Let's begin!

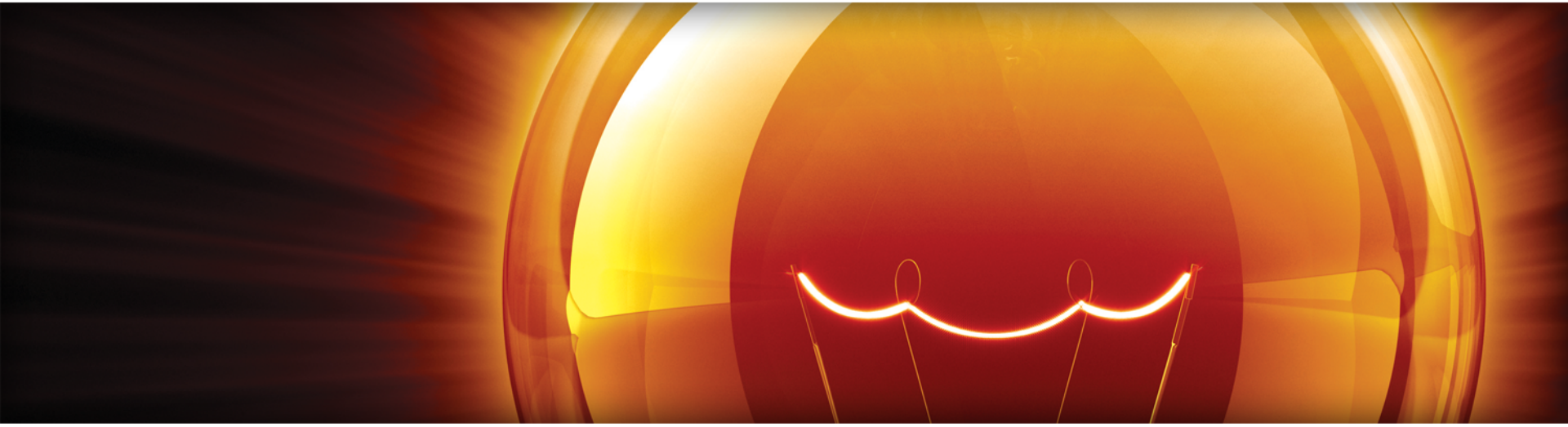
Develop Intelligence

- install Python 3
  - go to https://www.python.org/downloads/
  - Macs should have 2.7.10 installed by default, but it's OK to have Python 2 and 3 co-resident
- install PyCharm (optional)
  - Awesome IDE for Python
  - https://www.jetbrains.com/pycharm/
- follow along with all examples using Python shell, IDLE, PyCharm (or your favorite IDE)

The Zen of Python

- Along the way we'll visit some of the maxims from The Zen of Python
- `import this`
  - type that into your Python interpreter
  - …and let's take a look at the Python code
  - `this.__file__`

Python Syntax Review



Learning Solutions to Attract, Retain, and Grow your top technical talent.

Syntax Review

- dynamic typing, no declarations

```
>>> x = 1
>>> x = 'hello'
>>> x
'hello'
```

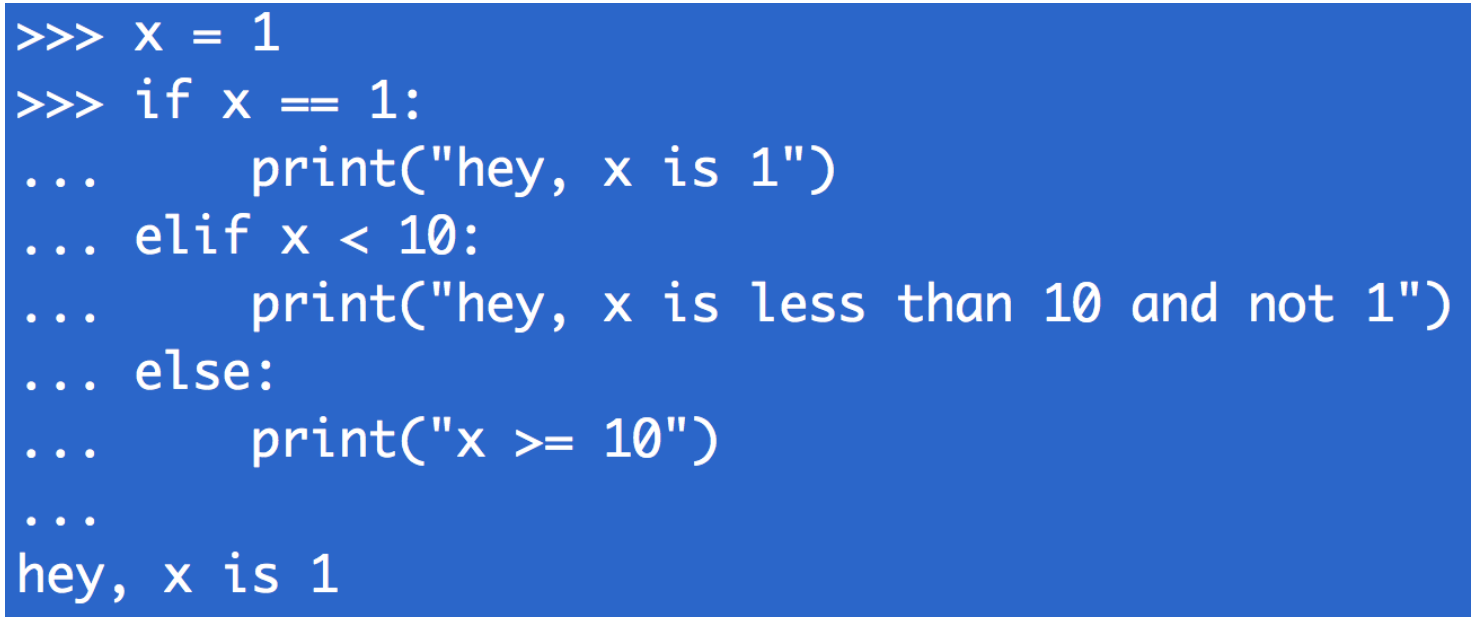
- …but strongly typed

```
>>> y = x + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> def func(arg):
...     return arg + 1
...
>>> func(2)
3
>>> func("hi")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in func
TypeError: Can't convert 'int' object to str implicitly
```

Syntax Review: indentation

- colons and indentation delineate blocks
  - no braces! (>>> `from __future__ import braces`)
  - __future__ modules gives you access to Python 3 features in Python 2

Syntax Review: if/elif/else



```
>>> x = 1
>>> if x == 1:
...     print("hey, x is 1")
... elif x < 10:
...     print("hey, x is less than 10 and not 1")
... else:
...     print("x >= 10")
...
hey, x is 1
```

- no parens needed around expression which is being checked
- remember that `print()` is a function in Python 3, so parens are required

Syntax Review: functions


```
>>> def simpfunc(x):
...     if x == 1:
...         print("hey, x is 1")
...     elif x < 10:
...         print("hey, x is less than 10 and not 1")
...     else:
...         print("x >= 10")
...
>>> simpfunc(1)
hey, x is 1
>>> simpfunc(5)
hey, x is less than 10 and not 1
>>> simpfunc(15)
x >= 10
>>> simpfunc(-3)
hey, x is less than 10 and not 1
```

wrapped preceding if statement in a function which does the printing but does not return a value…or does it?

testing the function… always a good idea!

Syntax Review: for loops

- for loops in Python are typically about iterating through a sequence of some sort (`list`, `set`, `dict`, etc.)

```
>>> for s in range(0, 10):
...     print(s)
...
0
1
2
3
4
5
6
7
8
9
```

```
>>> mylist = "small medium large".split()
>>> for size in mylist:
...     print(size)
...
small
medium
large
```

Syntax Review: scope


```
def outer():
    '''the next line gives us access to global x'''
    global x
    print("in outer(), global x =", x)
    x = 1

    def inner(x):
        '''this is NOT the global x!'''
        print("in inner(), local/param x =", x)
        x = 2
        print("in inner(), local/param x =", x)

    print("before inner(), x =", x)
    inner(x)
    print("after inner(), x =", x)

x = 0
print("at program start, global x is", x)
outer()
print("after calling outer(), global x is", x)
```

`global` keyword let us access to global vars inside a function

locally defined vars which have the same name as a var in an enclosing scope will hide the outer variable

which x is this?

Syntax Review: modules


```
public_data = "public stuff!"
_private_data = "private stuff!"

print("in mymodule")
```

private data is prefaced with an underscore (_)

```
Python 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015, 11:00:19)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__p
ackage__', '__spec__']
>>> from mymodule import *
in mymodule
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__p
ackage__', '__spec__', 'public_data']
>>> public_data
'public stuff!'
>>> _private_data
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_private_data' is not defined
```

when we `import` using the `from… *` syntax, all public data is added to our namespace

but not the private data

Syntax Review: modules (cont'd)

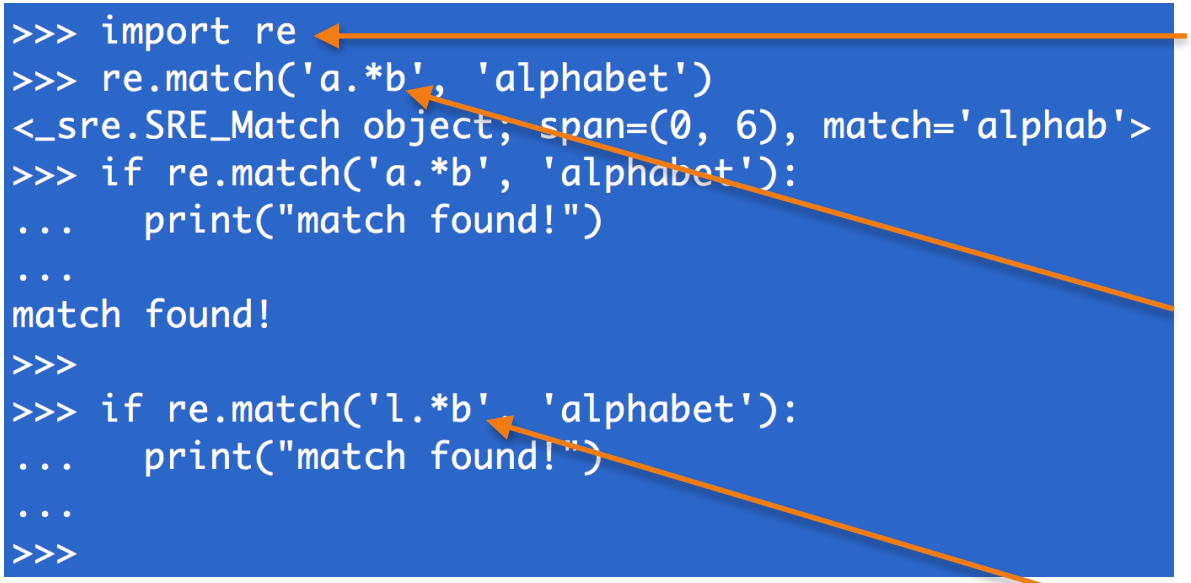
Develop Intelligence

- not the case if we use the `import module` syntax!

```
Python 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015, 11:00:19)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__p
ackage__', '__spec__']
>>> import mymodule
in mymodule
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__p
ackage__', '__spec__', 'mymodule']
>>> mymodule.public_data
'public stuff!'
>>> mymodule._private_data
'private stuff!'
```

private data is accessible

Syntax Review: regex



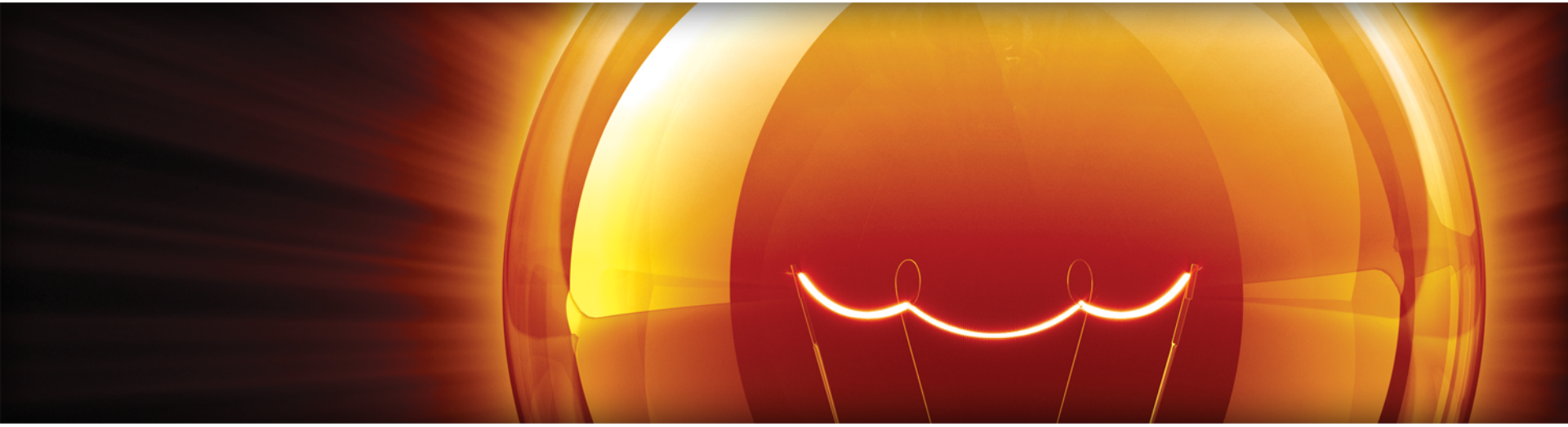
Syntax Review: regex (cont'd)


```
>>> import re
>>> o = re.search('l.*e', 'alphabet')
>>> o.re
re.compile('l.*e')
>>> o.re.pattern
'l.*e'
>>> o.string
'alphabet'
>>> o.start()
1
>>> o.end()
7
>>> o.string[o.start():o.end()]
'lphabe'
```

`re.match()` anchors the search to the beginning of the string, where `re.search()` will find a pattern anywhere

both functions return a `Match` object which contains, among other things, the start and end positions in the string where the match occurred

Python Datatype Overview



Learning Solutions to Attract, Retain, and Grow your top technical talent.

Datatype Overview

Develop Intelligence

- string
  - single or double quotes
  - if string contains one kind of quote, use the other to contain it rather than using a backslash
    - s = "This string isn't a problem quote"
    - s = 'This string is a “good” example'
    - ~~s = "This string is \”more difficult\” to read"~~
- list = ordered, comma-separated values in []’s
  - types can be mixed, but are intended to be homogeneous

```
>>> years = [ 1215, 1620, 1812, 1941 ]
>>> weird_list = [ 1, 2, 'three', (4, 5), True ]
>>> print(years[1], weird_list[3])
1620 (4, 5)
```

Datatype Overview: tuple

Develop Intelligence

- immutable list, but they imply some structure

```
>>> t = ("Gutzon Borglum", "Idaho", 1867)
>>> t[1]
'Idaho'
>>> t[1] = 'Montana'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

- …can contain mutable objects

```
>>> t = (1, 2, [3, 4])
>>>
>>> t[2][1] = 5
>>> t
(1, 2, [3, 5])
```

- singleton

```
>>> singleton = (1)
>>> singleton
1
>>> singleton = (1,)
>>> singleton
(1,)
```

Datatype Overview: tuple (cont'd)

Develop Intelligence

- when to use list vs. tuple?
  - tuples are heterogeneous data structures–structure vs. order
  - not just "constant lists" (see http://jtauber.com/blog/2006/04/15/python_tuples_are_not_just_constant_lists)
  - lists, on the other hand, are ordered sequences of homogeneous values (they don't have to be homogeneous, but typically are)

Datatype Overview: set

- unordered collection, no duplicates

```
>>> names = [ 'Anne', 'Betty', 'Cathy' ]
>>> myset = set(names)
>>> myset
{'Cathy', 'Anne', 'Betty'}
>>> myset.add('Lola')
>>> myset
{'Cathy', 'Anne', 'Lola', 'Betty'}
>>> myset.add('Anne')
>>> myset
{'Cathy', 'Anne', 'Lola', 'Betty'}
```

- set operations
  - union (|), intersection (&), difference (-), symmetric difference/XOR (^)

Datatype Overview: dict

Develop Intelligence

- a Python dictionary is what we call an "associative array" in other languages

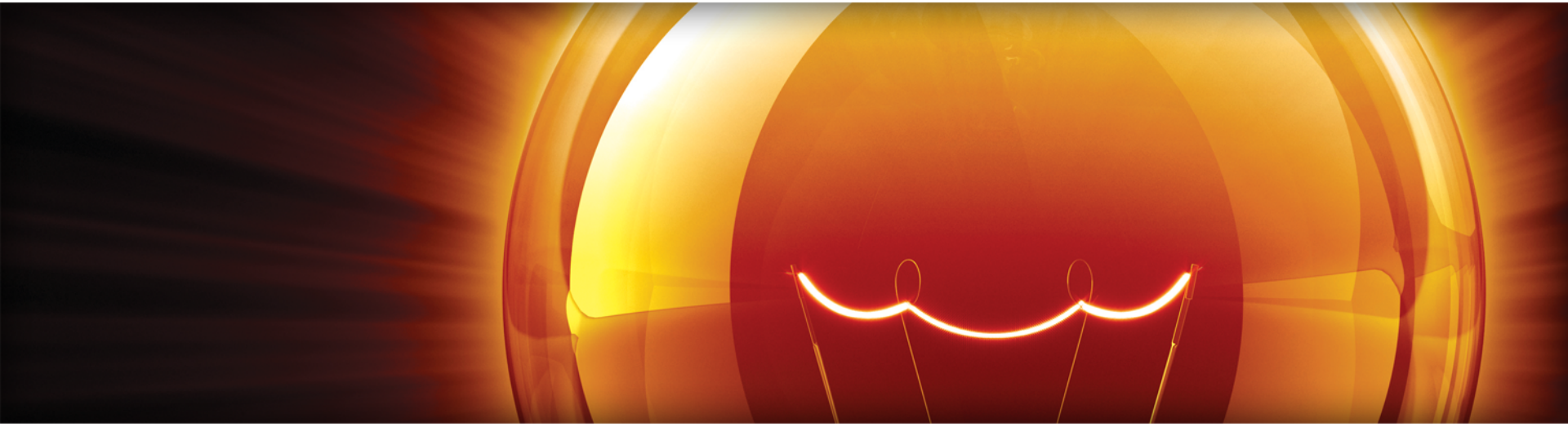
```
>>> mydict = { 'red' : 0, 'blue' : 1, 'green' : 2 }
>>> print(mydict['blue'])
1
>>> mydict['green'] = 5
>>> mydict
{'blue': 1, 'red': 0, 'green': 5}
```

```
>>> mydict = {}
>>> mydict
{}
>>> mydict['small'] = 0
>>> mydict['medium'] = 32
>>> mydict['large'] = 64
>>> mydict
{'medium': 32, 'large': 64, 'small': 0}
```

Datatype Overview: dict


```
>>> mydict.keys()
dict_keys(['medium', 'large', 'small'])
>>> mydict.values()
dict_values([32, 64, 0])
>>>
>>> for k, v in enumerate(mydict):
...     print(k, "=>", v)
...
0 => medium
1 => large
2 => small
```

Advanced Data Types: Ordered Dictionaries



Learning Solutions to Attract, Retain, and Grow your top technical talent.

Ordered Dictionaries

Develop Intelligence

- dictionaries, like sets, are unordered collections

```
>>> d = {}
>>> d['a'] = 3
>>> d['b'] = 6
>>> d['c'] = 0
>>> d
{'b': 6, 'c': 0, 'a': 3}
>>> d.keys()
dict_keys(['b', 'c', 'a'])
>>>
>>> d = { 'a': 3, 'b': 6, 'c': 0 }
>>> d
{'b': 6, 'c': 0, 'a': 3}
>>> d.keys()
dict_keys(['b', 'c', 'a'])
```

- Most of the time, what we're looking for is a way to map keys to associated values, so the ordering of the keys is irrelevant
- what if we want to iterate over the keys in a reliable manner?

Ordered Dictionaries

Develop Intelligence

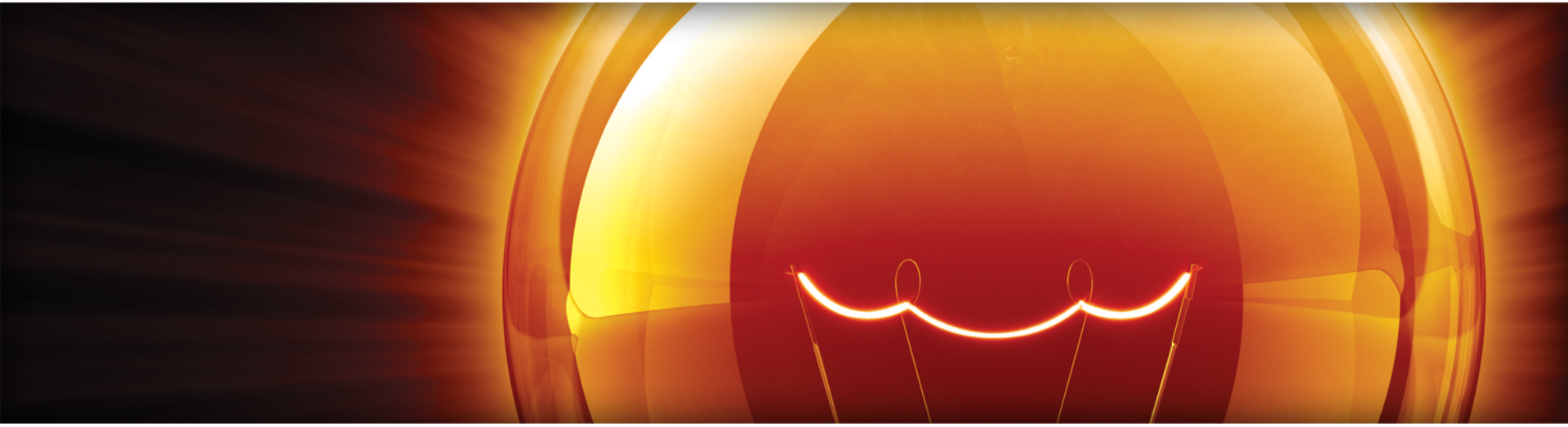
- use `OrderedDict`, found in the `collections` module

```
>>> from collections import OrderedDict
>>> d = OrderedDict()
>>> d['a'] = 3
>>> d['b'] = 6
>>> d['c'] = 0
>>> d
OrderedDict([('a', 3), ('b', 6), ('c', 0)])
>>> d.keys()
odict_keys(['a', 'b', 'c'])
>>> for key, val in d.items():
...     print("{} => {}".format(key, val))
...
a => 3
b => 6
c => 0
```

Lab 1: Ordered Dictionaries

- For two dictionaries to be equal, their keys and values must be equal, but the ordering of the keys (and corresponding values) is irrelevant. With `OrderedDicts`, they must have the same keys and the keys must be in the same order.
- Write Python code to show that this is the case.
  - Your code must show that two dictionaries which have the same key/values, but inserted in different orders, are equal.
  - Your code must also show that two OrderedDicts which have the same key/values, but inserted in different orders, are not equal.
  - Finally, show that two OrderedDicts which have the same key/value, inserted in the same order, are in fact equal.

Advanced Data Types: Default Dictionaries



Learning Solutions to Attract, Retain, and Grow your top technical talent.

Default Dictionaries

Develop Intelligence

- What if we need a default value for any key which does not already exist in the dict?
  - we could catch `KeyError`
  - we could use the `get()` method instead of `[]`

```
def count_words(text, debug=False):
    """Returns a dictionary of the words and how many times the word
    appeard in the string passed in."""
    count = {}
    for word in text.split():
        if debug:
            print("incrementing count for '{}'".format(word))
        try:
            count[word] += 1
        except:
            if debug:
                print("'{}' not in dict, starting count at 1".format(word))
            count[word] = 1
    return count
```

Default Dictionaries (cont'd)

Develop Intelligence

- or, we could use the `get()` method instead of `[]`

```
def count_words(text):
    """Returns a dictionary of the words and how many times the word
    appeard in the string passed in."""
    count = {}
    for word in text.split():
        '''ensure we always have a number by calling get with a second
        argument of 0, which causes the return value to be 0 if key is
        not already in the dict'''
        current = count.get(word, 0)
        count[word] = current + 1
    return count
```

Default Dictionaries (cont'd)

- or…use defaultdict

```
from collections import defaultdict

def count_words(text):
    count = defaultdict(int)
    for word in text.split():
        count[word] += 1
    return count
```

- `count` is a `defaultdict`, and the passed argument (`int` in this case) dictates what the default value will be
  - `int` = 0, `str` = "", `list` = []

Lab: Default Dictionaries

1. Write Python code which prompts the user for the name of a file, then opens the file and reads in its contents, counting the occurrences of each word using a `defaultdict`. Once the file has been read, print out the list of words in the file, sorted by frequency (i.e., most frequent to least frequent).
2. If you have time, deal with punctuation, e.g., "said", "said," and "said." should all be considered "said"

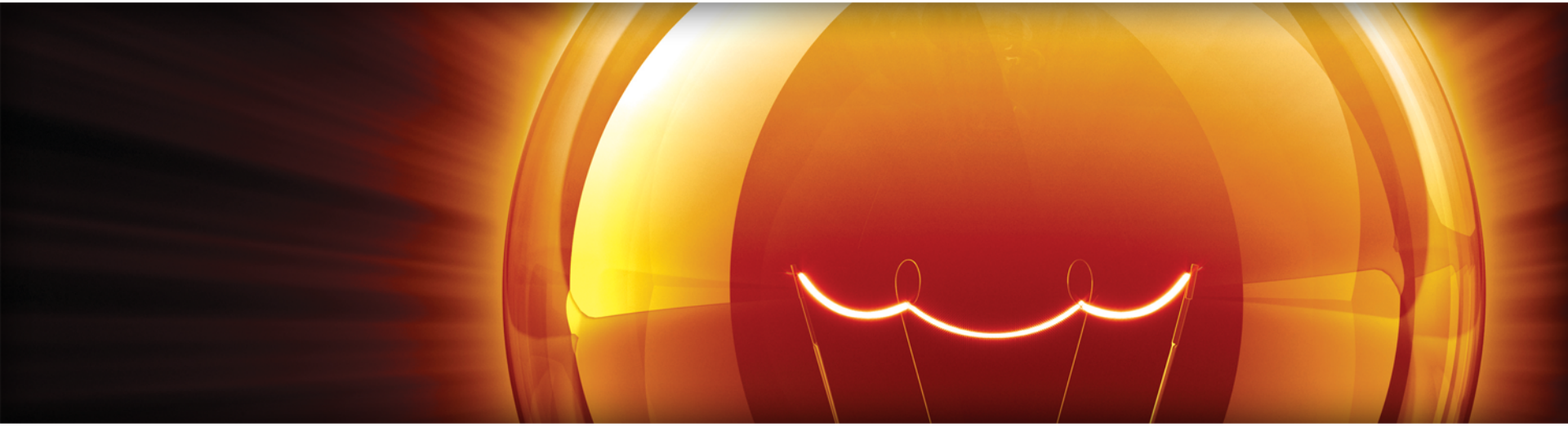
Here's some code which lets you read from a file. We will ignore error checking for the time being

```
>>> f = open('testfile', 'r')
>>> data = f.read()
>>> f.close()
>>>
>>> data
'Now is the time for all\ngood people to come to \nthe aid of their country\n'
```

Lab 2: Ordered Dictionaries

- Write code to open a file, where each line contains a first name, last name, and employee number. Store these data in an ordered dictionary. Once all the data have been read, print them out to the screen.

Advanced Data Types: deque



Learning Solutions to Attract, Retain, and Grow your top technical talent.

deque

- double ended queue (should be pronounced "deh-queue", but it's pronounced "deck")

```
>>> from collections import deque
>>> dq = deque(range(10), maxlen=10)
>>> dq
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.rotate(3)
>>> dq
deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6], maxlen=10)
>>> dq.rotate(-4)
>>> dq
deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], maxlen=10)
>>> dq.appendleft('a')
>>> dq
deque(['a', 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.extend('bcd')
>>> dq
deque([3, 4, 5, 6, 7, 8, 9, 'b', 'c', 'd'], maxlen=10)
>>> dq.extendleft((-1, -2, -3))
>>> dq
deque([-3, -2, -1, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
```

- `maxlen` is optional, otherwise unbounded
- `rotate(+n)` takes items from right and prepends to left of deck, vice versa for `(-n)`
- appending to full deck discard item(s) from other end
- `extendleft(iter)` appends successive items of iter to left of deque, so final position of items is reversed

deque (cont'd)

```
>>> dq
deque([-3, -2, -1, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.pop()
9
>>> dq.popleft()
-3
>>> dq
deque([-2, -1, 3, 4, 5, 6, 7, 8], maxlen=10)
>>> dq.remove(3)
>>> dq
deque([-2, -1, 4, 5, 6, 7, 8], maxlen=10)
>>> dq.reverse()
>>> dq
deque([8, 7, 6, 5, 4, -1, -2], maxlen=10)
>>> dq.append(0)
>>> dq.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'collections.deque' object has no attribute
 'sort'
>>>
>>> l = list(dq)
>>> l.sort()
>>> l
[-2, -1, 0, 4, 5, 6, 7, 8]
```

- `pop()`, `remove()`, `reverse()` work the same as they do on lists
- `popleft()` and `rotate()` (from previous slide) are specific to `deque`
- note that `sort()`-ing doesn't make sense, since this is a double-ended queue, but you can of course dump the items into a `list` and sort that

Lab 1: deque

Develop Intelligence

1. Write a function that takes a list of words and inserts them into a `deque` such that all the words that begin with vowels are on the right side of the deque, and all the words that begin with consonants are on the left side. There should be a separator between the words list in the deque, e.g.,

```
deque(['zoo', 'books', 'beta', '|', 'alpha', 'enter', 'oleander'])
```

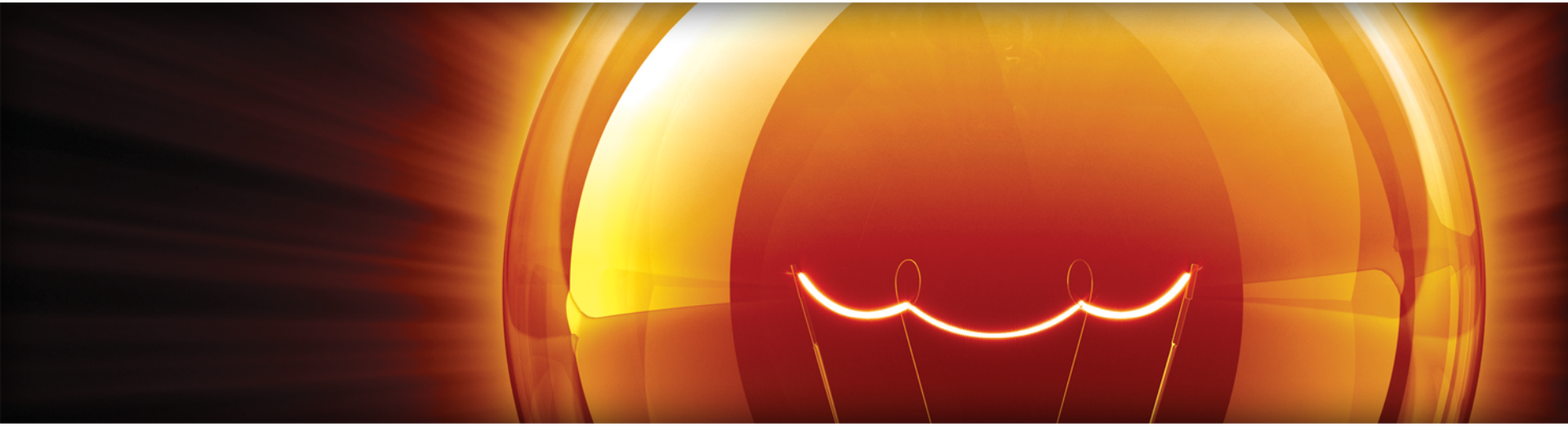
2. If you encounter a word in all upper case letters, you should remove a word from each end of the `deque`, reverse the `deque`, and replace the separator with the upper case word. So if you started with the `deque` above and then found 'HELLO', the deque should be changed to:

```
deque(['enter', 'alpha', 'HELLO', 'beta', 'books'])
```

Lab 2: deque

1. Create a stack class using a `deque`. Your class should implement the following methods: push (adds an element to top of the stack), pop (removes the top element), top (returns the top element but does not remove it), size (returns the number of items on the stack), and aslist (returns the stack as a list).
2. Add a method randomize, which jumbles the stack in random order (hint: check out the `random` module)
3. Add a sort method (as with #2, this is not something you would normally do with a stack)

Advanced Data Types: Named Tuples



Learning Solutions to Attract, Retain, and Grow your top technical talent.

Named Tuples

Develop Intelligence

- tuples are quite handy, but they are missing a key feature when using them as records–sometimes we want to name the fields
  - more efficient (i.e., less memory) than dictionaries because instances don't need to contain the keys themselves, as dictionaries do, just the values
- available from the `collections` module, `namedtuple()` returns not an individual object but a new class, customized for the given names

```
>>> import collections
>>> from collections import namedtuple
>>> Point = namedtuple('Point', 'x y')
>>> point1 = Point(1, 3)
>>> point1
Point(x=1, y=3)
>>> point2 = Point(-3, -2)
>>> point2
Point(x=-3, y=-2)
>>> point1[0], point2[1]
(1, -2)
```

first argument is the name of the tuple class itself; second argument is attribute names as an iterable of strings or a single space/comma-delimited string

Named Tuples: Cities Example


```
>>> from collections import namedtuple
>>>
>>> City = namedtuple('City', 'name country population coordinates')
>>> tokyo = City('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
>>> tokyo
City(name='Tokyo', country='JP', population=36.933, coordinates=(35.689722, 139.691667))
>>> tokyo.population
36.933
>>> tokyo.coordinates
(35.689722, 139.691667)
>>> tokyo[1]
'JP'
```

- `coordinates` is itself a tuple of latitude and longitude
- again, we can access fields by name or position

Named Tuples: Cities Example (cont'd)

Develop Intelligence

```
>>> City._fields
('name', 'country', 'population', 'coordinates')
>>> LatLong = namedtuple('LatLong', 'lat long')
>>> delhi_data = ('Delhi NCR', 'IN', 21.935, LatLong(28.613889, 77.208889))
>>> delhi = City._make(delhi_data)
>>> delhi._asdict()
OrderedDict([('name', 'Delhi NCR'), ('country', 'IN'), ('population', 21.935),
('coordinates', LatLong(lat=28.613889, long=77.208889))])
>>> for key, value in delhi._asdict().items():
...   print(key + ':', value)
...
name: Delhi NCR
country: IN
population: 21.935
coordinates: LatLong(lat=28.613889, long=77.208889)
```

- `_fields` is a tuple containing the field names
- `_make()` allows you to instantiate a named from an iterable (same as `City(*delhi_data)`)
- `_asdict()` returns a `collections.OrderedDict` built from the named tuple instance

Lab: Named Tuples

Create a named tuple called 'Card' (representing a playing card) which has two fields, rank and suit

Create a list of Cards, which, when initialized, contains all 52 cards in a deck. In other words, the list (or deck) should contain

```
[Card(rank=2, suit='spades'), Card(rank=3,
suit='spades'), Card(rank=4, suit='spades'),
Card(rank=5, suit='spades'), …
Card(rank='J', suit='hearts'), Card(rank='Q',
suit='hearts'), Card(rank='K', suit='hearts'),
Card(rank='A', suit='hearts')]
```

Named Tuples: Cards Example

Develop Intelligence

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

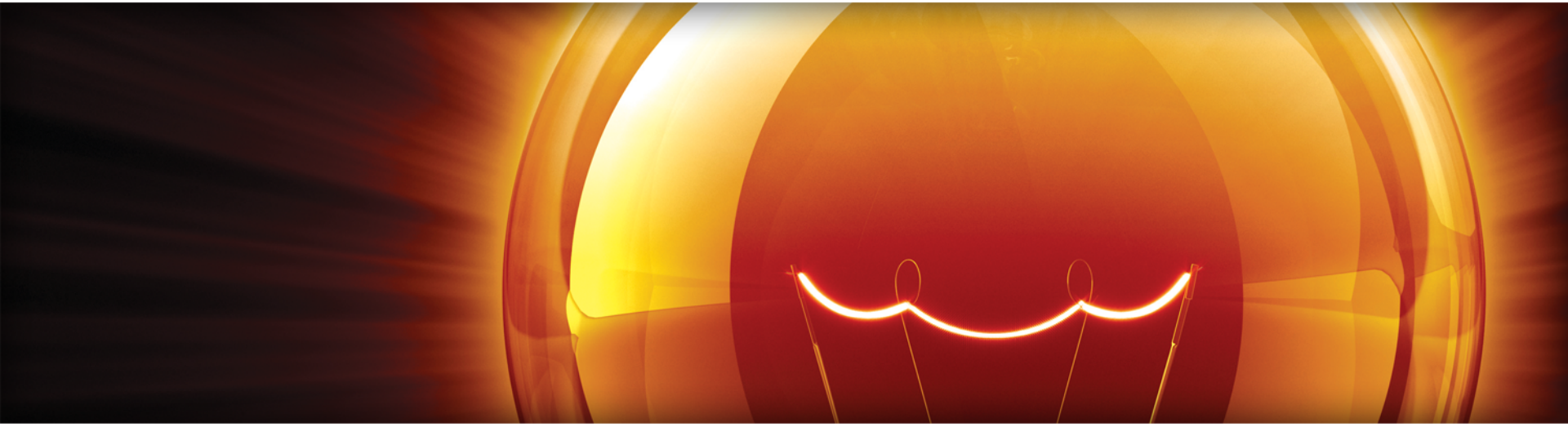
class DeckOfCards:
    ranks = []
    for n in range(2, 11):
        ranks.append(n)
    for p in 'JQKA':
        ranks.append(p)
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = []
        for suit in self.suits:
            for rank in self.ranks:
                self._cards.append(Card(rank, suit))

deck = DeckOfCards()
print(deck.ranks)
print(deck.suits)
```

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 'J', 'Q', 'K', 'A']
['spades', 'diamonds', 'clubs', 'hearts']
```

Advanced Data Types: Counters



Learning Solutions to Attract, Retain, and Grow your top technical talent.

Counters

- `dict` subclass for counting things
- unordered collection where the things being counted are `dict` keys and the counts are `dict` values
- here are some ways to initialize a counter…

```
>>> from collections import Counter
>>> c = Counter()
>>> c
Counter()
>>> c = Counter('Antidisestablishmentarianism')
>>> c
Counter({'i': 5, 's': 4, 'n': 3, 't': 3, 'a': 3, 'e': 2,
'm': 2, 'A': 1, 'l': 1, 'h': 1, 'r': 1, 'b': 1, 'd': 1})
>>> c = Counter({'red' : 5, 'blue' : -1})
>>> c
Counter({'red': 5, 'blue': -1})
>>> c = Counter(foo = 1, bar = 3)
>>> c
Counter({'bar': 3, 'foo': 1})
```

Counters (cont'd)

Develop Intelligence

```
>>> c = Counter(red=6, blue=5, green=3, pink=1, yellow=-3)
>>> c.elements()
<itertools.chain object at 0x1029e9518>
>>> list(c.elements())
['pink', 'green', 'green', 'green', 'blue', 'blue', 'blue', 'blue', 'blue', 'red',
 'red', 'red', 'red', 'red', 'red']
>>> c.most_common(3)
[('red', 6), ('blue', 5), ('green', 3)]
>>> d = Counter(red=3, blue=5, green=2, pink=0)
>>> c.subtract(d)
>>> c
Counter({'red': 3, 'pink': 1, 'green': 1, 'blue': 0, 'yellow': -3})
>>> c.items()
dict_items([('pink', 1), ('yellow', -3), ('green', 1), ('blue', 0), ('red', 3)])
>>> +c
Counter({'red': 3, 'green': 1, 'pink': 1})
>>> c
Counter({'red': 3, 'pink': 1, 'green': 1, 'blue': 0, 'yellow': -3})
>>> c = +c
>>> c
Counter({'red': 3, 'green': 1, 'pink': 1})
```

- `elements()` returns an iterator (which I used to generate a `list`)
- `most_common()` returns the x most common elements
- `subtract()` does what you'd think, preserves negative values
- + generates new list, discarding 0s or negatives

Counters (cont'd)

Develop Intelligence

- `update()` method updates the counts

```
>>> c
Counter({'red': 3, 'green': 1, 'pink': 1})
>>> c.update(red=1, green=5, pink=-2)
>>> c
Counter({'green': 6, 'red': 4, 'pink': -1})
>>> c.update('syzygy')
>>> c
Counter({'green': 6, 'red': 4, 'y': 3, 's': 1, 'z': 1, 'g': 1, 'pink': -1})
```

- `update()` can also accept another Counter

```
>>> d = Counter(red=6, y=7, g=9)
>>> c.update(d)
>>> c
Counter({'y': 10, 'red': 10, 'g': 10, 'green': 6, 's': 1, 'z': 1, 'pink': -1})
```

Counters (cont'd)

Develop Intelligence

- math operations on Counters are not quite what you'd expect…

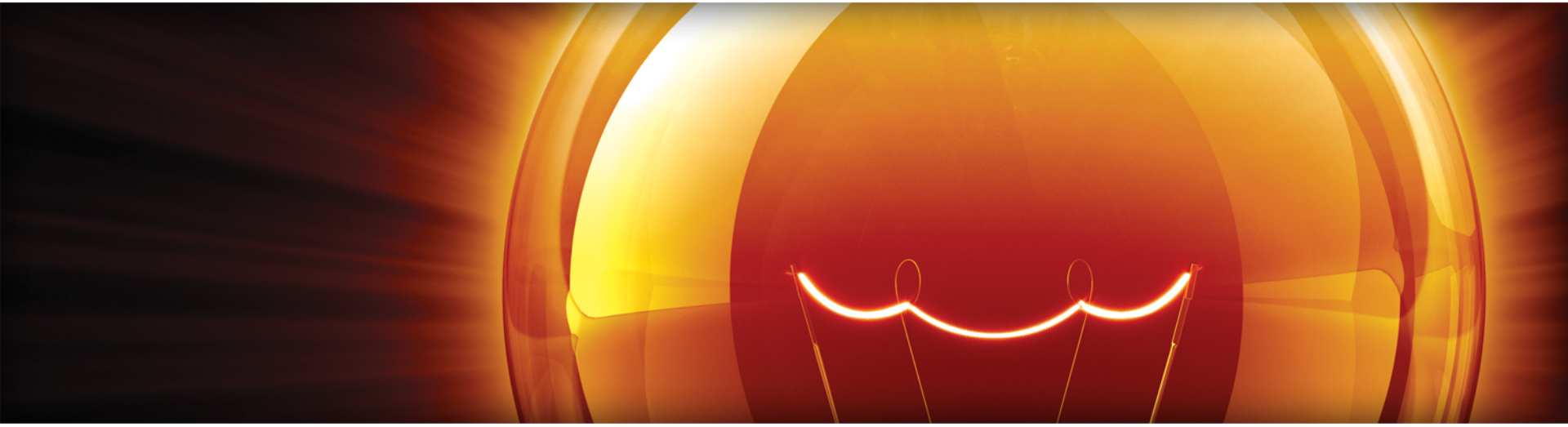
```
>>> c = Counter(a=3, b=1, c=4)
>>> d = Counter(a=1, b=2, c=5)
>>> c + d
Counter({'c': 9, 'a': 4, 'b': 3})
>>> c - d
Counter({'a': 2})
>>> c & d
Counter({'c': 4, 'b': 1, 'a': 1})
>>> c | d
Counter({'c': 5, 'a': 3, 'b': 2})
```

- – keeps only positive elements (cf. `subtract`)
- `&` = min(c[x], d[x])
- `|` = max(c[x], d[x])

Lab: Counters

- Use a `Counter` to count the words in a file. That is, read in a file, separate it into words, and use a Counter to count the number of occurrences of each word in the file. As before, don't worry about error checking for now. We will learn how to deal with errors shortly.

Functional Programming



Learning Solutions to Attract, Retain, and Grow your top technical talent.

Python functions

- functions are "first class" objects, i.e., a program entity that can be
  - created at runtime
  - assigned to a variable or element in a data structure
  - passed as an argument to a function
  - returned as the result of a function

```
>>> def fact(n):
...     '''returns n!'''
...     if n < 2:
...         return 1
...     else:
...         return n * fact(n-1)
...
>>> fact(3)
6
>>> fact(50)
30414093201713378043612608166064768844377641568960512000000000000
>>> fact.__doc__
'returns n!'
>>> type(fact)
<class 'function'>
```

function created at runtime

`__doc__` is an attribute of a function object

`fact` is an instance of the function class

Python functions (cont'd)

Develop Intelligence

- now let's examine the "first class" nature of a function

```
>>> f = fact
>>> f
<function fact at 0x102ab6ea0>
>>> f(8)
40320
>>> map(fact, range(9))
<map object at 0x102aaf518>
>>> list(map(fact, range(9)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320]
```

function fact assigned to variable f

…and the function can be called via that name

the function can also be passed as an argument to `map()`

- `map()` takes a function as its first argument and returns an iterable where each item is the result of applying the function to successive elements of the second argument (an iterable)

Higher order functions

Develop Intelligence

- a function that takes another function as an argument or returns a function as a result
  - `map()` (as well as `filter()` and `reduce()`)
  - `sorted()`—takes an optional `key` arg which lets you provide a function which is applied to each item for sorting,

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry',
'banana']
>>> sorted(fruits)
['apple', 'banana', 'cherry', 'fig', 'raspberry', 'strawberry']
>>> sorted(fruits, key=len)
['fig', 'apple', 'cherry', 'banana', 'raspberry', 'strawberry']
>>> def reverse(word):
...     return word[::-1]
...
>>> reverse('Python')
'nohtyP'
>>> sorted(fruits, key=reverse)
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
```

sort by length, rather than alphabetically

create a function to reverse a word

sort the list, using the reverse() function applied to each item

WTH?

Higher order functions (cont'd)

Develop Intelligence

```
>>> list(range(6))
[0, 1, 2, 3, 4, 5]
>>> list(filter(lambda n: n % 2, range(6)))
[1, 3, 5]
```

- `filter()` applies its first arg, a function, to its second argument–in this case, we are creating a lambda (an anonymous function), which, for any `n`, returns `n % 2`
  - we previously created the function `fact()` to pass to `map()`, and we could have used a lambda function
- we can further combine functions…

Higher order functions (cont'd)

Develop Intelligence

```
>>> list(filter(lambda n: n % 2, range(6)))
[1, 3, 5]
>>> list(map(fact, filter(lambda n: n % 2, range(6))))
[1, 6, 120]
```

- if this looks a little daunting, don't fret because we typically accomplish the above a *list comprehension*, which we will introduce in a moment
- important to understand because you may well run across stuff like the above in legacy code

Higher order functions (cont'd)

Develop Intelligence

- `reduce()` was a builtin function in Python 2 but now has been "demoted" to the `functools` module in Python 3

```
>>> from functools import reduce
>>> from operator import add
>>> reduce(add, range(101))
5050
>>> sum(range(101))
5050
```

this is the '+' function

- a reducing function produces a single aggregate result from a sequence or from any finite iterable object
- the most common use case of `reduce()`, summation, is better served by the `sum()` built-in available since Python 2.3
- many examples of reduce() are clearer when written as `for` loops!

lambda functions

- the `lambda` keyword creates an anonymous function within a Python expression
  - Python limits the body of lambda functions to be pure expressions
  - i.e., cannot make assignments or use any other Python statement such as `while`, `try`, etc.
  - best use of `lambda` is in the context of an argument list
  - remember we defined a function called `reverse()` when sorting the list of fruits…let's try again with a `lambda`…

lambda functions (cont'd)


```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry'
, 'banana']
>>> def reverse(word):
...     return word[::-1]
...
>>> sorted(fruits, key=reverse)
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
>>> sorted(fruits, key=lambda word: word[::-1])
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
```

- instead of defining a function which we used solely in the call to `sorted()`, we use an anonymous function

List Comprehensions ("listcomps")

Develop Intelligence

- Quick way to build a list!
- listcomps are more readable
- often faster
- which is easier to read?

```
>>> string = 'abcABC* '
>>> ascii_codes = []
>>>
>>> for char in string:
...   ascii_codes.append(ord(char))
...
>>> ascii_codes
[97, 98, 99, 65, 66, 67, 42, 32]
```

or…

```
>>> string = 'abcABC* '
>>> ascii_codes = [ord(char) for char in string]
>>> ascii_codes
[97, 98, 99, 65, 66, 67, 42, 32]
```

List Comprehensions ("listcomps")...cont'd

Develop Intelligence

- listcomps can generate a lists from the Cartesian product or two or more iterables…

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> tshirts = [(color, size) for color in colors for size in sizes]

>>> tshirts
[('black', 'S'), ('black', 'M'), ('black', 'L'), ('white', 'S'), ('
white', 'M'), ('white', 'L')]
```

listcomps (cont'd)

- Note that they are not list **in**comprehensions, so…
  - keep them short
  - use line breaks since they are ignored inside [] (and {}, ()) and you therefore don't need the ugly \ line continuation
  - don't use a listcomp unless you are doing something with the resulting list!

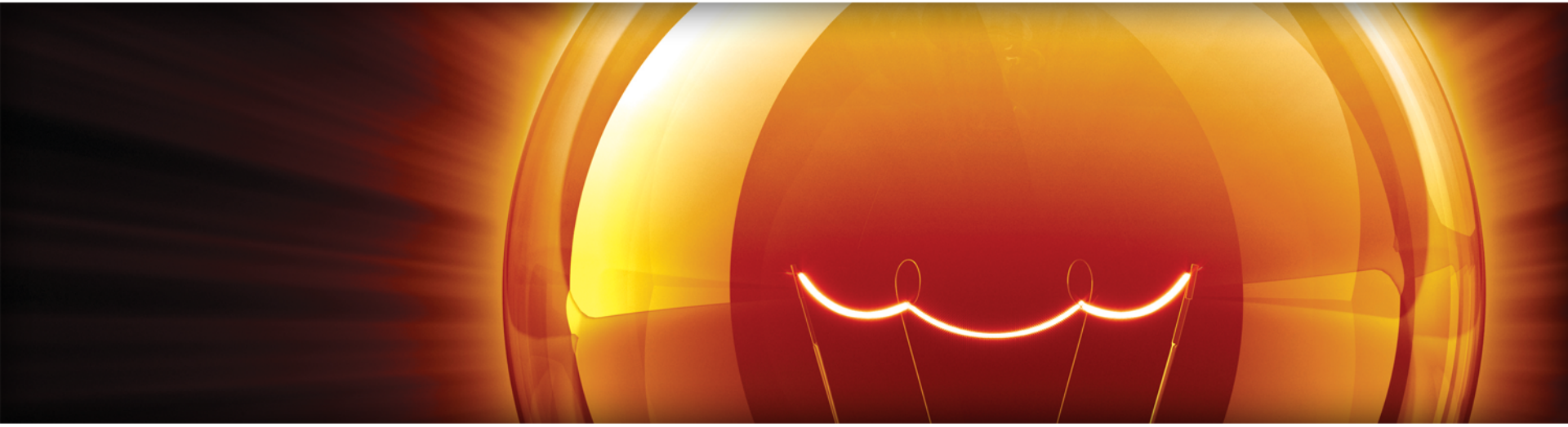
Lab: listcomps

Develop Intelligence

- Modify the DeckOfCards example (available in github.com/davewadestein/Python-Intermediate) to use listcomps when building `rank` (list of card ranks) and `_card` (the cards in the deck). Make sure your solution is easier to read than the `for` loops it is replacing.
- Start with Cartesian product example (colors x sizes of t-shirts) and add a third iterable, `sleeves = [ 'short', 'long' ]` and then create a new listcomp which generates the Cartesian product colors x sizes x sleeves. Once you've done this, `tshirts` should

```
>>> tshirts
[('black', 'S', 'short'), ('black', 'S', 'long'), ('black', 'M', 'short'),
 ('black', 'M', 'long'), ('black', 'L', 'short'), ('black', 'L', 'long'),
('white', 'S', 'short'), ('white', 'S', 'long'), ('white', 'M', 'short'),
('white', 'M', 'long'), ('white', 'L', 'short'), ('white', 'L', 'long')]
```

Do This Then That: Advanced else Blocks



Learning Solutions to Attract, Retain, and Grow your top technical talent.

Do This, Then That: `else` Blocks Beyond `if`

Develop Intelligence

- a perhaps underappreciated Python feature–the `else` clause can be used not only in `if` statements but also in `for`, `while`, and `try` statements
  - `for`: the `else` block will run only if and when the `for` loop runs to completion (i.e., not if the `for` is aborted via `break`).

```
# loop through data, ensuring an acceptable value can be found
for x in data:
    if acceptable(x):
        break
else:
    raise ValueError("No acceptable value found in {}".format(data))
```

- (By the way, how to determine whether the loop executed at all?)

while/else

Develop Intelligence

- `while`: the `else` block will run only if and when the `while` loop exits because the condition became false (i.e., not when the `while` is aborted via `break`).

```
1  import random
2  my_number = random.randint(1, 100)
3  guess = 0
4
5  while guess != my_number:
6      guess = int(input("Your guess (0 to give up)? "))
7      if guess == 0:
8          print("Sorry that you're giving up!")
9          break
10     elif guess > my_number:
11         print("Guess was too high")
12     else:
13         print("Guess was too low")
14 else:
15     print("Congratulations. You guessed it!")
```

- See https://github.com/davewadestein/Python-Intermediate

try/else

- `try`: the `else` block will only run if no exception is raised in the `try` block. The official docs also state: “Exceptions in the `else` clause are not handled by the preceding except clauses.”
- Might seem redundant…

```
try:
    dangerous_call()
    after_call()
except OSError:
    log('OSError...')
```

- `after_call()` will only run if `dangerous_call()` does not throw an exception, right? So what's the problem?

try/else (cont'd)


```
try:
    dangerous_call()
except OSError:
    log('OSError...')
else:
    after_call()
```

- now it's clear that the `try` block is guarding against possible errors in `dangerous_call()` and not in `after_call()`
- it's also more obvious that `after_call()` will only execute if no exceptions are raised in the `try` block.

`try/else` (cont'd)

Develop Intelligence

- use case from Python glossary (https://docs.python.org/3/glossary.html) EAFP = Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the LBYL style common to many other languages such as C.
- LBYL = Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many if statements.

  In a multi-threaded environment, the LBYL approach can risk introducing a race condition between “the looking” and “the leaping”. For example, the code, if key in mapping: return mapping[key] can fail if another thread removes key from mapping after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

Iteration

Develop Intelligence

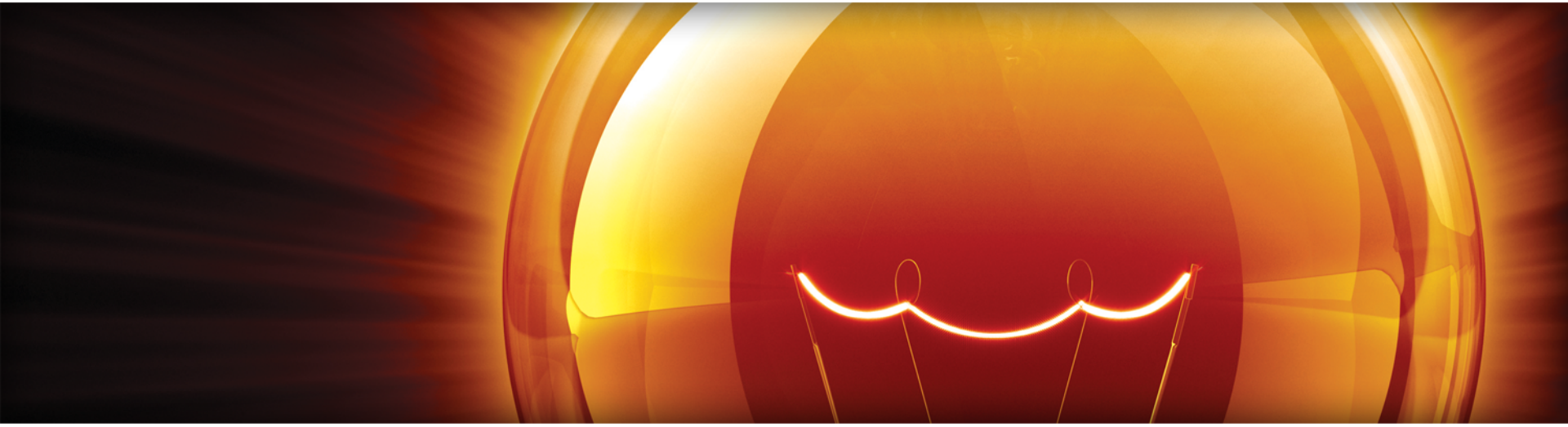
- iteration is fundamental to data processing
- when scanning datasets that don't fit in memory, we need a way to fetch the items lazily, i.e., one at a time and on demand
- every collection in Python is iterable, and iterators are used internally to support
  - for loops
  - looping over text files line by line
  - list, dict, and set comprehensions
  - tuple unpacking
  - unpacking actual parameters with * in function calls

Iteration

Develop Intelligence

- iteration is fundamental to data processing
- when scanning datasets that don't fit in memory, we need a way to fetch the items lazily, i.e., one at a time and on demand
- every collection in Python is iterable, and iterators are used internally to support
  - for loops
  - looping over text files line by line
  - list, dict, and set comprehensions
  - tuple unpacking
  - unpacking actual parameters with * in function calls

I'd like to have an argument



Learning Solutions to Attract, Retain, and Grow your top technical talent.

Arguments

Develop Intelligence

- functions generally need arguments in order to do something useful
  - normally we use positional arguments, i.e., we define them in some order in the function, and then supply them in the same order when calling that function
- advantage of keyword arguments is that you can pass arguments in a different order from the way they were defined
  - and, as we've seen you can skip arguments completely if they have a default value
- Python therefore encourages functions which support lots of arguments with default values
- ***"Explicit is better than implicit"***
  - in other words, arguments can be passed out of order ONLY if they're passed by keyword
  - keywords are even more explicit than positions because the call to the function actually documents the purpose of its arguments

Arguments (cont'd)

Develop Intelligence

- planning for flexibility when it comes to function arguments is especially important when you're writing code for other people
  - if you don’t know the exact needs of users who will be using your code, best to move assumptions into arguments that can be overridden later

```
def add_prefix(string, prefix='foo_'):
    return prefix + string

s = input('Enter a string: ')
print(add_prefix(s))
print(add_prefix(s, 'dive_'))

'''These examples fail because we are passing a positional
argument (s) after a keyword argument.

print(add_prefix(prefix='sand_', s))
sorted(reverse=True, list)
'''

'''Once a keyword argument is passed, ALL arguments following
must be keyword arguments'''

print(add_prefix(prefix='sand_', string=s))
```

Keyword Arguments

- there are actually two related concepts called "keyword arguments"
  - When you call a function you can specify some (or all) arguments by name. Required arguments which were not mentioned at all must have a default value.
  - You can define a function that takes arguments by name without specifying what those names are. These are pure keyword arguments, and can't be passed positionally.

Variable Positional Arguments

Develop Intelligence

- Consider a typical shopping cart. Adding items to the cart could be done one at a time or in batches

```
class ShoppingCart:
    def add_to_cart(items):
        self.items.extend(items)
```

- Adding a single item is now awkward, because the add_to_cart() method needs a list argument

```
cart.add_to_cart([item])
```

- Ideally, the method should support the standard syntax for single arguments, while still supporting multiple arguments, and we do this as follows:

```
def add_to_cart(*items):
    self.items.extend(items)
```

Variable Positional Arguments (cont'd)

Develop Intelligence

```
>>> def func(*args):
...     print("args passed are:", args)
...
>>> func()
args passed are: ()
>>> func(1)
args passed are: (1,)
>>> func(-12, 19, 'foo', 13782)
args passed are: (-12, 19, 'foo', 13782)
>>>
>>> func(['a', 'list'])
args passed are: (['a', 'list'],)
>>> func('this is a test'.split())
args passed are: (['this', 'is', 'a', 'test'],)
```

Lab: Variable Positional Arguments

Develop Intelligence

- write a function called `product()` which accepts a variable number of arguments and returns the product of all of its arg. With no args, `product()` should return 1.

```
product(3, 5) = 15
product(1, 2, 3) = 6
product(63, 12, 3, 9, 0) = 0
product( ) = 1
```

Variable Keyword Arguments

- What if a function needs extra configuration options, which typically have default values which are not overridden?
  - obvious way to do this would be to have the function accept a dict in which these value(s) can be specified
  - ```
    class ShoppingCart(object):
        def __init__(self, options):
            self.options = options
    ```
- Consider the need for a shopping cart in € as opposed to $

```
eurocart = ShoppingCart({'Currency' : 'EUR'})
```

- instead, use variable keyword arguments

```
def __init__(self, **options):
cart = ShoppingCart(Currency = 'EUR')
```
```

Variable Keyword Arguments


```
'''This function takes 2 positional arguments, followed by 0+
keywords arguments.'''
def func1(pos1, pos2=1, **kwargs):
    print("pos1 =", pos1)
    print("pos2 =", pos2)
    print("kwargs =", kwargs)

'''This function takes 2+ positional arguments, followed by 0+
keywords arguments.'''
def func2(pos1, pos2=1, *args, **kwargs):
    print("pos1 =", pos1)
    print("pos2 =", pos2)
    print("args =", args)
    print("kwargs =", kwargs)
```

Variable Keyword Arguments

Develop Intelligence

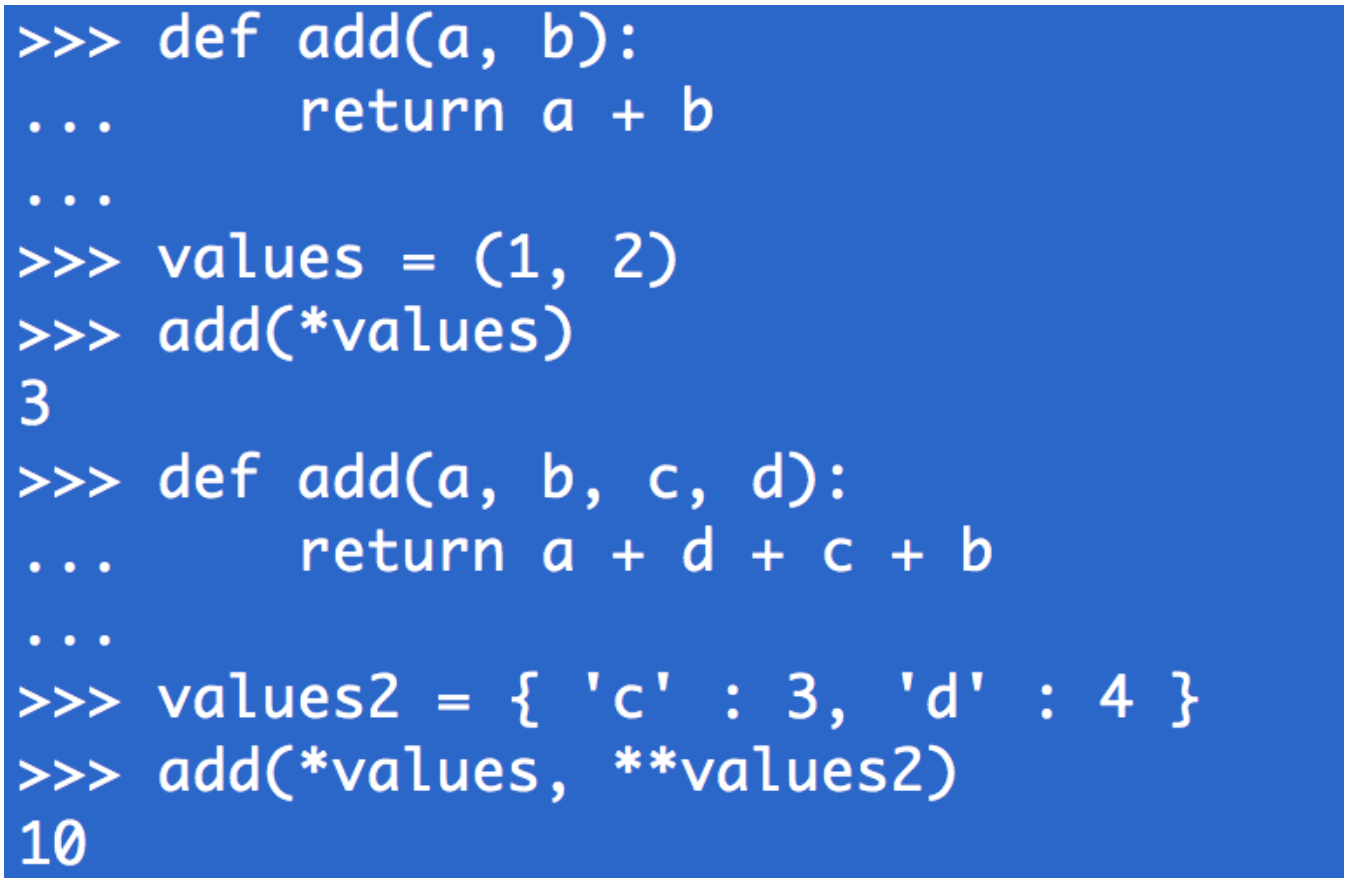
```
>>> from keywordargs import func1, func2
>>>
>>> func1()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: func1() missing 1 required positional argument: 'pos1'
>>> func1(5)
pos1 = 5
pos2 = 1
kwargs = {}
>>> func1(5, 6, foo=True, bar=False, baz='ooka')
pos1 = 5
pos2 = 6
kwargs = {'baz': 'ooka', 'foo': True, 'bar': False}
>>> func2(5, 6, 7, 8, 9, foo=True, bar=False, baz='ooka')
pos1 = 5
pos2 = 6
args = (7, 8, 9)
kwargs = {'baz': 'ooka', 'foo': True, 'bar': False}
```

Argument Recap

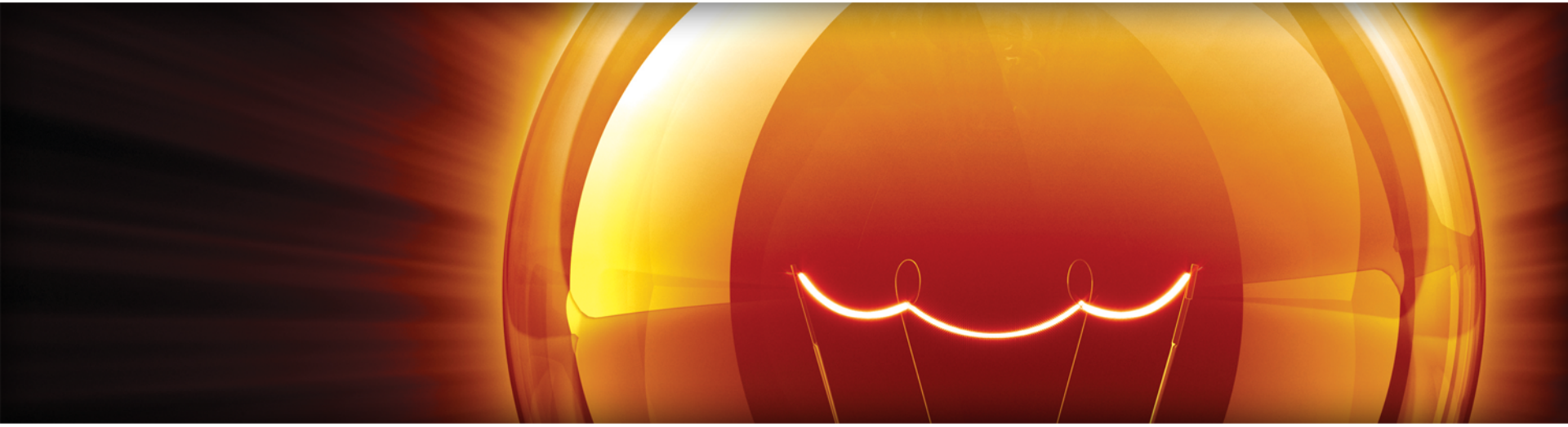
Develop Intelligence

- four types of args, in order of how they generally appear in a function:
  - Required arguments
  - Optional arguments
  - Variable positional arguments
  - Variable keyword arguments
    - required args must be satisfied prior to getting into the optional arguments
    - variable args pick up values that didn't fit into anything else, so they get defined at the end
  - refer back to `func1()`/`func2()` on previous slide

*/** in function call



Decorators



Learning Solutions to Attract, Retain, and Grow your top technical talent.

Decorators

Develop Intelligence

- sometimes you want to modify a function's behavior without explicitly modifying the function, e.g., pre/post actions, debugging, etc.
- suppose we have a set of tasks that need to be performed by many different functions, e.g.,
  - access control
  - cleanup
  - error handling
  - logging
- in other words, there is some boilerplate code that needs to be executed before or after what the function actually does

Decorators (cont'd)

- to scratch the surface of decorators we will rely on some concepts we already know:
  - nested functions
  - *args, **kwargs
  - everything in Python is an object, even functions

Decorators (cont'd)


```
def document_it(func):
    def new_function(*args, **kwargs):
        print('Running function: {}'.format(func.__name__))
        print('Positional arguments: {}'.format(args))
        print('Keyword arguments: {}'.format(kwargs))
        result = func(*args, **kwargs)
        print('Result: {}'.format(result))
        return result

    return new_function

def add_ints(a, b, foo = 'bar', datatype = 'int'):
    return a + b

print('Running plain old add_ints()')
print(add_ints(3, 5))

''' manual decorator assignment '''
cooler_add_ints = document_it(add_ints)

print('Running cooler add_ints()')
cooler_add_ints(3, 5)
```

inner function

here's where we invoke the function that was passed in to `document_it()`

Decorators (cont'd)


```
def document_it(func):
    def new_function(*args, **kwargs):
        print('Running function: {}'.format(func.__name__))
        print('Positional arguments: {}'.format(args))
        print('Keyword arguments: {}'.format(kwargs))
        result = func(*args, **kwargs)
        print('Result: {}'.format(result))
        return result
    return new_function

@document_it
def add_ints(a, b):
    return a + b

add_ints(3, 5)
```

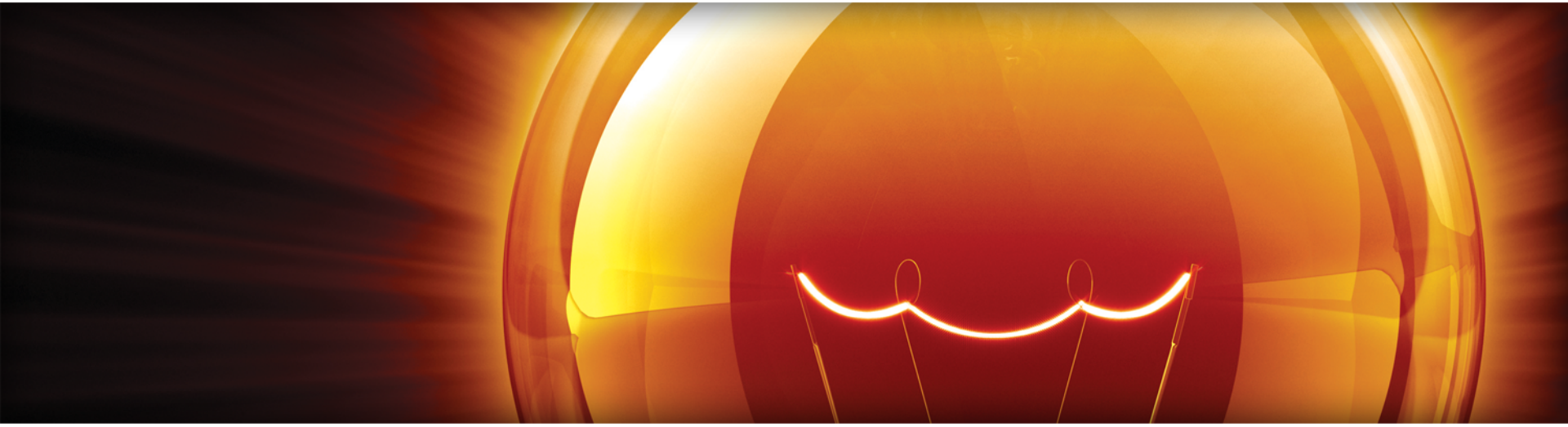
decorator shorthand for what we did in previous slide

Lab: Decorators

Develop Intelligence

- First, create a class, Coord2D, which represents a point in 2-D space (i.e., x and y coordinates). Implement the following methods:
  - distance(), which computes the distance between the current Coord2D object and another Coord2D object, i.e., d = c1.distance(c2), where c1 and c2 are Coord2D objects and d is a scalar. Use math.sqrt or math.hypot.
  - add(): adds a Coord2D object to the current one and returns a new Coord2D object which represents their sum, i.e.,
    c3 = c1.add(c2)
  - sub(), which subtracts a Coord2D object from the current one, i.e. c3 = c1.sub(c2)
  - noneg(), which returns a new Coord2D object whose coordinates are not negative, so c1 = c1.noneg() would ensure c1 has nonnegative coordinates
- Create a decorator to ensure that the arguments to a function, as well as its return value, are nonnegative. Decorate the add() and sub() methods.

Context Managers



Learning Solutions to Attract, Retain, and Grow your top technical talent.

The `with` statement

Develop Intelligence

- Sometimes you don't want to catch/silence exceptions, but want to be sure some cleanup is done regardless of what happens

```
def count_lines(filename):
    """Count the number of lines in a file."""

    file = open(filename, 'r')
    try:
        return len(file.readlines())
    finally:
        file.close()
```

- if file fails to open, exception will be thrown before `try/finally` block, and anything else that can go wrong will go wrong inside the block, so the file will be open by the time we get to the `finally` block…so what's the problem?

The `with` statement (cont'd)

Develop Intelligence

- `with` introduces a new block, like try, but with a very different purpose in mind!
  - The `with` statement sets up a temporary context and reliably tears it down, under the control of a context manager object
- e.g., `open()` allows you to run code in the context of that file, and cleanup is performed automatically, by way of a context manager–a special object that knows about the `with` statement and defines exactly what it means to have code executed in its context

```
>>> with open('myfile', 'w') as file:
...     file.write('Now is the time...')
...     file.closed
...
18
False
>>> file.closed
True
```

context managers

Develop Intelligence

- the context manager runs some code before the `with` clause is executed and runs some cleanup code afterwards
  - in the case of `open()`, the file is opened prior to the `with` block being entered, and closed at the end of the block
  - in this case, the context revolves around an open file object, which is made available to the block via the name given in the `as` clause
    - in other words, all operations inside the `with` clause are said to be executed within the context of the open file
  - in other words, there need not be such an object, and in that case, the `as` clause is optional
- context manager protocol consists of the `__enter__()` and `__exit__()` methods

context manager example: SuppressErrors

Develop Intelligence

```
class SuppressErrors:
    def __init__(self, *exceptions):
        """Populate list of exceptions to suppress.

        If list is empty, suppress ALL exceptions because all exceptions
        inherit from the base class Exception.
        """

        if not exceptions:
            exceptions = (Exception,)
        self.exceptions = exceptions

    def __enter__(self):
        """Nothing to do here. Exception handling occurs in __exit__()."""
        pass

    def __exit__(self, exc_class, exc_instance, traceback):
        """This method "suppresses" exceptions.

        Exception suppression is performed by virtue of the return value.
        If it completes without a return value, the original exception
        will be re-raised. Returning True catches the exception and
        suppresses it.
        """

        if isinstance(exc_instance, self.exceptions):
            return True
        return False
```

- a context manager to suppress exceptions
- `__enter__()` called just prior to execution of interior code
- `__exit__()` takes 3 args and is called when code block finishes
- let's try it…

context manager example: SuppressErrors (ver. 2)

Develop Intelligence

```
class SuppressErrors:
    def __init__(self, *exceptions):
        """Populate list of exceptions to suppress.

        If list is empty, suppress ALL exceptions because all exceptions
        inherit from the base class Exception.
        """

        print("in __init__() method of SuppressErrors")

        if not exceptions:
            print("ALL exceptions will be suppressed.")
            exceptions = (Exception,)
        else:
            print("Only these exceptions will be suppressed: ", end="")
            for exc in exceptions:
                print("{} ".format(exc.__name__), end="")
            print()

        self.exceptions = exceptions

    def __enter__(self):
        """Nothing to do here.

        Exception handling occurs in __exit__().
        NOTE: Whatever is returned from this method will be
        used to populate the variable in the 'as' clause'''
        """
        print("in __enter__() method of SuppressErrors")
        return 'test'
```

```
def __exit__(self, exc_class, exc_instance, traceback):
    """This method "suppresses" exceptions.

    Exception suppression is performed by virtue of the return value.
    If it completes without a return value, the original exception
    will be re-raised. Returning True catches the exception and
    suppresses it.
    """

    print("in __exit__() method of SuppressErrors")
    if isinstance(exc_instance, self.exceptions):
        return True
    # What happens if we don't return anything?
```

- instrumented with `print()` statements to show what is happening
- note that `__enter__()` now returns a value, which would populate variable in `as` clause
- …and `__exit__()` does not always return a value

context manager example: Looking Glass

Develop Intelligence

- frivolous example from the book **Fluent Python**

```
>>> from mirror import LookingGlass
>>> with LookingGlass() as what:
...     print("Lewis Carroll")
...     print(what)
...
llorraC siweL
YKCOWREBBAJ
>>> what
'JABBERWOCKY'
>>> print("back to normal")
back to normal
```

as before, the variable in the `as` clause (`what` in this case), gets the return value of the `__enter__()` function; it is not the context manager object, as we'll see on the next page

context manager example: Looking Glass (cont'd)

Develop Intelligence

```
 1 class LookingGlass:
 2     def __enter__(self):
 3         import sys
 4         self.original_write = sys.stdout.write
 5         sys.stdout.write = self.reverse_write
 6         return 'JABBERWOCKY'
 7
 8     def reverse_write(self, text):
 9         self.original_write(text[::-1])
10
11     def __exit__(self, exc_type, exc_value, traceback):
12         import sys
13         sys.stdout.write = self.original_write
14         if exc_type is ZeroDivisionError:
15             print('Please DO NOT divide by zero!')
16             return True
```

lines 3, 12: `sys` is `imported` multiple times–not a problem, Python caches it

line 5: override Python's "write to stdout" function with a reverse writing function

line 13: restore Python's "write to stdout" function by assigning it the callable object we squirreled away in line 4

Using Looking Glass without a `with` block

Develop Intelligence

```
>>> from mirror import LookingGlass
>>> manager = LookingGlass()
>>> manager
<mirror.LookingGlass object at 0x1021d9438>
>>> monster = manager.__enter__()
>>> monster == 'JABBERWOCKY'
eurT
>>> monster
'YKCOWREBBAJ'
>>> manager
>8349d1201x0 ta tcejbo ssalGgnikooL.rorrim<
>>> manager.__exit__(None, None, None)
>>> monster
'JABBERWOCKY'
```

contextlib

- module provides utilities for common tasks involving the `with` statement
- let's try a context manager for redirecting output which is destined for stdout ("standard output")

```
import sys

from contextlib import redirect_stdout

print("before the with statement")

with redirect_stdout(sys.stderr):
    print("NOTE! the output of help goes to stderr")
    help(pow)

print("after the with statement")
```

contextlib example

Develop Intelligence

- Easy to demonstrate when running from the command line. In Linux, the '>' redirects stdout to a file, and the '2>' redirects stderr ("standard error") to a different file…then we just look at what ended up in each file with the Linux command `cat`…

```
DWS-MacBook:Python-Intermediate dws$ python3 contextlib_ex.py >stdout 2>stderr
DWS-MacBook:Python-Intermediate dws$ cat stdout
before the with statement
after the with statement
DWS-MacBook:Python-Intermediate dws$ cat stderr
NOTE! the output of help goes to stderr
Help on built-in function pow in module builtins:

pow(x, y, z=None, /)
    Equivalent to x**y (with two arguments) or x**y % z (with three arguments)

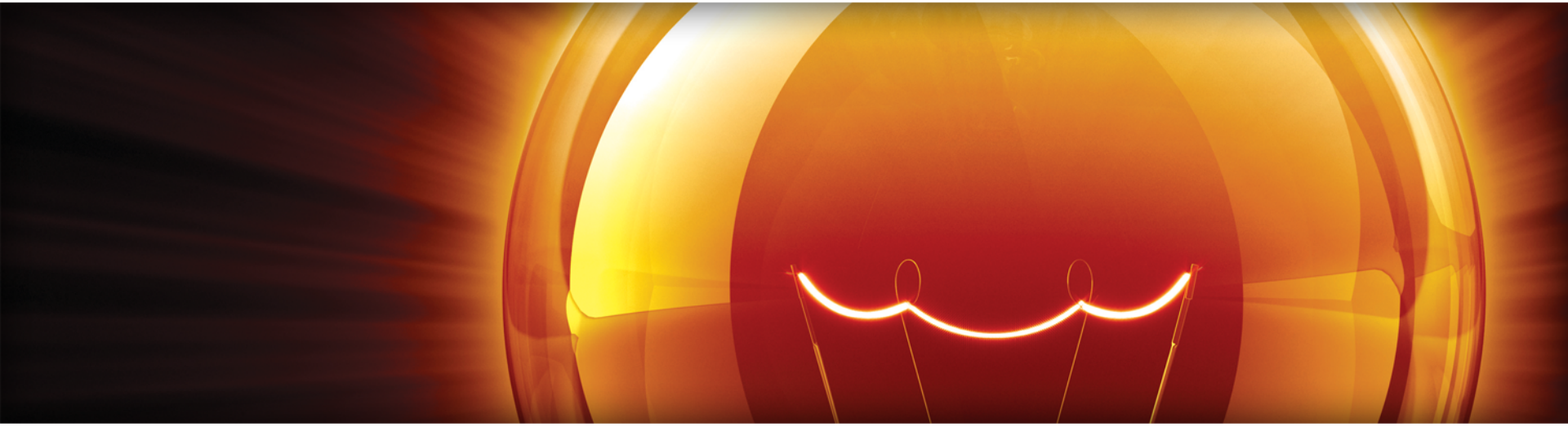
    Some types, such as ints, are able to use a more efficient algorithm when
    invoked using the three argument form.

DWS-MacBook:Python-Intermediate dws$
```

Lab: class vars vs. instance vars

- Variable set outside __init__ belong to the class and are shared by all instances of the class. Can be accessed via ClassName.var and classInstance.var.
- Variables created inside __init__ (and all other method functions) and prefaced with self. belong to the object instance and cannot be accessed via ClassName.
- create a class with some class variables as well as some instance variables so that you can prove that the above is the case. In other words, show that "self.vars" (instance vars) belong to the class instance, whereas class vars belong to the entire class and are shared by all instances of the class

Testing



Learning Solutions to Attract, Retain, and Grow your top technical talent.

Why test your code?

- No programmer is perfect, bugs always occur
- Code can be working and then bugs creep in as APIs change, requirements change, etc.
- Your code is no doubt part of a larger project, which is difficult to test “in the large”

Unit Testing

- The smallest testable parts of an application, called units, are individually and independently scrutinized to ensure they work.
- Your functions/methods/procedures should do ONE thing (and do it well). Testing that thing should be relatively easy to explain.
- Exercise the !$%@!$# out of the unit to be sure it works, especially with corner cases, not just the expected cases.
- white box testing

Unit Testing (cont'd)

- The smallest testable parts of an application, called units, are individually and independently scrutinized to ensure they work.
- Your functions/methods/procedures should do ONE thing (and do it well). Testing that thing should be relatively easy to explain.
- Exercise the !$%@!$# out of the unit to be sure it works, especially with corner cases, not just the expected cases.
- white box testing

Unit Testing (cont'd)

Develop Intelligence

- unit testing general rules
  - run fast
  - standalone
  - independent
  - run full test suite before/after coding sessions
  - write a broken unit test when interrupting your work
  - when debugging code, generate a new test which demonstrates the bug, or better yet, use test-driven development–write a test first which fails, because the code has not been implemented, then implement the code until the test passes

Unit Testing (cont'd)

Develop Intelligence

- unit testing general rules
  - run fast
  - standalone
  - independent
  - run full test suite before/after coding sessions
  - write a broken unit test when interrupting your work
  - when debugging code, generate a new test which demonstrates the bug, or better yet, use test-driven development–write a test first which fails, because the code has not been implemented, then implement the code until the test passes

pytest

Develop Intelligence

- easy, no boilerplate testing
- well tested with more than a thousand tests against itself
- used in many small and large projects and organizations
- needs to be installed, does not come with Python
  - on Mac sudo pip3 install pytest
  - on Windows, go to www.pytest.org

pytest (cont'd)

Develop Intelligence

```
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 4
```

this is the '+' function

function name begins with test_, so pytest will find it

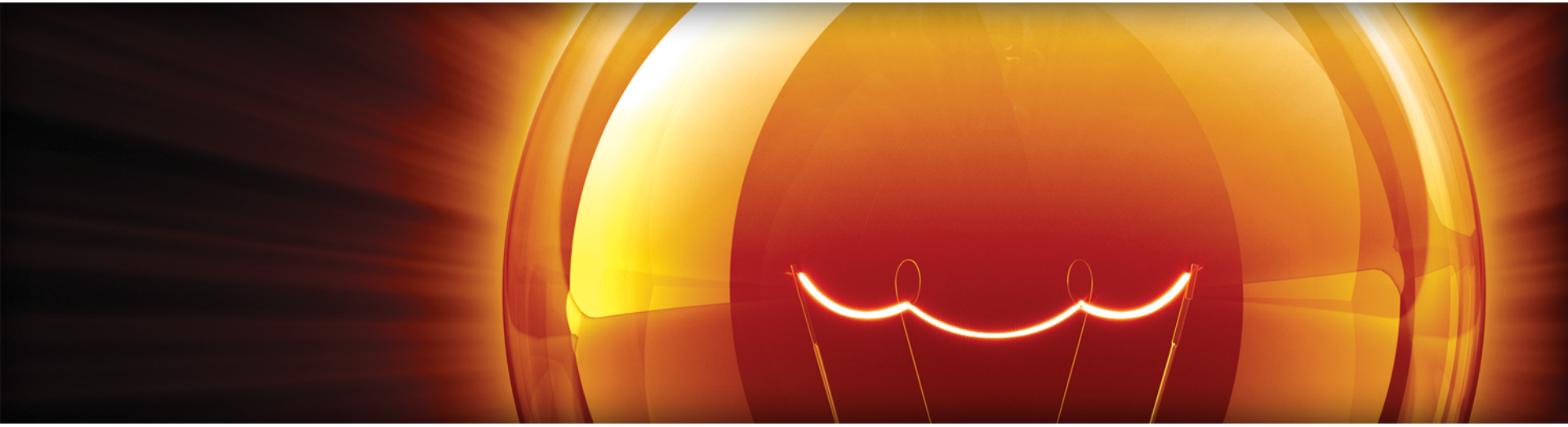
pytest (cont'd)

Develop Intelligence

```
def removeItem(s, i):
    s.remove(i)

def test_answer():
    testSet = set('12345')
    removeItem(testSet, '2')
    assert '2' not in testSet
~
```

Numerical Python (numpy)



Learning Solutions to Attract, Retain, and Grow your top technical talent.

numpy

Develop Intelligence

- open source module providing fast numerical routines for Python
- vector oriented computing
- efficiently implemented multi-dimensional arrays
- designed for scientific computation

numpy (cont'd)

Develop Intelligence

```
>>> import numpy as np
>>> cvalues = [25.3, 24.8, 26.9, 23.9]
>>> C = np.array(cvalues)
>>> print(C)
[ 25.3  24.8  26.9  23.9]
>>> F = C * 9 / 5 + 32
>>> print(F)
[ 77.54  76.64  80.42  75.02]
>>> type(F)
<class 'numpy.ndarray'>
>>> type(cvalues)
<class 'list'>
```

common idiom for importing numpy

create a numpy array from the Python list

scalar multiplication performed by numpy

numpy vs. pure Python


```
import numpy as np
import time

size_of_vec = 10000000

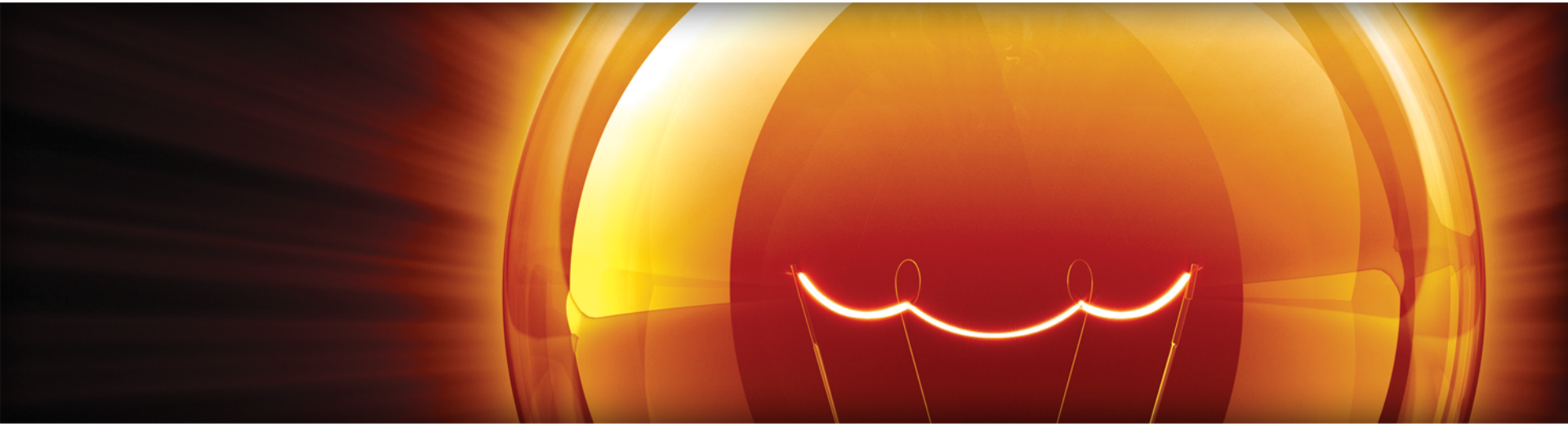
def pure_python_version():
    t1 = time.time()
    X = range(size_of_vec)
    Y = range(size_of_vec)
    Z = []
    for i in range(len(X)):
        Z.append(X[i] + Y[i])
    return time.time() - t1

def numpy_version():
    t1 = time.time()
    X = np.arange(size_of_vec)
    Y = np.arange(size_of_vec)
    Z = X + Y
    return time.time() - t1

t1 = pure_python_version()
t2 = numpy_version()
print("python %.2f, numpy %.2f (%.2fx faster)" % (t1, t2, t1/t2))
```

```
python 4.79, numpy 0.12 (38.53x faster)
```

Logging



Learning Solutions to Attract, Retain, and Grow your top technical talent.

Logging

- Logging is a means of tracking events that happen when some software runs
- We add logging calls to our code to indicate that certain events have occurred

```
import logging
logging.warning('Watch out!') # will print a message to the console
logging.info('I told you so') # will not print anything
```

- in the example above, logging output comes to the screen, but we can instead specify that it go to a file
- default log warning level is WARNING, so we don't see less important messages unless we change the level…

Logging (cont'd)

Develop Intelligence

```
import logging
logging.basicConfig(filename='example.log', level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
```

- in the basicConfig() call we set the name of the log file, and the logging level, so this time we see INFO and DEBUG messages as well

Level	When it's used
DEBUG	Detailed information, typically of interest only when diagnosing problems.
INFO	Confirmation that things are working as expected.
WARNING	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
ERROR	Due to a more serious problem, the software has not been able to perform some function.
CRITICAL	A serious error, indicating that the program itself may be unable to continue running.

Class vars vs. instance vars

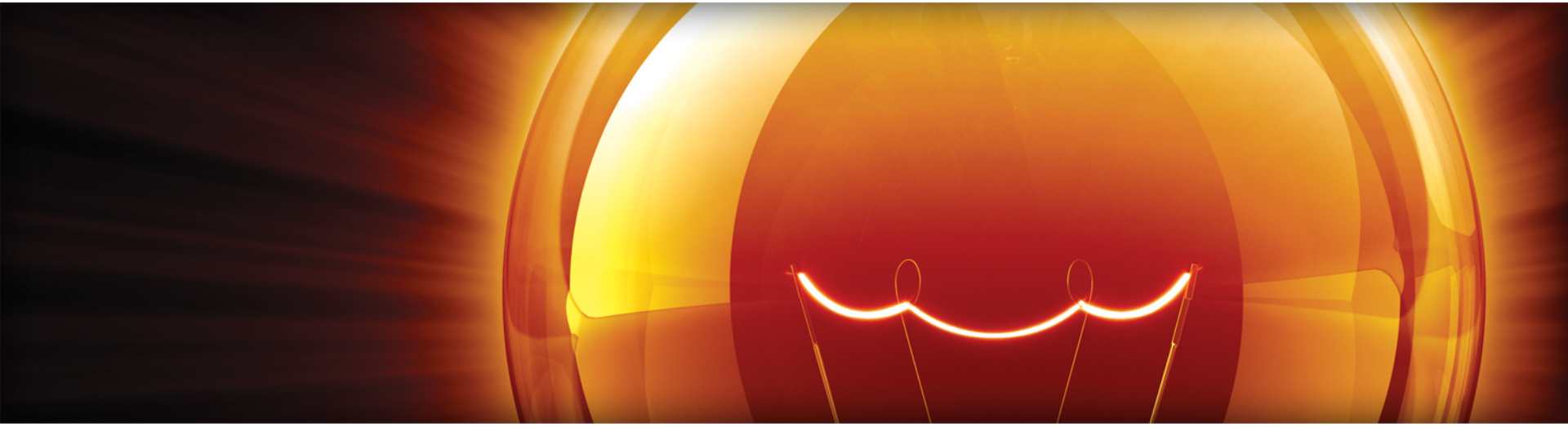
Develop Intelligence

```
>>> class foo():
...   class_data = 'this is shared by all instances'
...   def __init__(self):
...     self.instance_data = 'this is specific to the instance'
...
>>> f1 = foo()
>>> f1.class_data
'this is shared by all instances'
>>> f1.instance_data
'this is specific to the instance'
>>> f2 = foo()
>>> f2.class_data
'this is shared by all instances'
>>> f2.instance_data
'this is specific to the instance'
>>> foo.class_data
'this is shared by all instances'
>>> foo.class_data = 'I just changed shared data!'
>>> f1.class_data
'I just changed shared data!'
>>> f2.class_data
'I just changed shared data!'
>>> f1.instance_data = 'this is local to f1'
>>> f2.instance_data
'this is specific to the instance'
```

Python 2/3 differences

- In addition to official docs, I recommend this link: http://www.diveintopython3.net/porting-code-to-python-3-with-2to3.html

The Python Data Model



Learning Solutions to Attract, Retain, and Grow your top technical talent.

The Python Data Model

- if you learned another object-oriented language first, you may find it strange to write `len(list)` instead of `list.len()`.
  - "This apparent oddity is the tip of an iceberg that, when properly understood, is the key to everything we call Pythonic." –Luciano Ramalho, **Fluent Python**
  - the iceberg is the Python data model, which describes the API you can use to make your own objects play nice with others
- the interpreter invokes special (or so-called "magic") methods to perform basic operations on objects, often triggered by special syntax
  - these "magic" methods have leading/trailing double underscores
    - e.g., to parse `my_collection[key]` the interpreter calls `my_collection.__getitem__(key)`

The Python Data Model (cont'd)

Develop Intelligence

- special methods are intended to be called by the Python interpreter, not by you
  - ~~`my_object.__len__()`~~
  - len(my_object)
    - Python calles the `__len__()` method you implemented
- more often than not, special methods are implicit
  - e.g., `for i in x:`
    - causes the invocation of `iter(x)`, which in turn may call `x.__iter__()` if available
- the only special method that is frequently called by user code directly is `__init__()`, to invoke the initializer of the superclass in your own `__init__()` method

The Python Data Model (cont'd)

Develop Intelligence

- If you need to invoke a special method, it is usually better to call the related built-in function (e.g., `len`, `iter`, `str`, etc.)
  - these built-ins call the corresponding special method, but often provide other services and—for built-in types—are faster than method calls
- you should avoid creating custom special methods, e.g., `__foo__()` because they may in the future acquire some special meaning