answer_for_1_and_2():
    for l,r in ((1,2),
                (4,9),
                (-10,100)):
        assert addfunction(l,r) == l+r
```

Both of these have the effect of making it impossible to cheat in the way we did at the beginning.

In such a trivial case, it might be diffucult to appreciate the point of this. Why didn't we just write the stronger test and the correct implementation immediately?

In this particular case, it would be perfectly reasonable to do so. But, if the feature you are implementing is more complicated, it might help to break it down into smaller steps. In our example, the first test established the interface. Once we were happy with the interface, the further tests forced us to work on the implementation.

2.6 `py.test`

In this course we will use py.test as our testing framework.

2.7 Minimalistic demonstration of TDD

Let's apply TDD to everyone's favourite toy problem: *Fibonacci*.

We kick off with a simple test for the base case

```
def test_fib_of_0_should_be_0():
    assert fib(0) == 0
```

It's easy to pass this test by cheating

```
def fib(n):
    return 0
```

so we strengthen our test suite

```
def test_fib_of_1_should_be_1():
    assert fib(1) == 1
```

Now we have to work a bit harder, but we can still cheat:

```
def fib(n):
    if n == 0:
        return 0
    return 1
```

Before we strengthen the test suite, we notice the repetitive nature of the test, so we refactor the test suite, by replacing our two tests with a single data-driven one:

```
def test_fib_should_give_correct_outputs():
    for input_, expected_output in ((0,0),
                                    (1,1)):
        assert fib(input_) == expected_output
```

We rerun the test suite, to confirm that nothing signifcant has changed.

After refactoring, strengthening the test suite has become much cheaper:

```
def test_fib_should_give_correct_outputs():
    for input_, expected_output in ((0,0),
                                    (1,1),
                                    (2,1)):
        assert fib(input_) == expected_output
```

Given that we are using *pytest*, we can use its `parametrize` decorator:

```
from pytest import mark

@mark.parametrize('input_ expected_output'.split(),
                  ((0,0),
                   (1,1),
                   (2,1)))
def test_fib_should_give_correct_outputs(input_, expected_output):
    assert fib(input_) == expected_output
```

When we run the strengthened test suite, everything passes. This is suspicious, but if we look at it carefully, we can see that we just happened to be lucky, our cheat happens to hit on the correct answer for the extra test point. Adding another point will force us to do better.

```
from pytest import mark

@mark.parametrize('input expected_output'.split(),
                  ((0,0),
                   (1,1),
                   (2,1),
                   (3,2)))
def test_fib_should_give_correct_outputs(input_, expected_output):
    assert fib(input_) == expected_output
```

Oh good, it fails: Time to work on the implementation.

```
def fib(n):
    if n == 0:
        return 0
    if n < 3:
        return 1
    return 2
```

At this point we should notice that we will get stuck in an infinite loop of adding further test points, and responding with special cases, unless we stand back and inject a little bit of intelligence into the process. The intelligent part is to notice that `0` and `1` are base cases, while the rest can be built on top of this foundation:

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

In any non-trivial project, the day comes when we decide that some part of it needs to reimplemented. In this case, maybe someone called `fib(100)`, and we concluded that our implementation is not up to the task. We want to change the implementation, but we do not want to change the semantics of the code: time to refactor.

> Refactor `fib` by using an iterative algorithm. Make sure that your new implementation passes exactly the same tests.

Notice how the existing test suite pointed out errors in our refactored implementation, and/or gave us confidence that our implementation is at least as good as the original one.

2.7.1 TODO pytest-quickcheck

2.8 Combine the programs

`tkinterversion.py` and `pygletversion.py`

2.9 Particle

The two programs simulate the motion of a circular particle bouncing around a rectangular 2-dimensional universe. However, the way the program is written, it requires detailed inspection of the code to realize that this concept appears in both programs and that it is identical in both.

> Create a `Particle` class, to turn this abstract concept into a concrete one.

2.9.1 Public data are OK in Python

> Do not write *trivial* getters and setters in Python.

This is discussed in greater detail here.

2.9.2 `Particle` tests

> Make sure that your `Particle` class passes these tests.

2.10 Abstraction: display

> Write `TkinterDisplay` and `PygletDisplay` classes, with a common interface. Join the two programs into one, providing a command line switch to choose between the backends.

Note that the `update` method can be the same for both of them. Make them subclasses of a `Display` class which provides the `update` method.

2.10.1 TODO Interface discussions

Redraw the particle in the tk version, rather than moving it: `canvas.delete(tk.ALL)`

- `Particle.draw(display)`
- `Display.bounding_box()`
- `Display.go()`
- `Display.draw_circle()`, `Display.draw_square()`, etc.

How to deal with `@window.event`.

In the above, Display is completely decoupled from Particle (good, it's completely general), but Particle is aware of the Display interface (not so good).

2.11 Abstraction: vector

Our particles contain the abstract concept of *vector*, representing both the position and the velocity of the particles.

> Create a `Vector2D` class which passes these tests.

2.11.1 Operator overloading

Python operators can be overloaded by implementing special/magic methods.

2.12 Refactor `Particle`

> Refactor `Particle`, by using `Vector2D` in its internal representation.

2.13 TODO Extend particle interface

Did `Particle_test.py` help you in the process of refactoring `Particle`?

> Exend `Particle's` interface, to pass these tests.

2.14 Challenge exercise

If you are finding the pace of the course too slow, you may like to try to pass these tests while you are waiting for others to catch up..

2.15 Testing for exceptions

> Imagine you are writing a py.test test suite for Python's standard math module. The module contains a function called `sqrt`, which raises `ValueError` when it is given a negative number. Write a test which checks that `ValueError` is indeed raised in such situations. Don't do anything sophisticated yet: just trite a `try`-block in your test.

2.15.1 `py.test.raises`

Read the documentation on `py.test.raises`.

> Rewrite the test from the last exercise using `raises`. Make sure you understand the three different styles it supports
>
> 1. Passing a to-be-evaluted string (Yuck! Unpythonic. Avoid this style, avoid designing interfaces that use this style.)
> 2. Passing callable and arguments as separate arguments to `raises`
> 3. Using `raises` as a context manager.

2.16 Context managers

We briefly discussed context managers near the beginning of the course. They were primarily motivated by the desire to abstract try-finally constructs, but they lend themselves to use in a much broader set of circumstances.

A context manager is an object which controls and abstracts the behaviour of a program as `with`-blocks are entered and exited. We have already met two (simple) examples

1. `open` opens a file on entry, and closes it on exit
2. `py.test.raises` specifies desired exception types on the way in, and checks whether the code inside the block raises the specified exception on the way out.

A context manager is an object which implements two special methods:

1. `__enter__`
2. `__exit__`

In

```
with my_context_manager(...) as name:
    ....
```

- `__enter__` is called when the `with`-block is entered.
- Whatever `my_context_manager` returns, is bound to `name`.
- `__exit__` is called when the `with`-block is exited.

> Implement something equivalent to `py.test.raises` by writing a class (i.e. ignore the existence of `contextlib.contextmanager`, which we will explore shortly).

```
class myraises(object):

    def __init__(self, Exception_spec):
        self._Exception_spec = Exception_spec

    def __enter__(self):
        pass

    def __exit__(self, ExType, ExInst, tb):
        if ExType is self._Exception_spec or ExType in self._Exception_spec:
            assert True
        else:
            assert False
        return True
```

Notice that for any `with`-embedded call to a context manager, the following three things always occur exactly once, in exactly the same order

1. `__enter__` runs exactly once
2. The context manager returns a Python object (maybe `None`) which will be bound to the target name, if one is specified.
3. `__exit__` runs exactly once.

2.16.1 Generator functions revisited

Given

```
def cm(a):
    print "entering"
    yield a
    print "exiting"
```

execute

```
c = cm(input)
try:
   n = c.next()
   print n*n
finally:
   c.next()
```

once with `input` bound to `10` and once with `input` bound to `[]`.

TODO: StopIterations appear. Think about how to make them less distracting.

Notice that the following pattern is always repeated

1. The code which precedes `yield` in `cm` runs exactly once.
2. The value yielded in `cm` is bound to `n`.
3. The code which follows `yield` in `cm` runs exactly once.

This is, essentially, the same pattern as exhibited by context managers.

2.16.2 `contextlib.contextmanager`

`contextlib.contextmanager` is a (standard) utility which simplifies the writing of context managers. You express the context manager as a generator function with a single `yield`.

- Code which precedes the `yield` is executed on entry.
- Whatever is yielded is bound to the target.
- Code which follows `yield` is executed on exit.
- Exception handling is done naturally by wrapping the `yield` inside a try block. (Compare this to the ad-hoc way of dealing with exceptions in the implementation of plain context managers.)

From the perspective of the generator function, the `yield` expands into the body of the `with`-block.

The last example can therefore be rewritten as

```
import contextlib

@contextlib.contextmanager
def cm(a):
    print "entering"
    yield a
    print "exiting"

with cm(10) as n:
    print n*n

with cm([]) as n:
    print n*n
```

Notice how easy it becomes to write your own context managers, with `contextlib.contextmanager`.

2.16.3 Simple context manager exercise

Everyone should attempt this exercise right now.

> Write a context manager, `timer`, which passes the following test.

```
# A context manager for timing code. Note: do not use this for serious
# timings: the overhead will be too large when the timed duration is
# very short. For serious timings use the timeit module.

from pytest import mark
from time import sleep

@mark.parametrize('seconds_to_sleep', range(1,4))
def test_timer_should_report_approximately_correct_times(seconds_to_sleep):
    with timer() as t:
        sleep(seconds_to_sleep)
    sleep(1)
    assert abs(t.time - seconds_to_sleep) < 0.01
```

Your solution should be built around this skeleton:

Mouse over to see answer.

```
from contextlib import contextmanager
from time import time

class Timer_result(object):
    pass

@contextmanager
def timer():
```

```
    start = time()
    result = Timer_result()
    yield result
    result.time = time() - start
```

Note: This is just an exercise in getting used to the `contextlib.contextmanager` tool: I do not recommend that you take this approach to timing performance of your code; much better tools exist for that purpose in the standard library.

2.16.4 Challenge exercise

This is a more involved exercise, aimed at those who are finding the course a bit slow. Only attempt this if you find yourself waiting around for the rest of the course to catch up with you, or as an extra exercise outside of the course sessions.

Context manager exercise test suite.

2.17 Generalize vector to arbitrary dimension

The vector class we wrote earlier is specific to two dimensions. We could add a 3D version, which would be very similar, as would a 4D version. Rather than writing a bunch of similar classes, let's write a class factory.

This class factory should generate the desired class from a simple specification. The specification should consist of an iterable containing the names of the dimensions. The number of names determines the dimension of the vector.

For example

```
Vector3D = Vector('xyz')
v = Vector3D(1.0, 2.0, 3.0)
assert v.x == 1.0
assert v[0] == 1.0
```

In the first line, we use the class factory (`Vector`) to make a 3-dimensional vector, whose components will be accessible via the attributes `x`, `y`, and `z`, as well as via the itemgetter syntax `[n]`, with indices `0`, `1` and `2`.

You may like to use this test suite to guide you.

Note that the test suite assumes that `Vector2D = Vector('xy')` and `Vector3D = Vector('xyz')` have been provided as convenience bindings out of the box.

2.17.1 Generating classes at runtime

1. Creating a class in a function

   Here is a simple template showing how to generate classes at runtime.

```
def class_factory(class_name, bindings):

    class The_Class(object):
        pass

    for name, value in bindings.items():
        setattr(The_Class, name, value)
```

```
    The_Class.__name__ = class_name

    return The_Class
```

2. Calling `type`

   When Python executes a class block, it collects three items of information in the process

   1. The name of the class.
   2. The direct superclasses of the class.
   3. The bindings established within the block.

   It then passes this information to `type` which creates a new type with the given specification. We can instruct `type` to create new types directly, ourselves. So, the earlier class generation example can be expressed more concisely like this.

```
def class_factory(class_name, bindings):
    return type(class_name, (object,), bindings)
```

2.18 Closures in loop surprise

When installing methods or properties in the generated Vector class, we may run into a problem where entities which should refer to different components of the vector, all refer to the last component. Here's a demonstration of the essence of this problem

```
class Vector(object):

    def __init__(self, x,y,z):
        self._data = [x,y,z]

for i, name in enumerate('xyz'):
    setattr(Vector, name,
            property(lambda self:self._data[i]))

v = Vector(1,2,3)

# x,y,z should be 1,2,3 but are, in fact, 3,3,3
assert v.x == v.y == v.z == 3
```

The problem arises because all the closures created inside the loop are closures of a single 'instance' of the binding of `i`. The first closure is created while `i` is bound to `1`; the second closure is created while `i` is bound to `2`, but it is the same instantiation of the `i` variable. Therefore the two closures enclose the same binding, and the first closure sees that the binding has changed.

In order to get the closures to see different bindings, you must enter a new scope which creates a new (separate) binding of `i`, for each closure you create. In Python, functions are the only mechanism available to us for creating local nested scopes, so the solution is to warp the closure creation operation in a function whose purpose is to create the fresh binding of `i`.

In brief

```
def fubar(n):
    return [          (lambda : i)     for i in xrange(n)]

def foo(n):
    return [(lambda x:(lambda : x))(i) for i in xrange(n)]

[fn() for fn in fubar(5)] # Shared bindings
[fn() for fn in foo  (5)] # Separate bindings
```

Notice how, in `foo`, the outer `lambda` creates a fresh binding of `x`, each time around the loop. The resulting closures (the inner `lambda`) enclose separate bindings of `x`.

In `fubar`, all of the closures (the only `lambda` appearing inside `fubar`) enclose `i`: a binding which is created only once per invocation of `foo`.

Writing the same idea out in a more verbose way:

```
def fubar(n):
    r = []
    for i in xrange(n):
        def closure():
            return i
        r.append(closure)
    return r

def foo(n):
    r = []
    for i in xrange(n):
        closure = make_closure_with_separate_binding_of_i(i)
        r.append(closure)
    return r

def make_closure_with_separate_binding_of_i(i):
    def closure():
        return i
    return closure

[fn() for fn in fubar(5)] # Shared bindings
[fn() for fn in foo  (5)] # Separate bindings
```

2.19 TODO assert_evolution

Show the demo.

Discuss writing tests for the different particle motions: constant speed, monotonically increasing speed, etc.

> Pass the these tests.

2.20 TODO multimethods

Languages such as Java, C++, C#, Smalltalk, Ruby and Python have *single dynamic dispatch*.

Dispatch is the process of finding the function, out of a collection of similarly named functions, which should run in a given situation. For example, in the context of

```
class Foo(object):

    def doit(self):
        print "Foo"

class Bar(object):

    def doit(self):
        print "Bar"
```

the following call to `doit`

```
foo_or_bar.doit()
```

might invoke on of two different `doit` functions.

Dynamic dispatch (also known as late binding) is carried out at run time, as opposed to *static* dispatch, which takes place at compile time. An example of static dispatch is function overload resolution in C++.

In all the languages mentioned above, dynamically dispatching functions have a special syntax which treats one of the arguments of the function differently from the others. In Python (and Java, C++, C# and even Ruby)

```
arg1.method(arg2, arg3, arg4)
```

is essentially a function call with four arguments: conceptually it is equivalent to

```
method(arg1, arg2, arg3, arg4)
```

Python's `self` makes this equivalence rather obvious, while the other languages go out of their way to make it look more magical than it really is.

The first argument is given a special syntax, because it is the only one whose type will be taken into consideration during dynamic dispatch. In any of these languages, dynamic dispatch only takes the dynamic type of a single object into consideration at any one time.

Languages such as Julia, Clojure, Common Lisp and Dylan, provide multimethods/generic functions as a standard language feature.

```
@multimethod(Rock, Rock)
def winner(left, right):
    return None

@multimethod(Rock, Paper)
def winner(left, right):
    return right

@multimethod(Paper, Rock)
```

```
def winner(left, right):
    return left

@multimethod(Rock, Scissors)
def winner(left, right):
    return left

# and so on ...
```

2.20.1 TODO Python 3 implementation

In Python 3 introduces function annotations, which would allow us to use a different syntax:

```
@multimethod
def winner(left:Rock, right:Rock):
    return None

@multimethod
def winner(left:Rock, right:Paper):
    return right

@multimethod
def winner(left:Paper, right:Rock):
    return left

@multimethod
def winner(left:Rock, right:Scissors):
    return left

# and so on ...
```

2.21 TODO Generalize bounce components

Point out that `move` is now dimension independent, but `boundary_bounce` is not.

Make boundary_bounce dimension independent.

Use to demonstrate use of functions as data

2.22 TODO Generators: send and throw

Reversible stepper exercise.

Mention that we've already seen throw in action in contextmanager

Maybe a mutating iterator exercise? A utility which allows you to modify a non-sequence container in a for-loop.

2.23 Default arg trap

Default arguments are evaluated once: at the time the function is defined. This may lead to surprising behaviour when the default value is mutable

```
def fubar(a,x=[]):
```

```
    x.append(a)
    return x

fubar(1)  # [1]
fubar(2)  # [1, 2]
fubar(3)  # [1, 2, 3]
```

The idiomatic way to avoid this, is

```
def foo(a,x=None):
    if x is None:
        x = []
    x.append(a)
    return x

foo(1)  # [1]
foo(2)  # [1]
foo(3)  # [1]
```

2.24 Visitor

Visitor for adding operations

```
class TypeA(object):

    def do(self, operation):
        operation.onA(self)

class TypeB(object):

    def do(self, operation):
        operation.onB(self)

a,b = TypeA(), TypeB()

# None of the above code needs to be changed if you want to add new
# operations

class OperationA(object):

    def onA(self, thing):
        print "Operation A on type A"

    def onB(self, thing):
        print "Operation A on type B"

opA = OperationA()
# You can perform opA on type a and type b
a.do(opA)
b.do(opA)

# You can add new operations without changing the code of the
# types. Note how this contrasts with normal OO style where you can
# add new types without touching any of the old code, but to add new
# operations you must modify all the types involved

class OperationB(object):

    def onA(self, thing):
        print "Operation B on type A"
```

```
    def onB(self, thing):
        print "Operation B on type B"

opB = OperationB()

a.do(opB)
b.do(opB)
```

Visitor for multiple dispatch

```
class A(object):

    def meth(self, other):
        other.methA(self)

    def methA(self, other):
        print "AA"

    def methB(self, other):
        print "BA"

class B(object):

    def meth(self, other):
        other.methB(self)

    def methA(self, other):
        print "AB"

    def methB(self, other):
        print "BB"

a,b = A(), B()
```

Visitor replaced by multimethods

```
from multimethod import multimethod

class A(object):
    pass

class B(object):
    pass

a,b = A(), B()
# New operations can be added without touching the class *AND* they are multiply dynamically dispatching

@multimethod(A,B)
def operation(_1,_2):
    print 'AB'

@multimethod(B,A)
def operation(_1,_2):
    print 'BA'

@multimethod(A,A)
def operation(_1,_2):
    print 'AA'

@multimethod(B,B)
def operation(_1,_2):
```

```
    print 'BB'

operation(a,a)
operation(a,b)
operation(b,a)
operation(b,b)
```

2.25 Rock, Paper, Scissors as Visitor

2.26 State pattern

```
class Happy(object):

    def comment(self):
        print "I'm so happy"

    def sing(self):
        print "Yippeeee !"

    def change(self):
        self.__class__ = Sad

class Sad(object):

    def comment(self):
        print "Oh noooooo"

    def sing(self):
        print "I don't want to sing, I'm too depressed"

    def change(self):
        self.__class__ = Happy

me = Happy()
me.comment()
me.sing()
me.change()
me.comment()
me.sing()
```

Discuss state as a possible implementation of oscillating species of particles.

Discuss strategy as an alternative.

Provide tests.

2.27 TODO Mock

2.28 Changing bindings through closures

Hopefully the intent of the following code is clear

```
def make_box(thing):

    def get_contents():
        return thing
```

```
    def set_contents(new_thing):
        thing = new_thing

    return get_contents, set_contents

get_thing, put_thing = make_box(1)
get_thing() # 1
put_thing(2)
get_thing() # Still 1 !
```

`get_contents` and `set_contents` are supposed to be two closures sharing the binding of `thing`.

However, performing L^{n}GB analysis of the name `thing` in `set_contents`, we realize that `thing`, by virtue of being bound (by assignment) in `set_contents` is local to `set_contents`, and therefore *shadows* `thing` in the enclosing scope (`make_box`): `set_contents` is not a closure at all; it does not share a binding with `get_contents`.

To make this work, we would need to tell Python that the act of binding `thing` in `set_contents` should not make it local. `global` does that, but it takes it too far: it jumps right past the binding of `thing` in `make_box`.

In Python 3, the new `nonlocal` keyword does exactly what we want. In Python 2, if you really insist on creating such a structure, you will have to resort to the trick of mutating rather that rebinding the shared location.

```
def make_box(thing):
    wrapped_thing = [thing]

    def get_contents():
        return wrapped_thing[0]

    def set_contents(new_thing):
        wrapped_thing[0] = new_thing

    return get_contents, set_contents

get_thing, put_thing = make_box(1)
get_thing() # 1
put_thing(2)
get_thing() # 2
```

`get_contents` and `set_contents` are both closures over `wrapped_thing`. `set_contents` changes the value of `wrapped_thing` without rebinding it, by mutating it. This means that no local binding is made in `set_contents` and that the desired binding is not shadowed.

2.29 TODO Write colour class

Static pseudo constructors, readers, conversion. Give test suite

Colour the particles. Give more tests.

2.30 TODO Profile before optimizing

cProfile

2.31 TODO Dummy display

2.32 TODO Cython

cython-etc demo

2.33 TODO Cythonize Vector2D

The dynamically generated vector class in not performance friendly. Use a hand-coded 2d Vector implementation. Aren't you glad you have one in your version control system. You do have one in your version control system, don't you!

Aren't you glad that you have a set of tests for the Vector's behaviour?

3 Tools

3.1 Mercurial

We will keep the code that we develop in the course in a *Distributed* Version Control System (DVCS) called Mercurial.

Mercurial should be pre-installed for you on the machines provided. You may have to install it yourself if you are working on your own machine.

From the perspective of this course, it is interesting to note that Mercurial is written almost entirely in Python. A tiny, performance critical poriton of the code, is written in C. I thoroughly recommend this approach to software development in the 21st century:

> Write your code in a powerful, expressive, enjoyable, high-productivity language (such as Python); use a low-level language for any small, perfomance critical portions where pure-Python is too slow.

Tools such as Cython (which we will explore at the end of the course) can make writing the low-level poritions, much easier.

For the purposes of this course, a basic understanding of

- `hg help`
- `hg init`
- `hg add`
- `hg ci`
- `hg log`
- `hg serve`

- `hg clone`
- `hg pull`
- `hg push`
- `hg update`
- `.hgignore`

should suffice.

Maybe, just maybe, you might also need to use `hg merge`. You will definitely need to understand `hg merge` if you adopt Mercurial for long-term use.

> Create your own, local Mercurial repository. Download the files `pygletversion.py` and `tkinterversion.py` and store them in your repository.

3.2 py.test

`py.test` is a testing framework which largely arose from the need to write an extensive test suite in the development of PyPy. As is often the case for tools which were created by experts for their own use, `py.test` is a very good tool.

The first and most obvious advantage of `py.test` over `unittest` (Python's standard testing framework), is the reduction in boilerplate code. The simplest `unittest` test looks something like this:

```
import unittest

class Tests(unittest.TestCase):

    def test_foo(self):
        self.assert_(1==1)

unitest.main()
```

The `py.test` equivalent is

```
def test_foo():
    assert 1==1
```

(ensuring that the test is stored in a file of the form `test_*.py` or `*_test.py`).

On top of this, `py.test` provides a wealth of features including

- Automatic test discovery
- Excellent error reporting
- Debugger integration

- Distributed test execution
- Sophisticated data injection
- Awareness of other Python test frameworks

3.3 TODO Mock

Mock

Mock was accepted into Python's standard library at version 3.3

3.4 cProfile

A profiler is a utility which measures where a program spends most of its resources.

Typically, around 10% of the source lines of any program are responsible for spending around 90% of the resources. If you are going to optimize your program, you **must** work on the critical 10%: optimizing any other part of your code will not give you significant improvements in performance.

It is usually almost impossible to work out which part of your code is responsible for the greatest resource expenditure, without the use of a profiler.

> Always profile your code before optimizing, to ensure that you know which part of the code needs optimizing.

There are 3 profiler modules in Python's standard library. You should use `cProfile` and ignore the other two.

For our current purposes, start with

```
python -m cProfile -o my_program.stats my_program.py
```

```
from pstats import Stats
stats = Stats('my_program.stats')
stats.print_stats()
stats.sort_stats('time').print_stats(20)
stats.strip_dirs()
stats.sort_stats('time').print_stats(20)
```

For more information, read the documentation.

3.5 TODO timeit

```
python -m timeit
```

3.6 Cython

Cython is a tool which helps in the development of Python programs which inculde both high-level and low-level code.

While Cython is a sophisticated tool with many features, the fundamental idea that we will explore in this course is the following:

Cython allows you to make your Python programs run more quickly by annotiating it with type declarations, in critical places. This allows Cython to compile the code down to efficient C, devoid of the runtime dynamism which hampers perfomance.

Thus, this function

```
def fib(n):
    if n < 2:
        return 1
    return fib(n-1) + fib(n-2)
```

when annotated with Cython type declarations, like this

```
cpdef int fib(int n):
    if n < 2:
        return 1
    return fib(n-1) + fib(n-2)
```

and compiled with Cython, speeds up by around two orders of magnitude.

To see this in action, unpack this tarball and, in the resulting `cython-etc` directory, do

```
python benchmark.py 30
```

(Expect some spurious error messages to appear during compilation.)

4 Recommended exercises

- Redirect standard streams (context managers)
- Currying (decorators)
- Coroutines (generator send)
- Creation proxy (descriptor protocol)

5 Topics

5.1 TODO Magic numbers

Magic numbers are literal numbers which appear in the source code without explicit reference to their meaning.

5.2 Nested functions, closures

```
1: def make_adder(a):
2:     def adder(b):
3:         return a + b
4:     return adder
5:
6: add3 = make_adder(3)
7: add4 = make_adder(4)
8: print add3(10)
9: print add4(10)
```

5.3 Unqualified name lookup: L^{n}GB

> Create a table of all the names which are bound in the above example, the lines on which those names are looked up, the lines on which they are bound, and the scope in which they are bound.

Mouse over to see answer.

name	looked up on line	bound on line	scope
`a`	3	1	local to `make_adder`
`add3`	6	8	global
`add4`	7	9	global
`adder`	4	2	local to `make_adder`
`b`	3	2	local to `adder`
make_adder	6,7	1	global

An *un*-qualified name is one that appears on its own, without information about the namespace in which is being sought. A qualified name is one which *does* have a namespace specification prepended to it. Thus in `a.b`, `a` is unqualified, but `b` is qualified.

When performing unqualified name lookup, Python proceeds as follows.

1. Is the name being looked up inside a function body? No: goto 5.
2. Is the name *bound* in this function? Yes: found in local scope. THE END.
3. Is this function defined inside another function? No: goto 5.
4. Is the name bound in any of the enclosing functions? Yes: found in the narrowest enclosing scope which binds this name, this function is a closure over the name. THE END.

5. Is the name bound in the global scope? Yes: found in global scope. THE END.
6. Is the name present in `__builtins__`? Yes: found in builtins. THE END.
7. Name not found. `NameError` exception is raised. THE END.

Notice that class scopes are **not** searched when names are looked up inside methods.

Notice that this can lead to situations where a name bound in one scope can *shadow* (render invisible from this location) the same name bound in another scope:

```
1: list = []  # shadows builtin list
2:
3: def outer(list): # shadows global and builtin list
4:     def middle():
5:         list = [1]  # shadows global, builtin and outer's list
6:         def inner():
7:             list = 2  # shadows global, builtin, outer's and middle's list
```

This whole process is summarized by the mnemonic LGB (local, global, builtin) which is intimately familiar to *every* Python programmer, including you.

Before Python 2.1, after failing to find a binding in a local scope, the process jumped straight to global scope, ignoring any bindings in enclosing nested functions. As of Python 2.1 the surrounding scopes are searched before looking in the global scope. To emphasize this, I like to augment the LGB mnemonic to L^nGB.

It is only in the case of nested function scopes that unqualified name lookup searches surrounding nested scopes: in the case of nested classes, the old-fashioned LGB (going straight from the inner local scope to the global scope) persists.

To make the new behaviour available in Python 2.1, one needed to `from __future__ import nested_scopes`. It became the only option as of Python 2.2.

Note that the L part of L^nGB analysis is performed at compile-time; the G and B parts are performed at run time.

5.3.1 `UnboundLocalError`

```
def foo():
    print a
    a = 3
```

1. At *compile* time, Python decides that `a` is local to `foo`, because it is bound inside `foo`.
2. At *run* time, Python tries to read the value of `a`, but it has not been bound yet: `UnboundLocalError`.

5.4 Qualified name lookup: ICS^n

An *un*-qualified name is one that appears on its own, without information about the namespace in which is being sought. A qualified name is one which *does* have a namespace specification prepended to it. Thus in `a.b`, `a` is unqualified, but `b` is qualified.

Consider the class `Foo` and its instance `f`:

```
class Foo(object):

    a = 1

f = Foo()
```

Inspect the namespaces of both the class and the instance

```
dir(Foo)
dir(f)
Foo.__dict__
f.__dict__
```

Perform the following mutations

```
f.a = 2
f.b = 3
```

and study the namespaces again

```
dir(Foo)
dir(f)
Foo.__dict__
f.__dict__
```

From this we can see that

- `a = 1` is a class attribute which is also accessible through instances of the class.
- `a = 2` and `b = 3` are instance attributes.
- Instance attributes *shadow* class attributes.

When performing qualified name lookup, Python proceeds as follows.

1. Is the name an attribute of the object on which qualifies it? Yes: found as an instance attribute. THE END.
2. Is the name an attribute of the *type* of the object which qualifies it? Yes: found as a class attribute. THE END.
3. Is the name an attribute of a *superclass* of the object which qualifies it? Yes found as a superclass attribute.
4. Name not found. `AttributeError` exception is raised. THE END.

This is the standard behaviour for normal classes, but there are mechanisms which can override or otherwise alter the procedure described above, in certain situations.

By analogy to L^nGB, we can summarize this procedure with the mnemonic ICS^n.

Note that ICS^n is a runtime mechanism.

5.5 Generator functions: `yield`

5.5.1 Iterator protocol

Python's iterator protocol is dead simple. It contains 3 ingredients:

1. `iter(...)`
2. `.next()` (`next(...)` in Python 3)
3. `StopIteration`

```
iterator = iter(iterable)
try:
    while True:
        ...
        iterator.next()
        ...
except StopIteration:
    print "Done"
```

5.5.2 Duck typing

Because of duck typing, it really doesn't matter *how* you provide these ingredients. Anything that provides the required features can be an iterator and/or iterable.

5.5.3 Generator functions

Generator functions are a convenience mechanism for writing iterables and iterators more easily.

> Write a generator function which will create iterables containing the items `1`, `2` and `3`.

```
def first_three():
    yield 1
    yield 2
    yield 3
```

> Write a generator function which will create iterables containing items which start with the one specified in its first argument, and whose difference is specified by the second argument.

```
def infinite(start, step):
    current = start
```

```
    while True:
        yield current
        current += step
```

Have you noticed that the latter can be thought of as a container of infinite size? This touches on the extremely important topic of laziness, which will discuss in depth later in the course.

Have you noticed that the second generator function is polymorphic? You probably thought of it as generating numbers, but it is capable of generating all sorts of things. (Duck Typing) Try:

```
infinite('', 'a')
```

5.5.4 Without generator functions

> Reimplement the two iterables in the previous exercises as classes, without using `yield`.

```
class First_three(object):

    def __init__(self):
        self.where = 0

    def __next__(self):
        if self.where < 3:
            self.where += 1
            return self.where
        raise StopIteration

    def __iter__(self):
        return self

class Infinite(object):

    def __init__(self, start, step):
        self._next = start
        self._step = step

    def __next__(self):
        self._current = self._next
        self._next = self._current + self._step
        return self._current

    def __iter__(self):
        return self
```

Generator functions don't give us the ability to do anything that we couldn't do without them, but they do make a whole class of ideas **much** easier to express.

5.5.5 Duck Typing

Notice that we didn't have to inherit from anything, didn't need to register anywhere. All we did was provide the necessary features for the object to behave as an iterable (`iter`, `.next()` and `StopIteration`) by any means available to us.

5.6 Decorator syntax

```
@XXX
def YYY(...):
    ...
```

is just syntactic sugar for

```
def YYY(...):
    ...
YYY = XXX(YYY)
```

We will cover this in depth, later in the course.

5.7 Writing decorators

```
def one(_):
    return 1

def inc(n):
    return n+1

def inc_by(n):
    def inc(x):
        return x+n
    return inc

import operator
from functools import partial

@partial(operator.add, 3)
@inc_by(3)
@inc
@inc
@inc
@one
def add(a,b):
    return a+b

print add

def report_args(fn):
    @functools.wraps(fn)
    def proxy(*args, **kwds):
        print "args are", args, kwds
        return fn(*args, **kwds)
    proxy.wrapped_function = fn
    return proxy

@report_args
def add(a,b):
    return a+b

add(1,3)

########################################
@inc_result_by(3)
def add(a,b):
    return a + b
```

```
assert add(1,2) == 6

# Don't forget functools.wraps

def add(a,b):
    return a + b
add = inc_result_by(3)(add)
```

5.7.1 TODO Decorator exercises

- Class decorator: chain; upgrade subset of methods to return `self`

5.8 `global`

Compare and contrast

```
def make_adder(aa):
    a = aa
    def adder(b):
        return a + b
    return adder
```

with

```
def make_adder(aa):
    global a
    a = aa
    def adder(b):
        return a + b
    return adder
```

How does the `global` declaration of `a` affect the behaviour of the code?

Mouse over to see answer.

`a` becomes a global variable; it is no longer local to `make_adder`. It is no longer enclosed by `adder`. The single *global* value of `a` is now shared by `add3` and `add4` and may be affected by any subsequent calls to `make_adder`.

Note that funciton parameters cannot be declared `global`.

Note that the L part of L^{n}GB analysis is performed at compile time and the whole funciton body is inspected while making a decision about whether a variable is local or not. Consequently, it is possible (thought not recommended) to declare a name global *after* its first binding in the function.

5.9 Binding mechanisms

In all of the following examples the name `a` is bound in the enclosing scope (that which surrounds the text in which `a` appears).

1.
```
a = ...
```

2.
```
def a(...):
    ...
```

3.
```
class a(...):
    ...
```

4.
```
import a
import b as a
from b import a
from b import c as a
```

5.
```
for a in ...:
```

6.
```
except XXX as a:
```

7.
```
with ... as a:
```

If you understand binding and its consequences in any one of these contexts, you understand it in all of them. This is an example of Python's regularity: a simple underlying mechanism pervades the language. Understanding such mechanisms thoroughly, makes you are more powerful Python programmer.

There is another situation in which names are bound, but this one differs in that the name is not bound in the surrounding scope, but inside the function being defined

-
```
def b(a):
    ...
```

In *all* of the above, the behaviour is guaranteed to be consistent wherever you see it (the exception being that parameter names cannot be declared `global`). There are other other situations which have much in common with binding, such as

- ```
  b.a = ...
  ```
- ```
  b.[x] = ...
  ```
- ```
  setattr(namespace, 'a', value)
  ```

but these are generally more complicated, and, because of operator overloading, can result in arbitrary code being execued and arbitrary effects taking place.

5.10 Value vs reference semantics

Does Python have value or reference semantics?

Can you predict the output of the following?

```
a = 1

def foo():
    a = 2

print a
```

Mouse over to see answer.

`a` will be `1`

```
b = [1]

def bar():
    b[0] = 2

print b[0]
```

Mouse over to see answer.

`b[0]` will be `2`

Formally, Python exhibits *value semantics* … but the values are references.

I don't think the formal answer is very helpful because, in practice, Python *seems* to exhibit a bit of both. It is far better to
```

understand exactly what is going on, rather than remembering a formal name.

5.10.1 TODO Describe what happens

5.11 @statcimethod

Given

```
class Foo(object):

    def normal(*args):
        return args

    @staticmethod
    def static(*args):
        return args

    @classmethod
    def class_(*args):
        return args
```

What will be the output of each of the following?

1. `Foo().normal()`
2. `Foo .normal()`
3. `Foo().static()`
4. `Foo .static()`
5. `Foo().class_()`
6. `Foo .class_()`

5.12 The private member-data dogma

In languages such as Java and C++, there is a very strong dogma stating that

> **All** member data **must** be private (or protected).

By religiously sticking to this principle, you can guarantee that the interface of your class (hierarchy) can remain invaraint, even when you change your mind about its implementation details. By pre-emptively pretending that member data are really member functions (by wirting trivial getters and setters), clients remain ignorant of where the data really are. If the organization of the data behind this interface changes, the clients need not be affected.

Trivial getters and setters are a means of making member data look like member functions. In contrast to Java and C++, Python also provides the means to make member functions look like member data, therefore there is no point (in Python) to pre-emptively pretend that member data are functions.

Let's write a simple class representing rectangles ignoring the data-must-be-private dogma.

```
class Rectangle(object):
```

```
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def get_area(self):
        return self.width * self.height

    def set_area(self, new_area):
        self.height = float(new_area) / self.width
```

The interface to this class looks like this

```
r = Rectangle(2,3)
assert r.width          ==  2
assert r.height         ==  3
assert r.get_area()     ==  6
r.width = 4
assert r.get_area()     == 12
```

In non-trivial code, one is often led to changing the internal organization of a component. In our toy example, this might amount to the decision to have `width` and `area` be the real data, and `height` to be a function of the former two. Our class would thus become

```
class Rectangle(object):

    def __init__(self, width, height):
        self.width = width
        self.area = width * height

    def get_height(self):
        return float(self.area) / self.width

    def set_height(self, new_height):
        self.area = self.width * new_height
```

The interface to this class looks like this

```
r = Rectangle(2,3)
assert r.width        ==  2
assert r.get_height() ==  3
assert r.area         ==  6
r.width = 4
assert r.area         == 12
```

Notice that the interface has changed: clients using the old interface would find that their code no longer works.

The Java/C++ solution to this problem, is to make all the components of the interface look like functions. This is achieved by making all the data members private, and making those which belong in the interface effectively public by writing trivial getters and setters:

```
class Rectangle(object):

    def __init__(self, width, height):
        self.__width = width
        self.__height = height
```

```
    def get_height(self):
        return self.__height

    def set_height(self, new_height):
        self.__height = new_height

    def get_width(self):
        return self.__width

    def set_width(self, new_width):
        self.__width = new_width

    def get_area(self):
        return self.__width * self.__height

    def set_area(self, new_area):
        self.__height = float(new_area) / self.__width
```

This is much more verbose that the original version, and it's difficult to spot the code which does something useful among all the noise created by the trivial getters and setters, but (in Java and C++) the pain is worth it in the long run because …

The interface of this class is

```
r = Rectangle(2,3)
assert r.get_width()  ==  2
assert r.get_height() ==  3
assert r.get_area()   ==  6
r.set_width(4)
assert r.get_area()   == 12
```

If we perform the same internal reorganization as before

```
class Rectangle(object):

    def __init__(self, width, height):
        self.__width = width
        self.__area = width * height

    def get_height(self):
        return self.__height

    def set_height(self, new_height):
        self.__area = new_height * self.__width

    def get_width(self):
        return self.__width

    def set_width(self, new_width):
        self.__width = new_width

    def get_area(self):
        return self.__area

    def set_area(self, new_area):
        self.__area = new_area
```

Even after this reshuffle, the interface used by the clients remains unchanged. This is why we make all member data private (or protected) in Java and C++.

In Python, even the naively written, public-data-in-interface-containing version of the class can be reorganized internally without changing the interface:

```
class Rectangle(object):

    def __init__(self, width, height):
        self.width = width
        self.area = width * height

    @property
    def height(self):
        "This will be the doc of the property"
        return float(self.area) / self.width

    @height.setter
    def height(self, new_height):
        self.area = self.width * new_height
```

The interface of this class is identical to that of the original.

It is perfectly OK to have public data in the interface of a Python class.

5.13 Properties

Properties allow the use of functions to implement something that looks like a data member.

```
from random import randint
class Foo(object):

    def __init__(self):
        self._max = 100

    @property
    def datum(self):
        "Docstring"
        print "Running the getter."
        return randint(self._max)

    @datum.setter
    def datum(self, value):
        print "Running the setter."
        self._max = value
```

Instances of `Foo` appear to have a data attribute called `datum`, but in reality, functions handle each request to read or assign the member:

```
f = Foo()
f.datum
f.datum = 10
```

Note that properties can be read-only

```
property(getter)
```

or even write-only

```
property(fset=setter)
```

Note that properties are bound as **class attributes**.

Note that properties only work correctly when installed in new-style classes.

In pre-decorator-syntax versions of Python, the above example would be written as

```
from random import randint
class Foo(object):

    def __init__(self):
        self._max = 100

    def get_datum(self):
        print "Running the getter."
        return randint(self._max)

    def set_datum(self, value):
        print "Running the setter."
        self._max = value

    datum = property(get_datum, set_datum, doc='docstring')

    del get_datum, set_datum
```

5.14 Special/Magic methods

Names starting and ending with two underscores are formally known as as *special names*, and colloquially known as *magic names*.

Generally speaking, these are names that you do not typically access directly: you usually let Python access them on your behalf in certain well defined situations.

- `__builtins__` is accessed by Python's runtime as the B part of L^{n}GB
- `__new__` and `__init__` are accessed by Python's runtime when classes are instantiated.
- `__call__` is accessed by Python's runtime when objects are called.
- `__iter__` and `__next__` are accessed by Python's runtime when objects are iterated.
- `__add__` is accessed by Python's runtime when objects are added.
- `__getitem__` or `__setitem__` are accessed by Python's runtime when objects are subscripted/indexed.
- `__enter__` and `__exit__` are accessed by Python's with blocks.

Magic names can often be thought of as hooks into protocols, or means of overloading operators.

While it is not unusual to implement methods with magic names, it is unusual to access magic methods directly by name. The main reason for accessing special methods by name, is in the implementation of extended inherited magic methods. For

example

```
class Sub(Super):

    def __init__(self, *args, **kwds):
        self.frob(*args, **kwds)
        Super.__init__(self, *args, **kwds)
```

5.15 Exceptions

Think of exceptions as

1. Messages from the top of the call stack, to anyone who cares lower down the stack
2. An alternative channel of data flow out of functions (the primary channel being `return`)
3. An additional branching mechanism (along with `if`, `for` and `while`)

The core of the message carried by an exception is its type. To create new kinds of messages, create new types:

```
class TimeToGoHome(Exception):
    pass
```

To send such messages explicitly, `raise` them

```
raise TimeToGoHome("You have an appointment with the plumber.")
```

To receive such messages, use the `try ... except` construct

```
try:
    job.work_hard()
except TimeToGoHome:
    job.drop_everything()
    go_home()
```

5.15.1 `finally`

```
initialize()
try:
    do_work()
finally:
    clean_up() # Always! Regardless of whether an exception is
               # unwinding the stack or not
```

5.16 Context managers

The try-finally construct usually needs to be accompanied by boilerplate code. Context managers are a mechanism for creating abstractions which hide the boilerplate.

The most obvious example is that of ensuring that file handles are

```
try:
    stream = open(...)
    do_stuff(stream)
finally:
    stream.close()
```

can be expressed more cleanly as

```
with open(...) as stream:
    do_stuff(stream)
```

`open` is a *context manager*. When used in conjunction with `with`-statements, context managers ensure that certain code is executed on entry to the `with`-block, and other code is executed when the block is exited.

The `with`-statement became available in Python 2.5 (`from __future__ import with_statement`).

5.16.1 Context manager development

5.16.2

5.16.3

5.17 New-style classes

Initially, Python's built-in types (`int`, `float`, `list`, `dict` etc.) were fundamentally different from the user-defined types (classes).

Some of the more obvious consequences of this are:

- It is impossible to subclass the built-in types.
- `type(X(...))` returns
    - `X`, if `X` is a built-in type
    - `types.InstanceType`, if `X` is a class, regardless of what the actual class is.
- `type(X)` returns
    - `type`, if `X` is a built-in type
    - `types.ClassType`, if `X` is a class
- The need for language implementors to implement and maintain two separate object models
- The need for language users to learn two separate object models

This was perceived to be a design flaw, and Python underwent a process of *Type-class unification*: built-in types and classes were given a single, shared, underlying sturcture. In order to do this, it was necessary to change the behaviour and structure of classes at a fundamental level, which would break backwards compatibility in subtle ways. It was therefore decided to keep the original classes (for backward compatibility), even after type-class unification, alongside the new ones.

Therefore, from Python 2.2, when writing a class, you may choose between a creating a *classic* class or a *new-style* class.

By default (in Python 2), classes are classic. To make a new-style class, make it a direct or indirect subclass of *object*. So

```
class Foo:
    pass
```

is a classic class, while

```
class Foo(object):
    pass
```

is an equivalent new-style class.

This mechanism for distinguishing betwen the two was chosen, because it did not require any new syntax to be introduced.

In Python 3 you have no choice: all classes are new-style. So, while

```
class Foo:
    pass
```

is a classic class in Python 2, it is a new-style class in Python 3.

You should get in the habit of making all your classes new-style (inherit directly or indirectly from `object`, remember?), because

- Only new-style classes will be available in Python3; by writing new-style classes you avoid any nasty backward-incompatibility surprises when someone (maybe you) gets around to porting the code to Python 3.
- Certain features, such as `property` and `super` only work properly in new-style classes

5.18 TODO `__future__`

A mechanism for mitigating the impact of backward compatibility breaking features.

5.19 TODO Duck Typing

5.20 TODO Dynamic dispatch and multimethods

5.21 Generating classes at runtime

5.21.1 TODO Write class definition inside function

5.21.2 TODO `type`

`class` is just syntax sugar for collecting objects and passing them to `type`.

Instantiating `type` creates instances of `type`: classes.

You can subclass `type`, creating *different kinds of types*. A subclass of `type` is called a *metaclass*: the type of the type of an object.

Metaclasses were mostly used to change the process of object construction. Now that we have class decorators, these have replaced most uses of metaclasses.

Can you think of something that can be done with metaclasses but not with class decorators?

How does at type affect the behaviour of its instances, **after** the instance has been created?

6 Challenge exercises

6.1 Use old versions for now

Look at the index of challenge exercises.

Provide mind-bending exercises (preferably in the form of test-suites) and tell them to get on with it.

6.2 TODO Multimethods

6.3 TODO Redirect streams

6.4 TODO Partial

tests, solution

6.5 TODO Currying

Implementation:

```
def curry(fn, old_args = ()):
    def proxy(*new_args):
        all_args_so_far = old_args + new_args
        if len(all_args_so_far) >= fn.func_code.co_argcount:
            return fn(*all_args_so_far)
        else:
            return curry(fn, all_args_so_far)
    return proxy
```

test:

```
@curry
def add5(a,b,c,d,e):
    return a+b+c+d+e

assert add5(1,2,3,4,5) == 15
assert add5(1,2,3,4)(5) == 15
assert add5(1,2)(3,4)(5) == 15
assert add5(1)(2)(3)(4)(5) == 15
```

```
print "All OK"
```

6.6 TODO Descriptors

6.7 TODO Method binding

[explanatory test suite]

6.8 TODO Rich comparison explorer

N.B. this uses a stream redirection decorator, which is (approximately) the solution to one of the extra exercises in this course.

[Explanatory comments, solution and tests].

6.9 TODO Dynamic module creation

6.10 TODO Monkeypatching

Maybe as AOP, decorate some class or module with a cross-cutting concern.

6.11 TODO Metaclasses

6.12 TODO Decorators instead of metaclasses

6.13 TODO SQL generation

[Outline solution].

6.14 TODO Coroutines

7 Summary

7.1 Duck Typing

Duck Typing is vital to getting the most out of Python.

7.2 Version Control

Just do it. With git and Mercurial, there's no excuse! Use a DVCS for everything you do.

7.3 Testing

Helps to

- Focus your efforts on one task
- Document how your code works

- Give confidence in your code
- Refactor confidently

And it's much easier in Python than in Java and C++

py.test is a very useful test framework. It was written by Python experts for their own intensive use, and, as such, contains many very useful features.

7.4 We practised writing lots of Python code

Classes and functions are the bread and butter of Python programming.

7.5 Fundamental understanding

A solid understanding of certain Python ideas will make you a much more effective Python programmer.

For example

- Universal behaviour of bindings
- LGB (and ICS)
- Duck Typing
- Rebinding vs. mutation
- Everything is a first-class object (functions, classes, modules etc.)

7.6 Duck Typing

This is **really** important. If you don't emphasize Duck Typing in your Python programming, then you are likely to be struggling against the language, rather than benefitting from it.

7.7 Context managers

The `with` statement allows you to provide and use abstractions for patterns in which some code is is embedded inside some other code.

the contextlib.contextmanager decorator makes it very easy to write your own.

7.8 Duck Typing

Just in case you forgot, Duck Typing is really important in Python.

7.9 Decorators

The decorator syntax provides a convenient way of providing utilities.

Beyond the syntax, it's all built on the idea of passing functions or classes as first class objects into the decorator, and unleashing the full power of Python to manipulate them.

A typical decorator implementation will be a closure made by writing 3 nested functions.

- The outer level accepts the parameter to the decorator
- The middle level **is** the decorator
- The inner level is the replacement of the object being decorated

7.10 Generators

Intended as a tool for easy creation of iterators. Iterators are used to create lazy components. Lazy components enable the efficient handling of large (even infinite) data sets.

Generators lend themselves to all sorts of interesting (ab)uses.

Another way of looking at them is as suspendable functions which maintain state between invocations.

Either way, they can make all sorts of tasks much easier.

7.11 Metaprogramming

Python allows you to manipulate the structure of the program components from within the program. This allows you to write utilities which create program components.

7.12 Duck Typing

Don't forget Duck Typing

7.13 Dynamism

Python allows you to change all sorts of things at run time. This doesn't mean that you should be doing it all the time. Used judiciously, it can be of great help.

7.14 Profiling

Don't optimize your code without profiling it first. If you don't profile before optimizing, it is likely that you will be optimizing the wrong part of your code. This usually results in your code getting uglier and more fragile, without any useful gain in speed.

Python has a (not particularly sophisticated) profiling module in its standard library (cProfile).

7.15 Extensions

Python is designed to be extendable in low-level languages, via the Python/C API.

Tools exist to make this task easier, such as SWIG or Boost.python.

We looked at Cython, which allows us to compile pure python code into C and use it as an extension module. It also allows you to add type declarations in places where you want to forgo dynamicm in exchange for increased performance. Cython can also be used as a bridge between Python and existing low-level libraries.

We also looked at ctypes, a standard FFI (Foreign Function Interface) module, which allows you to use C shared libraries from Python.

7.16 Duck Typing

Did I mention Duck Typing ?

7.17 Closures

Closures provide a quick and cheap way of combining functionality and state. In this repsect the can sometimes act as a cleaner, cheaper, more efficient substitute for classes.

You can use nested function definitions to implement function factories. These return closures which are functions belonging to some family or related functions.

Closures are usually the easiest and neatest way of implementing decorators.

7.18 Course scope

People with vastly different backgrounds and needs come to this course. It is difficult to cater for everyone.

I hope that I have managed to strike a reasonable balance between more and less advanced topics, and that there was something useful for everyone.

Validate