

PyCon 2009 Chicago

March 26, 2009

## Python 401: Some Advanced Topics (A Tutorial)

```
paginatorstuff = models.TextField()
pagctype = models.CharField(max_length=10)
pagid = models.IntegerField()
class Meta:
    db_table = 'page'
class Admin:
    list_display = ('pagpath',)
    def __unicode__(self):
        if self.pagsecid:
            section = self.pagsecid
        else:
            section = "(None)"
```



### Class Objectives

In this class, we will

- Think more deeply about how the interpreter works
- Take another look at % string interpolation
- Demonstrate how to implement iterators
  - Including class-based iterators
- Investigate ways of creating generators
- Help you to understand descriptor-based name lookups and properties
- Describe and demonstrate metaclasses
- Challenge you to think of applications
- Maybe have a little fun
  - As geeks will when talking programming
- Regard some of this as Python 3 preparation

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-2



## String Interpolation

```
paginatorstuff = models.TextField()
pagctype = models.CharField(max_length=10)
pagid = models.IntegerField()
class Meta:
    db_table = 'page'
class Admin:
    list_display = ('pagpath',)
    def __unicode__(self):
        if self.pagsecid:
            section = self.pagsecid
        else:
            section = "(None)"
```



## String Formatting Basics (1)

- **Well-known feature with some little-known implications**
  - Superseded in 3.0 with a new formatting system
- **Format strings are the left-hand operand of “%” operator**
- **They contain *conversion specifiers*, each introduced by a percent sign**
  - A letter indicates what type of formatting is required
- **Three different forms:**
  - format-string % value
  - format-string % tuple
  - format-string % mapping



## String Formatting Basics (2)

- **With a single value, there must be a single format effector in the format string**
  - Format effectors are like `%s`—a percent sign followed by a code
- **With a tuple there must be a format effector for each item in the tuple**
  - Format effectors are as for single values
- **For a mapping, items are looked up by name**
  - Format effectors are like `%(name)s`
- **The interpreter examines the format**
  - Determines whether mapping is required or not
- **Next it examines the right-hand operand**
  - If a single value or a tuple, matches the number of elements to the number of format effectors

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-5



## Formatting Operations

- **Examples of different right-hand operands**
  - *A single value:* (if LHS has only one format code)

```
>>> "It's a %s today" % dow
'It's a Sunday today'
```
  - *A tuple:* format specifiers take successive tuple elements

```
>>> "%s owes me %.2f" % ("Dougie", 123.45)
'Dougie owes me 123.45'
```
  - *A dict or other mapping:* format specifiers include a parenthesized mapping key

```
>>> "My %(relation)s is aged %(age)d" % {
    "relation": "brother", "age": 60}
'My brother is aged 60'
```
- **Q: How do I interpolate a tuple?**
- **A: Put it as the only element in another tuple!**

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-6



## Conversion Specifiers

- The "%" character, which marks the start of the specifier.
- Mapping key (optional), consisting of a parenthesised sequence of characters (for example, (somename)).
- Conversion flags (optional), which affect the result of some conversion types.
- Minimum field width (optional). If specified as an "\*" (asterisk), the actual width is read from the next element of the tuple in *values*, and the object to convert comes after the minimum field width and optional precision.
- Precision (optional), given as a "." (dot) followed by the precision. If specified as "\*" (an asterisk), the actual width is read from the next element of the tuple in *values*, and the value to convert comes after the precision.
- Length modifier (optional).
- Conversion type.

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPy 401-7



## Anatomy of a Conversion Specifier

**%[(mapping-key)][#0- +][width][.precision][hLl]t**

**Optional.** Selects mapping element to use for value lookup

**Conversion flags:**

**#** Use "alternate form" (if defined for this type)

**0** pad with zeros

**-** left-adjust converted value

**(space)** space if positive, "-" if negative

**+** Insert sign, whether + or -

Not used by Python, but acceptable to C libraries

\* indicates width and/or precision should be taken from value list

Conversion type – see next slide

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPy 401-8



## Most Common Conversion Types

<b>s</b>	String – uses value's <code>str()</code> method
<b>d</b>	Decimal (integer)
<b>F, f</b>	Floating-point decimal: value will be rounded to an appropriate number of places
<b>G, g</b>	General: fixed or floating, depending on data value
<b>c</b>	Single character
<b>X, x</b>	Unsigned hexadecimal
<b>%</b>	Inserts a literal percent sign

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-9



## Simple Formatting Examples

- **Conversion specifiers can include a length and a number of places**
  - Negative lengths indicate left-justification
  - If length begins with "0" zero-filling will be used

```
>>> "%12s %5d" % ("Welcome", 23)
'   Welcome    23'
>>> "%-12s %5d" % ("Welcome", 23)
'Welcome      23'
>>> "%(lang)s: %(ct)d" % {"lang": "Python", "ct": 23}
'Python: 23'
>>> "%8.2f" % 314.159
' 314.16'
>>> "image%03d.png" % 2
'image002.png'
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-10



## Formatting with Variable Width and Precision

- Use an asterisk to make width and/or precision data dependent

- This only works with tuple data
- Mappings have no “next element”

```
>>> for (i, j) in ((8, 2), (10,5), (0, 0)):
...     print "%*.*f" % (i, j, 3.14159)
...
      3.14
      3.14159
3
>>> for (i, j) in ((8, 2), (10,5), (0, 0)):
...     print "%(num)*.*f" % {"i":i, "j":j, "num": 3.14}
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: * wants int
>>>
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-11



## Fun With Mappings

- locals() and globals() return mappings of the given namespace

```
>>> def fmt(date, amount, narr):
...     return "%(date)12s %(narr)-25s %(amount) 8.2f" % locals()
...
>>> fmt("12-Dec-2009", 1234.56, "PC Purchase")
' 12-Dec-2009  PC Purchase                1234.56'
>>> fmt("12-Dec-2009", -1234.56, "PC Purchase")
' 12-Dec-2009  PC Purchase                -1234.56'
>>>
```

- When the formatting engine sees %(name) it knows to expect a mapping
  - Gets value from the `__getitem__()` method of the right-hand operand
- So, can we provide an instance with `__getitem__()` as the right-hand operand?

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-12



## Can Program `__getitem__()` to Requirements

- This is a very simple example, but I hope you get the idea

```
>>> class C:
...     def __getitem__(self, name):
...         return name*2
...
>>> c = C()
>>> *(a)s *(bb)s" % c
'*(a)s *(bb)s'
```

```
>>> "%(a)s %(bb)s" % c
'aa bbbb'
```

```
>>>
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-13



## An Also Use This to Extract Instance Attributes

- Just use the instance's `__dict__` as the mapping!
  - Works with classes too, but less interesting

```
>>> class CC:
...     def __init__(self, one, two, three):
...         self.one = one
...         self.two = two
...         self.three = three
...
>>> "%(one)-12s %(three)d %(two)15s" % \
      CC("ONE", 22222, 3).__dict__
'ONE          3          22222'
```

```
>>>
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-14



## Or Even Go Right Over the Top

- Not something to try on user input
  - Due to the risk of “anything injection” ☹
- Can perform arbitrarily complex computations on parenthesised text

```
>>> class Bogus():
...     def __getitem__(self, expr):
...         return eval(expr, globals(), locals())
...
>>> a = 3
>>> b = 4
>>> c = "I dunno"
>>> bogus = Bogus()
>>> '%(a*b)d %( ".join([c]*b))s' % bogus
'12 I dunno, I dunno, I dunno, I dunno'
>>>
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-15



Python 401: Lesson 2

## Iteration in Python

```
    navbarstuff = models.TextField()
    pagctype = models.CharField(max_length=10)
    pagid = models.IntegerField()
class Meta:
    db_table = 'page'
class Admin:
    list_display = ('pagpath',
                    'pagctype',
                    'pagid')
    def __unicode__(self):
        if self.pagsecid:
            section = self.pagsecid
        else:
            section = "(None)"
```

 **Holden  
Web**  
What Do You Need to Know Today?

## Iteration in Python

- **Python has several iteration contexts**
  - Basically, anywhere the keyword `for` is used
  - `while` isn't quite the same
- **You are probably used to iteration over containers**
  - `for item in list:`
  - `for item in tuple:`
  - `for (key, value) in dict.items():`
- **How does the interpreter interact with these objects to iterate over them?**
  - It uses its *iteration protocol*
- **Back in “the old days” iteration used an object's `__getitem__()` method**
  - Called it repeatedly with successively higher index values
  - `IndexError` caused silent termination of the loop
  - This mechanism is still available as a fallback
- **Nowadays a somewhat more sophisticated and general mechanism is used**

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-17



## Iteration Support in 2.1 and Before (and Still, If Needed)

- **Lists and tuples have a `__getitem__(self, i)` method**
  - `for` loops would call it repeatedly with successively higher values of `i`
  - Failure of `__getitem__()` occurred on exhaustion of the list/tuple
  - Indicated by an `IndexError` exception
    - Which the interpreter swallows silently
- **Could not iterate directly over dicts before 2.2**
  - Had to call `d.keys()` or `d.items()` or `d.values()`
  - Then iterate over that
  - Now iterating over a dict gives you the keys
- **And let's not forget strings**
  - They act like containers too
  - Though they are anomalous
    - Because there is no character data type
    - Characters are just strings of length one
  - In fact strings today are still not iterables

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-18



## Legacy Iteration Support Demonstrated

```
>>> class Thing:
...     def __init__(self, howbig):
...         self.howbig = howbig
...     def __getitem__(self, i):
...         print "Accessing item", i
...         if i > self.howbig:
...             raise IndexError
...         return i * i
...
>>> for i in Thing(2):
...     print "Got", i
...
Accessing item 0
Got 0
Accessing item 1
Got 1
Accessing item 2
Got 4
Accessing item 3
>>>
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-19



## Python 2.2 (and Later) Iteration Protocol

- **Python 2.2 introduced *iterables* and *iterators***
  - All iterators are iterable
  - The interpreter constructs iterators as required
- **The older mechanism remains as a fallback**
  - `__getitem__()` is now mostly used for random access on sequences
    - Also used on (more general) mappings, of course
- **The new iteration protocol lets classes specify iteration behavior**

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-20



## Iterables vs Iterators

- WTF?
- Suppose you were to write  
lst = [1, 2, 3, 4]  
pairs = [(i, j) for i in lst for j in lst]
- Would you expect this result?  
[(1, 2), (3, 4)]
  - No?
  - Because?
- An *iterable* can be iterated over many times in parallel
- The interpreter creates an *iterator* for each iteration
- With full program control we can create iterables and iterators at will

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-21



## The Iterator Protocol

- Calling `iter()` on an iterable object returns an iterator
  - Call's object's `__iter__()` method
  - Easiest option is for `__iter__()` to just return `self`
    - Which is what iterators do
    - But multiple iterations then interfere with each other
- Sophisticated iterable implementations (list, tuple, etc.) use `__iter__()` to create a new *iterator* for each call
  - In all cases the object returned by `__iter__()` has a `next()` method that produces the values on successive calls
  - Which really just says “it’s an iterator”
- Returned iterator object must have a `next()` method
  - *Either* returns next value in the iteration
  - *Or* raises `StopIteration` to indicate no more values

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-22



## Observing Iterables

- An *iterable* is something you can iterate over
- It must implement an `__iter__()` method
- This method must return an *iterator*
  - Iterators implement `__iter__()` and `next()`
- Lists, tuples and strings are iterables
  - Dicts too, now iterate – over keys

```
>>> l = [1,2,3]
>>> l.__getitem__(2)
3
>>> ll = l.__iter__()
>>> ll
<listiterator object at 0x7ff2c38c>
>>> dir(ll)
['_class_', ..., '__iter__', ... , 'next']
>>>
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-23



## Implementing Iterables and Iterators (1)

- Python defines the “iterator protocol” that iterators must implement
- It's very simple. The first method is also required of iterables
- `obj.__iter__()`
  - Must return a (new?) iterator object that can iterate over all the objects in the iterable**
    - Returning a new object means several iterations can be in progress at once over the same iterable
      - Each iteration will be independent
    - Returning self is therefore probably only appropriate for iterators
- `obj.next()`
  - Must return the next object in the iterator or, if the iterator is exhausted, raise a `StopIteration` exception**
    - The interpreter traps the exception internally when iterating
    - It must be caught if `next()` is called directly
    - This is *not* required for iterables

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-24



## Implementing Iterables and Iterators (2)

```
class EveryOther:
    def __init__(self, seq):
        self.seq = seq
        self.idx = 0
    def next(self):
        self.idx += 1
        if self.idx >= len(self.seq):
            raise StopIteration
        value = self.seq[self.idx]
        self.idx += 1
        return value
    def __iter__(self):
        return self

print [x for x in EveryOther([1, 2, 3, 4, 5, 6, 7, 8,
9])]
[2, 4, 6, 8]
```

- Why not print `EveryOther([ ... ])`?

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-25



## “Iterability” Is a Fundamental Property

- **We don't always want to create the sequences**
  - Costs us space
  - Costs us time
  - Costs us customers!!!
- **Implementing iterators is easy**
- **There are some useful iterators readily available to you**
  - `xrange` – like `range`, but an iterator
  - `dict.iteritems` – iterates over (key, value) pairs
  - `dict.iterkeys` – iterator equivalent of `dict.keys()`
- **The `itertools` standard library module contains many very helpful functions**
  - Though we don't have time to cover it here
  - The functions it defines work with iterators as well as sequence objects
  - Use the documentation ...

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-26



## Don't Forget enumerate ( )

- Have you ever written something like this?  

```
index = 0
for value in lst:
    # do something with index and value
    index += 1
```
- Have you ever forgotten the increment?
  - I don't believe you if you said "no"
  - I should trust my audiences more?
- There's a much easier way (if you understand unpacking assignment):  

```
for index, value in enumerate(lst):
    # do something with index and value
```
- It's an iterator:  

```
>>> dir(enumerate([1, 2, 3]))
[... , '__iter__', ..., 'next']
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-27



Python 401: Lesson 3

## Generators and Generator Expressions

```
gnavbarstuff = models.TextField
pagctype = models.CharField(m
pagid = models.IntegerField()
class Meta:
    db_table = 'page'
    def __str__(self):
    list_display = ('pagpath'
    unicode__(self):
    if self.pagsecid:
        section = self.pagsec
    else:
        section = "(None)"
```

 **Holden  
Web**  
What Do You Need to Know Today?

## Generator Functions

- **How do we handle the situation where computation is needed to create each element of the sequence?**
- **A class of objects called *generators* are also iterable**
- **You can define your own *generator functions***
  - Body is like normal function body
  - But it contains one or more `yield` expressions
  - This is determined by static analysis
- **Some care is required in reading**
  - Nothing actually screams “THIS IS A GENERATOR FUNCTION”!
- **Calling the function returns a generator-iterator**
  - Often just referred to as a “generator”

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-29



## Generators: Beyond the Container

- **New keyword `yield`**
- **`yield` can only be used inside a function body**
  - Easily policed with static checks
- **But does the function return anything?**
- **Functions containing a `yield` expression do not “run” when called!**
  - Instead the call returns a *generator-iterator*
  - When used in a loop the generator `yields` successive values
- **Guess what: generator iterators obey the iterator protocol!**
  - No need to create a sequence to hold the values!
- **Funkiness: since 2.5 `yield` becomes an *expression***
  - Can feed values back in to the generator function
  - But use cases are few and far between, so I ignore it here
  - Closest thing to coroutines in Python right now?

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-30



## Introduction to Generators

- **A generator can be used in place of a sequence**
  - In any iterative context
  - It creates values, *one by one, on demand*
  - “Demand”, in this sense, means a call of the generator’s `.next()` method

- **How do you use these generators?**
  - The usual way is just to iterate over them

```
generator = generatorFunction()
for x in generator:
    process(x)
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-31



## Generator Function Example

- **Suppose we needed [1, 1, 2, 2, ..., N, N]**
  - For some value of N to be decided by the user

```
>>> def pairsupto(N):
...     for i in range(N): # 0, 1, ..., N-1
...         yield i+1
...         yield i+1
...
>>> [j for j in pairsupto(3)]
[1, 1, 2, 2, 3, 3]
>>>
```

- **Duplication is easy with two successive `yields`**

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-32



## Iterating Over a Generator

- **When the function is called the function namespace is created**
  - Naturally it contains the function arguments
- **The function body starts running on the first call of `.next()`**
  - `yield` expression causes `.next()` to return the expression's value
- **The namespace continues to exist until**
  - *Either* the function body returns (raising `StopIteration`)
  - *Or* the generator is no longer referenced
    - Resulting, ultimately, in garbage-collection

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-33



## Iterating Over a Generator (Summary)

- A
- **Iterating over the generator repeatedly calls its `.next()` method**
  - Each time `.next()` is called the function body resumes
  - A `yield` expression generates a result for `.next()`
    - Expression value passes into iteration
    - Namespace is retained by generator for re-activation
  - When the function returns (terminates) this causes `.next()` to raise `StopIteration`

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-34



## Advantages of Generators

- **Can express producer-consumer algorithms more naturally**
  - Generation of values is cleanly separated from their processing
- **Each generator has its own namespace**
  - Can `yield` at whatever point(s) is convenient
  - Simplifies production logic and clarifies code
- **Complex logic can be layered inside the generator API**
  - Avoids messy solutions like callback arguments
  - No matter how complex the production algorithm
  - Generators can use other generators
    - A little like dataflow processing
- **Generator function yields each value as it's produced**
  - Call for next value automatically resumes generator
- **Consumer simply iterates over the generator**

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-35



Python 401: Lesson 4

## Descriptors and Properties

```
    navbarstuff = models.TextField()
    pagctype = models.CharField(max_length=10)
    pagid = models.IntegerField()
    class Meta:
        db_table = 'page'
        :
        list_display = ('pagpath',
                        'unicode_',
                        )
        if self.pagsecid:
            section = self.pagsecid
        else:
            section = "(None)"
```

 **Holden  
Web**  
What Do You Need to Know Today?

## Attribute Lookup: `x.foo`

- There's a well-defined algorithm for attribute and method lookup (qualified names on an instance)
  - First look in the instance's dict: `x.__dict__`
  - Then look in `x.__class__.__dict__`
  - Then look in the instance's class's superclass's `__dict__`
    - Multiple superclasses: so-called *C3 algorithm*
  - If you don't find it, it may be a virtual attribute
  - So call `x.__class__.__getattr__()` with the attribute name as argument
    - If `__getattr__()` not present, raise `AttributeError`
- Old-style and new-style classes use slightly different mechanisms
  - Otherwise there's no need for two different styles, right?
  - For new-style access, interpreter just calls `__getattribute__()`
  - Which does all the required funky stuff

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-37



## Attribute Lookup Mechanics in New-Style Classes

*acknowledgements to Alex Martelli*

```
• x = C()
return x.foo is approximately equivalent to:
if hasattr(C, 'foo'):
    d = C.foo
    D = d.__class__
    if hasattr(D, '__get__') \
        and (hasattr(D, '__set__')
            or 'foo' not in x.__dict__):
        return D.__get__(d, x, C)
elif 'foo' in x.__dict__:
    return x.__dict__['foo']
elif 'foo' in type(x).__dict__:
    return type(x).__dict__['foo']
else:
    for base in type(x).__mro__:
        if 'foo' in base.__dict__:
            return base.__dict__['foo']
if hasattr(x, '__getattr__'):
    return x.__getattr__('foo')
raise AttributeError("x has no attribute 'foo'") # else barf!
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-38



## A Note About Instance Methods

- **When you look up a *callable* on an instance**
  - The interpreter creates a “bound method”
  - This is specific to the instance (it knows “its self”)
- **This means that a method call carries object creation overhead:**

```
>>> import timeit
>>> t1 = timeit.Timer("c.method()", "from c import c")
>>> t2 = timeit.Timer("x()",
                    "from c import c; x=c.method()")
>>> t1.timeit()
0.24699997901916504
>>> t2.timeit()
0.16300010681152344
```

- **c.py reads as follows**

```
class C(object):
    def method(self):
        pass
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-39



## Properties

- **A way of interposing code between client and server of a namespace**
  - Getter, setter and deleter plus a doc string
- **Getter: access a value as a name**
  - Return result of a function (method) call
- **Setter: bind a value to a name**
  - Run method to distribute value components among multiple attributes
  - Do complex state-dependent validations
- **Deleter: “delete” name**
  - Will normally invalidate attempts to use getter
  - Until setter is used again
- **Doc: right, documentation**
  - No, that's *not* the one you can miss out!
- **Properties are most easily established by use of the `property()` built-in**

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-40



## The property Function

- `property(fget=None, fset=None, fdel=None, doc=None)`
- When a specific argument is not provided, that functionality is unavailable
- How does the interpreter know when an attribute is a property?
  - We looked at that already in new-style attribute lookup
  - Looks for `__get__`
  - And `__set__` if being written
  - Can also make use of `__delete__`
- In the simple read-only case property can be used as a decorator
  - `fget` is first argument to `property`
  - `@property`

```
def f(...):  
    ...  
is equivalent to  
def f(...):  
    ...  
f = property(fget=f)
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-41



## Read-Only Attributes

- Simplest properties are *read-only*:

```
class Person(object):  
    def __init__(self, firstname, lastname):  
        self.firstname, self.lastname = firstname, lastname  
    def fullname(self):  
        return self.firstname + ' ' + self.lastname  
    fullname = property(fullname)
```

```
me = Person("Steve", "Holden")  
me.fullname  
'Steve Holden'  
>>> me.fullname = "Steve Holden"  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: can't set attribute  
>>>
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-42



## property Can Be Used As a Decorator\*

- Equivalent declaration:

```
class Person(object):
    def __init__(self, firstname, lastname):
        self.firstname, self.lastname = firstname, lastname
    @property
    def fullname(self):
        return self.firstname + ' ' + self.lastname
```

- It really is just syntax!
- Note that this calls `property` with one argument
  - The getter
- Consequently this creates a read-only property
  - No setter or deleter (or doc ...)

\* See Appendix A

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-43



## Property Creation Can Pollute Your Namespace

```
class Person(object):
    def __init__(self, name="Unknown Individual"):
        self.fullname = name
    doc = "The person's name"
    def fget(self):
        if (hasattr(self, 'firstname')
            and hasattr(self, 'lastname')):
            return "%s %s" % (self.firstname, self.lastname)
        else:
            raise AttributeError("Need both first and last names")
    def fset(self, name):
        try:
            self.firstname, self.lastname = name.split(None, 1)
        except ValueError:
            self.firstname, self.lastname = name, ""
    def fdel(self):
        del self.firstname
        del self.lastname
    fullname = property(fget, fset, fdel, doc)
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-44



## Defining Properties Without Namespace Pollution

```
def Property(func)
    return property(**func()) # Think about this ...

class Person(object):
    @Property # Note non-standard decorator usage ...
    def fullname():
        doc = "The person's name"

        def fget(self):
            return "%s %s" % (self.first_name, self.last_name)

        def fset(self, name):
            self.first_name, self.last_name = name.split(None, 1)

        def fdel(self):
            del self.first_name
            del self.last_name

    return locals() # Cheesy, but it works
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-45



## Property Lookup

- ```
>>> dir(Person.fullname)
['__class__', '__delattr__', '__delete__', '__doc__',
 '__get__', '__getattr__', '__hash__', '__init__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__set__', '__setattr__', '__str__', 'fdel', 'fget',
 'fset']
```
- Properties must be in the *class's* `__dict__`, *not* the instance's:

```
>>> class Thing(object):
...     pass
>>> def getter(self):
...     return "This is me"
>>> thing = Thing()
>>> thing.who = property(getter)
>>> thing.who
<property object at 0x7ff30d74>
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-46



## Attribute Lookup: Old-Style vs New-Style (1)

```
>>> class Old: pass
...
>>> class New(object): pass
...
>>> Old.__dict__
{'__module__': '__main__', '__doc__': None}
>>> New.__dict__
<dictproxy object at 0x7ff614ac>
```

- The dictproxy object is *like* a dict
  - Can find out a *lot* about it with `dir` and `inspect.classify_class_attrs`
  - But not much point, since you can't create instances
    - It's been castrated, and has no `__new__` method
  - Internal use only
  - Designed to avoid too much tweaking of class `__dict__`s

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-47



## Attribute Lookup: Old-Style vs New-Style (2)

- Some simple implications

```
>>> class N(object): pass
...
>>> N.__init__
<slot wrapper '__init__' of 'object' objects>
>>> object.__dict__
<dictproxy object at 0x7ff6147c>
>>> object.__dict__['__init__']
<slot wrapper '__init__' of 'object' objects>
>>> object.__dict__['__init__'] = 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dictproxy' object does not support item assignment
>>> object.__init__ = 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't set attributes of built-in/extension type 'object'
>>>
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-48



## Attribute Lookup: Old-Style vs New-Style (3)

- There's actually a (relatively) simple reason for this:

```
>>> dir(object.__dict__['_init_'])
['_call__', '__class__', '__delattr__', '__doc__',
 '__get__', '__getattr__', '__hash__', '__init__',
 '__name__', '__new__', '__objclass__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__str__']
>>>
```

- **Object.\_\_init\_\_ is a *read-only property!***

- No `__set__` or `__delete__` methods
- You can, however, read its documentation
  - Not that it's much help ...

```
>>> object.__dict__['_init_'].__doc__ # object.__init__.__doc__
'x.__init__(...) initializes x; see x.__class__.__doc__ for
signature'
>>> object.__doc__
'The most base type'
>>>
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-49



## `__getattr__`, `__setattr__` and `__getattribute__`

- There are differences between new-style and old-style classes
- Implementing `__setattr__` allows you to use your own data store
- Classes have always used `__getattr__` if attribute access failed
  - Of course, you have to *define* `__getattr__`
  - Otherwise you get an immediate `AttributeError`
  - Looking for `__getattr__` and (if found) calling it is the final fallback
- New-style classes have altered the paradigm
  - And what the hell is `__getattribute__`?
  - Allows you to specify *all attribute lookup semantics*
  - Great from the point of view of control
  - But likely to run like a drain
    - Native implementation is in C
    - You alternative in Python *will* be slower

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-50



## Python 401: Lesson 5

# Metaclasses

```
    navbarstuff = models.TextField()
    pagctype = models.CharField(max_length=10)
    pageid = models.IntegerField()
class Meta:
    db_table = 'page'
class Admin:
    list_display = ('pagpath',)
    def __unicode__(self):
    if self.pagsecid:
        section = self.pagsecid
    else:
        section = "(None)"
```



## Python is A Dynamic Language

- Can modify classes dynamically

```
>>> def class_with_method(func):
...     class klass: pass
...     setattr(klass, func.__name__, func)
...     return klass
...
>>> def say_foo(self): print 'foo!'
...
>>> Foo = class_with_method(say_foo)
>>> foo = Foo()
>>> foo.say_foo()

foo!
```



## Python Is Fully Introspective

- **It has always revealed its inner workings**
  - Before 2.2 could not inherit from built-in types
- **2.2 introduced a new object model**
  - Retained “classic” classes for compatibility
    - They disappear from 3.0 onward
  - Made it easier (trivial!) to subclass built-ins
- **Much of the magic remains the same**
  - But there are some interesting differences

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-53



## 2.2 Object Model Changes

- **Builtins became subclassable**
  - All are subclasses of object, the ultimate ancestor
- **Creating an instance now calls class's `__new__()` method**
  - This is expected to return the new object
- **Class's `__init__()` method is also called**
  - If the new object is an instance of the class

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-55



## Factories Make Things



© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-56



## Python Factory Functions (1)

- **Function factory:**

```
>>> def adder(n):
...     def addN(x): # defines function
...         return x+n
...     return addN # returns it!
>>> f = adder(3)    # a function that adds 3
>>> f
<function addN at 0x7ff284fc>
>>> f(6)
9
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-57



## Python Factory Functions (2)

- **Class factory:**

```
>>> def Adder(n):
...     class addN(object):
...         def __call__(self, x):
...             return x+n
...     return addN
...
>>> c = Adder(3) # A class whose callable
                # instances add 3

>>> c
<class '__main__.addN'>
>>> f = c()
>>> f(6)
9
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-58





© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-59



## `type()` is a Class Factory

- Can call with three arguments to build a class

```
>>> X = type('X',(),{'foo':lambda self:'foo'})
>>> X
<class '__main__.X'>
>>> x = X()
>>> x
<__main__.X object at 0x7ff301ac>
>>> x.foo()
'foo'
>>>
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-60



## type is Also a Class

- So we can subclass it!

```
class Printable(type):  
    def whoami(cls):  
        print "I am a", cls.__name__
```

- `Printable.__call__ == type.__call__ [????]`  
— By inheritance

```
Foo = Printable('Foo', (), {})
```

```
Foo.whoami() # class method call ...
```

```
I am a Foo
```

© Copyright Holden Web LLC: All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-61



## In Fact, type is a Metaclass

- Which means that its instances are classes ...
- Which makes them callable ...
- And when they are called they create ... instances!
- Since calling a new-style class runs its `__new__()` and its `__init__()`
- There's a particular way to call the metaclass

© Copyright Holden Web LLC: All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-62



## Class Creation (1)

```
class klass(b1, b2, b3):  
    code of class body
```

- **Interpreter compiles the class body**
- **Then calls the metaclass with a defined argument set**
  - What does this do?
  - How is the metaclass identified?

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPy 401-63



## Class Creation (2)

- **metaclass(name, bases, namespace)**
  - Calls `metaclass.__new__`
    - Creates a metaclass instance (*i.e.* a new class)
  - Calls `metaclass.__init__`
    - Initializes the metaclass instance
- **name: Name of class being defined**
- **bases: tuple of base classes (for inheritance)**
- **namespace: mapping from class body**
  - Contains class attributes and methods

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPy 401-64



## So, What are Metaclasses Good For?

- **Why do we like to define our own classes?**
  - Control over *instance* behavior
- **So why would we want to define metaclasses?**
  - Control over *class* behavior
- **Suppose, for example, we wanted our classes to trace all their method calls ...**

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-65



## How to Trace Functions

- **Define a decorator!\***

```
def traced(f):
    def tracer(*args, **kw):
        print "*** %s(%s)" % (
            f.__name__,
            ", ".join(
                [repr(arg) for arg in args]+
                ["%s=%s" % (k, repr(v))
                 for (k, v) in kw.items()]
            ))
        return f(*args, **kw)
    return tracer
```

\* See Appendix A

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-66



## Tracing Example

```
from trace import traced
@traced
def myfun(a, b="hello"):
    return a*b

if __name__ == "__main__":
    print myfun(1)
    print myfun(2, b="banana")
    print myfun(a="apple", b=3)
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-67



## Output from Tracing Example

```
$ python tracexample.py
** myfun(1)
hello
** myfun(2, b='banana')
bananabanana
** myfun(a='apple', b=3)
appleappleapple
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-68



## After That Diversion ...

- We want our classes to automatically trace all calls to all their methods
- We could just decorate the methods as we declare them
  - But that's not very Pythonic ...
- But when the class is defined
  - The metaclass is called to create it
  - So let's have our metaclass do the work!

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-69



## The Metaclass

```
from trace import traced

class Tracer(type):
    def __new__(cls, name, bases, cdict):
        for n in cdict:
            if callable(cdict[n]) \
                and not n.startswith("__"):
                cdict[n] = traced(cdict[n])
        return type.__new__(cls, name, bases, cdict)
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-70



## The Test

```
class MyClass(object):
    __metaclass__ = Tracer
    def method1(self, a):
        print "Method 1:", a
    def method2(self, a, b=10):
        return "Method 2: %s" % (a*b)

mc = MyClass()
mc.method1("Hooray!")
print mc.method2("Easy", b=3)
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-71



## The Output

```
$ python meta.py
** method1(<__main__.MyClass object at
0x7ff2982c>, 'Hooray!')
Method 1: Hooray!
** method2(<__main__.MyClass object at
0x7ff2982c>, 'Easy', b=3)
Method 2: EasyEasyEasy
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-72



## And finally ...

- Setting the metaclass to Tracer automates method call tracing ...
- So a class whose *metaclass* is Tracer can be used as a base class for traced classes
- Because it will inherit its metaclass from the base class

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-73



Python 401: Lesson 6

## What Have We Learned?

```
    navbarstuff = models.TextField()
    pagctype = models.CharField(max_length=255)
    pagid = models.IntegerField()
    class Meta:
        db_table = 'page'
        admin:
            list_display = ('pagpath',
                'unicode_(self)',
                if self.pagsecid:
                    section = self.pagsecid
                else:
                    section = "(None)"
```

 **Holden  
Web**  
What Do You Need to Know Today?

## I've Tried to Do My Bit!

- Thanks for supporting the PSF and PyCon
- Hope you learned some new things about Python
- If so, please tell me what
  
- If you are here for the rest of PyCon ...
  - If you are new, enjoy it
    - It's my very favorite conference
    - *And* my baby
  - If you aren't new, help the newbies enjoy it

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-75



## Discussion



© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-76



## Python Decorators

```
gnavbarstuff = models.TextField(m
pagctype = models.CharField(m
pagid = models.IntegerField()
class Meta:
    db_table = 'page'
class Admin:
    list_display = ('pagpath'
    __unicode__(self):
    if self.pagsecid:
        section = self.pagsec
    else:
        section = "(None)"
```



### Why Isn't This in the Write-Up?

- **This additional material is to benefit those who attended the tutorial**
  - Supporting information for those not familiar with decorators
- **It will not be taught in the class**
- **But I hope it will nevertheless be interesting**



## Decorators

- `@d`  
`def f(...):`  
...  
is a simple piece of syntactic sugar
- **Meaning:**  
`def f(...):`  
...  
`f = d(f)`
- **Normally, though not inevitably, the decorator returns a function**
  - This wraps `f`'s functionality
  - Applying some "façade" to the standard function
  - Typically adding extra behaviors

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-79



## Simple Decorator Example

- **Suppose you want to see the arguments to certain function calls**

```
def tracing(f):
    def decorated(*arg, **kw):
        print "*** %s(%s)" % (
            f.__name__,
            ", ".join(
                [repr(a) for a in arg] +
                ["%s=%s" % (k, repr(v))
                 for (k, v) in kw.items()]
            ))
        return f(*arg, **kw)
    return decorated
```
- **This beats the lines you get from tracebacks**
  - You actually see the *values* of the arguments!

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-80



## Testing Results

- A simple test to demonstrate

```
if __name__ == "__main__":
    @tracing
    def somefunc(a, b, c=[]):
        pass
    for i in range(5):
        somefunc(i, i*i)
        somefunc(1, i, c=[0]*(i//2))
-----
$ python tracing.py
** somefunc(0, 0)
** somefunc(1, 0, c=[])
** somefunc(1, 1)
** somefunc(1, 1, c=[])
** somefunc(2, 4)
** somefunc(1, 2, c=[0])
** somefunc(3, 9)
** somefunc(1, 3, c=[0])
** somefunc(4, 16)
** somefunc(1, 4, c=[0, 0])
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-81



## Little Decorator Problem

- When you decorate a function you lose its `__name__` and `__doc__`

```
>>> @tracing
... def f(a, b):
...     "My happy little function"
...     return a*b
...
>>> f.__name__ # The name of the decorator
'decorated'
>>> f.__doc__  # The docstring of the decorator
>>>
```

- Fortunately the new `partial` module contains a solution

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-82



## functools.wraps

- This internal decorator transfers the wrapped function's characteristics to the wrapper:

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print 'Calling decorated function'
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print 'Called example function'
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-83



## functools.partial

- Allows you to partially specify the arguments of a function

```
>>> from functools import partial, wraps
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
>>> type(basetwo)
<type 'functools.partial'>
>>>
```

- Again, the function name and doc string will not be correct
- Can't use wraps as a decorator in this case
  - Don't have access to original function declarations
  - But could use it directly in code:  
`basetwo = wraps(int)(partial(int, base=2))`
  - Note carefully that `wraps(x)` returns a callable decorator

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-84



## Context Managers

```
paginatorstuff = models.TextField  
pagctype = models.CharField(m  
pagid = models.IntegerField()  
class Meta:  
    db_table = 'page'  
class Admin:  
    list_display = ('pagpath'  
    __unicode__(self):  
    if self.pagsecid:  
        section = self.pagsec  
    else:  
        section = "(None)"
```



## The with Statement and Context Managers

- **with** `expr [as variable]:` # introduced as a future feature in 2.5  
    `controlled-suite`
- **expr value should be a *context-manager* object**
  - Must have `__enter__()` and `__exit__()` methods
- **`expr.__enter__()` is called**
  - if `as` clause is present, call result is bound to `variable`
- **Controlled-suite is executed**
  - It may terminate normally
  - It might raise an exception
- **`expr.__exit__()` is then called**



## Unwrapping the Context

- For a normal exit
  - `expr.__exit__(None, None, None)` is called
- If controlled-suite raises an exception
  - `expr.__exit__(type, value, traceback)` is called
  - Best if that exception is re-raised
  - Swallowing exceptions silently is usually a bad idea
    - Though the `__exit__()` method can do so by returning `True`
    - `__exit__()` should *not* re-raise the exception
    - That will be done automatically when `__exit__()` returns

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-87



## Context Manager Example (1)

- Sample code to run database transactions
  - `__enter__()` returns a cursor that can be used to operate on the DB
  - `__close__()` commits if no error, else rolls back

```
class DBConnection:
    ... __init__() can create connection or use one passed in ...
    def __enter__(self):
        # Code to start a new transaction
        return self.conn.cursor()
    def __exit__(self, type, value, tb):
        if tb is None:
            # No exception, so commit
            self.conn.commit()
        else:
            # Exception occurred, so rollback.
            self.conn.rollback()
            # return False # if exception hiding required

with DBConnection(...) as cursor:
    ... use cursor to access database ...
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-88



## Python 2.5: @contextlib.contextmanager

- Intended for use with the `with` statement
- Wraps a generator function, which must implement precisely one `yield` expression
- When the generator yields, the `with` statement runs its controlled suite

```
from __future__ import with_statement # Only required for 2.5
from contextlib import contextmanager
@contextmanager
def tag(name):
    print "<%s>" % name
    yield
    print "</%s>" % name
>>> with tag("h1"):
...     print "foo"
...
<h1>
foo
</h1>
```

- This is *not* the recommended way to generate HTML ...

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-89



Python 401: Appendix C

## Python Unit Testing

```
    navbarstuff = models.TextField()
    pagctype = models.CharField(max_length=10)
    pagid = models.IntegerField()
class Meta:
    db_table = 'page'
class Admin:
    list_display = ('pagpath',
                    'pagctype',
                    'pagid')
    def __unicode__(self):
        if self.pagsecid:
            section = self.pagsecid
        else:
            section = "(None)"
```

 **Holden  
Web**  
What Do You Need to Know Today?

## Why Isn't This in the Write-Up?

- **This additional material is to benefit those who attended the tutorial**
- **It will probably not be taught in the class**
- **But we hope it will nevertheless be interesting**
- **Everyone should know a little about testing ...**
  - Test-driven development is becoming common in open source

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPy 401-91



## Unit Testing Framework

- **Unit tests are intended to be “go/no-go” tests**
  - They are *not* integration or system tests
  - They test the *components* for correct operation
  - They should ideally be *short*
- **Testing is important for a number of reasons**
  - The tests are, in test-driven development, the specifications
    - Written before the code, which is then written to pass them
  - They validate refactorings and bug fixes
    - So can catch bad fix injections before they take root
  - They allow other people to make changes to the code
  - They improve code quality
- **It's easy to start, and you'll never regret it**
  - People talk about becoming “dot addicted”

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPy 401-92



## unittest Library Module (“PyUnit”)

- Modeled after the Java “JUnit” testing framework
- The building block is the *test case*
  - Single scenario that must be set up and checked for correctness
  - Built by subclassing `unittest.TestCase`
    - Simplest way is to override the `runTest()` method
  - Can use `TestCase` methods to assert required conditions
    - Test succeeds if all assertions succeed
- Simple example from Python docs (`test1.py`):

```
import unittest
from widget import Widget

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def runTest(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50),
                          'incorrect default size')
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-93



## Initial Widget Implementation

- Just enough to see the test running
  - Makes imports succeed – nothing else

```
#
# widget1.py: the simplest thing that can possibly error
#
class Widget:
    pass

$ python test1.py
E
=====
ERROR: runTest (__main__.DefaultWidgetSizeTestCase)
-----
Traceback (most recent call last):
  File "test1.py", line 6, in runTest
    widget = Widget('The widget')
TypeError: this constructor takes no arguments
-----

Ran 1 test in 0.001s

FAILED (errors=1)
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-94



## Test Results

- **Tests can have three outcomes**
  - Succeed – clearly this is the one we want to see
  - Fail – “Oh bugger, back to the drawing board”
  - Error – The test didn’t conclude naturally for some reason
- **In this case we got an error**
  - Our object’s `__init__()` method signature didn’t match the call
  - We may have to re-write our tests to overcome errors
    - Not necessary in this case as it’s a clear problem with the Widget
    - So let’s add a compatible method to the widget
- **If at first you don’t succeed ...**

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPy 401-95



## First Widget Revision

- **Purpose of this revision was simply to lose the error**
  - Not necessarily expecting success

```
$ cat widget2.py
#
# widget2.py: the next simplest thing that can possibly error
#
class Widget:
    def __init__(self, title):
        self.title = title

$ python test1.py
E
=====
ERROR: runTest (__main__.DefaultWidgetSizeTestCase)
-----
Traceback (most recent call last):
  File "test1.py", line 7, in runTest
    self.assertEqual(widget.size(), (50, 50), 'incorrect default size')
AttributeError: Widget instance has no attribute 'size'
-----
Ran 1 test in 0.001s

FAILED (errors=1)
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPy 401-96



## Rats, We Got Errors Again

- **But this is progress!**
  - Now we know something else to change
  - There's a `size()` method being called, dagnabbit
    - For now we'll just provide a stub
  - We'll also have to default the size
    - May as well change `__init__()`

```
#
# widget3.py: the simplest thing that can possibly fail
#
class Widget:
    def __init__(self, title, size=(50, 50)):
        self.title = title
        self._size = size
    def size(self):
        pass
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-97



## Our First Real Failure!

```
python test1.py
F
=====
FAIL: runTest (__main__.DefaultWidgetSizeTestCase)
-----
Traceback (most recent call last):
  File "test1.py", line 7, in runTest
    self.assertEqual(widget.size(), (50, 50), 'incorrect default size')
AssertionError: incorrect default size
-----

Ran 1 test in 0.001s

FAILED (failures=1)
```

- **YAAAAAYYYY!!!!**
  - Our test ran to completion for the first time
  - Who'd have thought a *failure* could be so exciting?

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-98



## And Finally ...

- **So now we know the test fails**
  - The `size()` method isn't returning the size ...
  - Which we might have anticipated
    - But didn't need to, because our test pretty much told us what to do

- **One (final?) revision?**

```
$ cat widget.py
#
# widget4.py: the simplest thing that can possibly work
#
class Widget:
    def __init__(self, title, size=(50, 50)):
        self.title = title
        self._size = size
    def size(self):
        return self._size

$ python test1.py
.
-----
Ran 1 test in 0.000s

OK
```

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-99



## Isn't This a Lot of Work for One Test?

- **Not really – and think of the advantages**
  - We have a known-working primitive `Widget` on which to build
    - Or would you rather crank our something that doesn't work?
    - That would have been *much* quicker
    - In the beginning, at least
  - We are starting to understand the significance of unit testing
- **Doesn't this create an awful lot of files?**
  - It would if we had to build all our tests this way
  - But fortunately, the `unittest` framework allows you to bundle them
- **Rather than overriding `RunTest()`, define multiple `testXXX()` methods**
  - The standard `runTest()` will run each one
  - Printing consolidated results
  - It's all rather friendly, really ...

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-100



## What If a Test Needs Fixtures Set Up?

- The `TestCase` class will run a `setUp` method before the test
  - It will also run a `tearDown` method afterwards
  - Whether the test succeeds or not
- If many tests have common `setUp` and `tearDown` requirements?
  - Define a `TestCase` subclass with those methods installed
  - Then let your real tests subclass from that

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-101



## A (Slightly) More Comprehensive Set of Tests

- Note that we write these tests *before* we make any changes to `widget.py`
  - Because they are going to tell us what changes we need to make!

```
$ cat test2.py
import unittest
from widget import Widget

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
        self.widget = None

    def testDefaultSize(self):
        self.failUnless(self.widget.size() == (50,50),
            'incorrect default size')

    def testResize(self):
        self.widget.resize(100,150)
        self.failUnless(self.widget.size() == (100,150),
            'wrong size after resize')

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

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-102



## To the Tests, To the Tests!

```
$ python test2.py
EEE
=====
ERROR: testDefaultSize (__main__.WidgetTestCase)
-----
Traceback (most recent call last):
  File "test2.py", line 9, in tearDown
    self.widget.dispose()
AttributeError: Widget instance has no attribute 'dispose'
=====
ERROR: testResize (__main__.WidgetTestCase)
-----
Traceback (most recent call last):
  File "test2.py", line 17, in testResize
    self.widget.resize(100,150)
AttributeError: Widget instance has no attribute 'resize'
=====
ERROR: testResize (__main__.WidgetTestCase)
-----
Traceback (most recent call last):
  File "test2.py", line 9, in tearDown
    self.widget.dispose()
AttributeError: Widget instance has no attribute 'dispose'
-----
Ran 2 tests in 0.002s
```

- Are we downhearted?
  - No – the changes are screaming out at us!

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-103



## Let's Add the `dispose()` and `resize()` methods

- To start with, these can be quite simple
  - That's called British understatement ☺

```
$ cat widget5.py
#
# widget5.py: A more complex but still working widget (we hope)
#
class Widget:
    def __init__(self, title, size=(50, 50)):
        self.title = title
        self._size = size
    def size(self):
        return self._size
    def resize(self, height, width):
        self._size = (height, width)
    def dispose(self):
        pass

$ python test2.py
..
-----
Ran 2 tests in 0.000s
```

OK

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-104



## Higher-Level Test Groupings

- **A `unittest.TestCase` can house a number of individual tests**
  - With common `setUp` and `tearDown` requirements
  - You can create an instance to run just one of the tests
    - Simply give the test method name as a creator argument
    - *E.g.* `resizeTestCase = WidgetTestCase('testResize')`
- **TestCase instances can be grouped together in a `unittest.TestSuite`**
  - Often in functions for testing convenience
  - In fact, you can also add a `TestSuite` to a `TestSuite`
- **There's more**
  - But I hope by now you will be encouraged to read the documentation

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-105



## Unit Testing At White Wolf *(Thanks to Richard Tew)*

- **Based on the standard `unittest` framework**
- **Because code can be edited and then reloaded on the fly**
  - Unit tests are applied whenever a module in the test client is reloaded
  - Failures are automatically reported
    - To the client or server application
    - To the logger
- **The environment is complex**
  - Maybe you are testing a game service
  - It might depend on other services
    - Normally referenced through service instance variables
    - Automatically generated
- **Tests are therefore enabled by the use of *mock objects***
  - Emulate the behavior of components missing from the test environment

© Copyright Holden Web LLC. All rights reserved. Not to be reproduced without prior written consent.

HWPY 401-106



## Structure of a White Wolf Test Case

- To show the skeleton outlines

```
import mock, unittest

class MyTestCase(unittest.TestCase):
    def setUp(self):
        # Mock out all modules and builtins (or something).
        mock.Setup(self, globals())
        # Create a make-do instance of the service this file defines.
        svc = MyService()
        # As we are manually creating the service,
        # we need to install its dependencies by
        # hand. So mock out them to do nothing.
        svc.otherService = mock.Mock("otherService")
        # Mock a service this service accesses as "built in"
        self.smRef = mock.Mock("sm", insertAsGlobal=True)

    def tearDown(self):
        mock.TearDown(self)
```