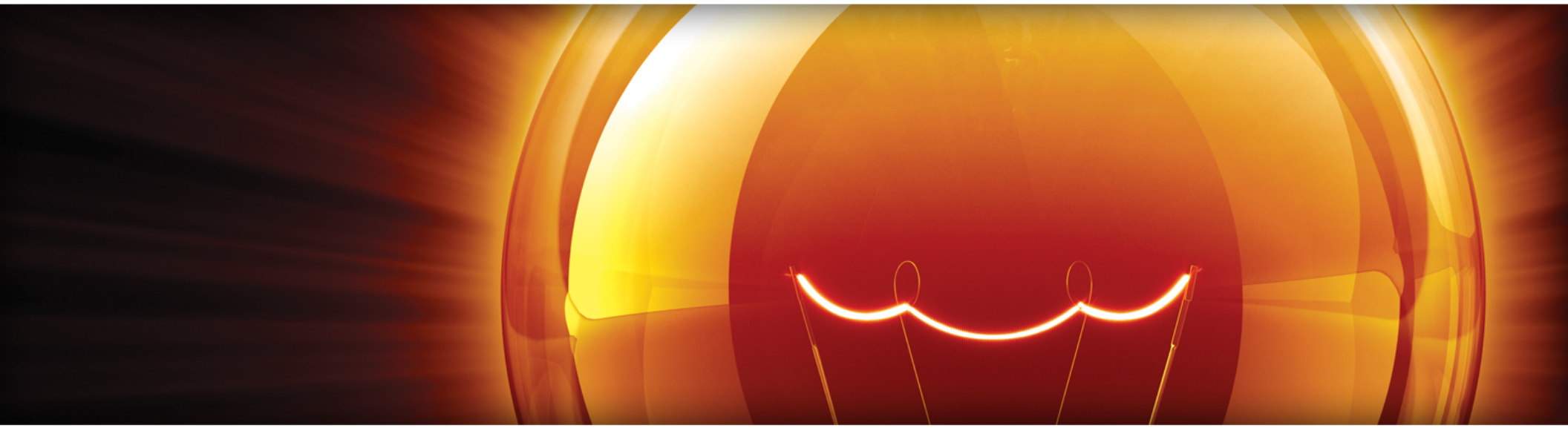
Develop Intelligence



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

Core Python

Develop Intelligence

- Instructor:
  Dave Wade-Stein
  dws@developintelligence.com
  720-936-7783
- Class Time: 9-5
  Lunch: 12-1
  10 min. break every hour

About You (and me)

- Have you used Python before?
- What is your job title/what do you do?
- What do you want to get out of the course?

Today's Agenda

- What? Why? Why NOT?
- 2 vs. 3
- Installation
- Comparing Python to other Languages
- Variables/Typing
- Built-in Functions
- Python Arithmetic
- Strings
- Indentation/syntax/code blocks/colons
- Controls: if, elif, and else
- Loops

What is Python?

- a scripting language
- a programming language
- a command interpreter
- dynamically typed
- object-oriented

Why Use Python?

Develop Intelligence

- Popular
- easY
- fasT
- pitHy
- brOad
- efficieNt
- plus…
  - Batteries included
  - Large community
  - Likable!

Why NOT Use Python?

- CPU-bound applications can be faster in compiled languages (but not always!)
- high-level…far from the metal
- what else?

Python 2 vs. 3

Develop Intelligence

- "Python 2.x is legacy, Python 3.x is the present and future of the language"
- Python 3.0 released in 2008, 2.7 released in 2010, and there will be no 2.8
- Why might you want to use 2.X?
  - deploying to an environment you don't control, that may impose a specific version, rather than allowing a free selection from available versions
  - you need to use a specific third party package or utility that doesn't yet have a version compatible with Python 3, (and porting that package is non-trivial)

Let's Begin: Python Installation

Installation

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

- BDFL = Guido van Rossum
- `import this`
  - type that into your Python interpreter
  - …and let's take a look at the Python code
    - `this.__file__`
  - …by the end of the course you will understand this Python code

Don't worry, we will understand this code by Day 3!


```
s = """Gur Mra bs Clguba, ol Gvz Crgref

Ornhgvshy vf orggre guna htyl.
Rkcyvpvg vf orggre guna vzcyvpvg.
Fvzcyr vf orggre guna pbzcyrk.
..."""

d = {}
for c in (65, 97):
    for i in range(26):
        d[chr(i+c)] = chr((i+13) % 26 + c)

print("".join([d.get(c, c) for c in s]))
```

Python vs. Mothra

C++

```
#include <iostream>

int main() {
  std::cout << "Hello, world!\n";
}
```

Java

```
public class Main {
    public static void main (String[] args) {
        System.out.format("Hello, world!\n");
    }
}
```

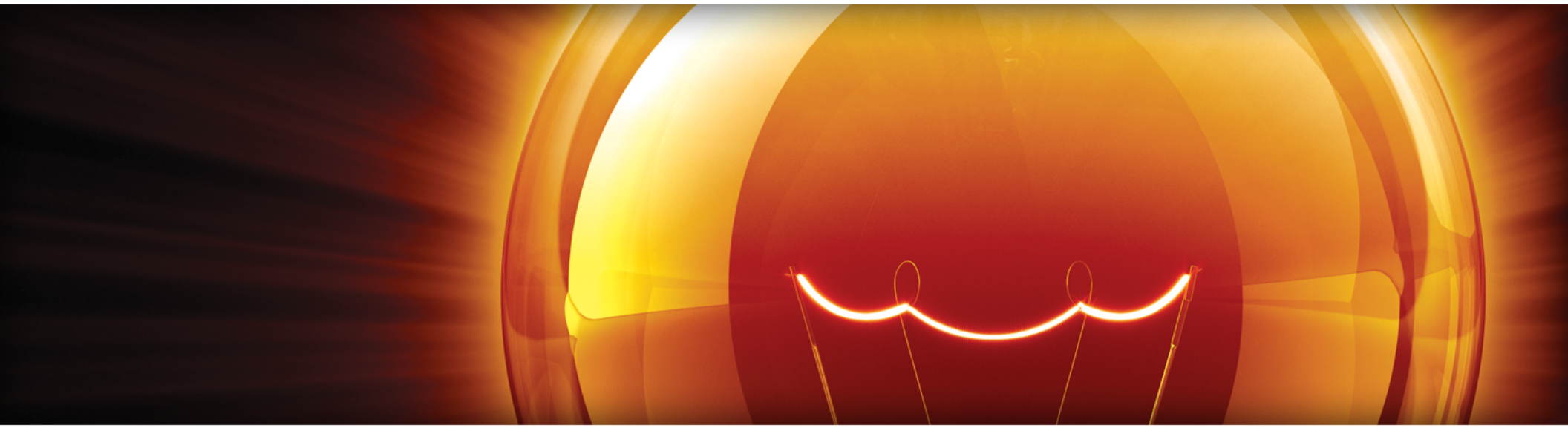
bash

```
echo "Hello, world!"
```

Python

```
print("Hello, world!")
```

Variables and Typing



Variables/Typing

Develop Intelligence

- no declarations
- basic data types are int, float, string, boolean
- everything is an object (keep this in mind)
- dynamically typed

```
>>> x = 3.5
>>> x
3.5
>>> x = 'hello'
>>> x
'hello'
```

Variables/Typing (cont'd)

Develop Intelligence

- strongly typed

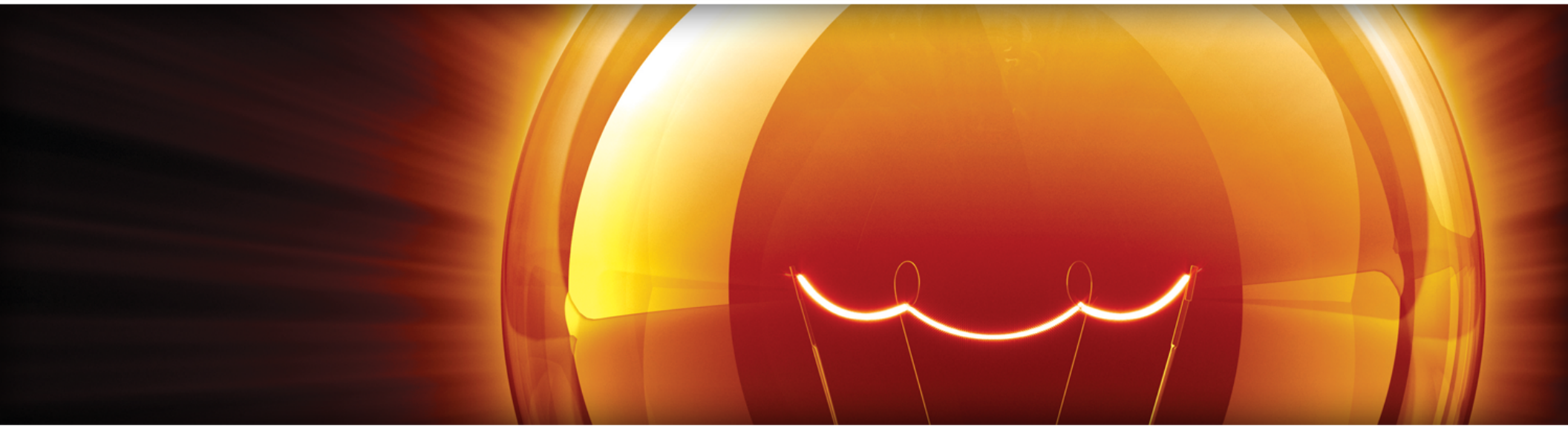
```
>>> x
'hello'
>>> print(x + 3)
Traceback (most recent call last):
  File "<pyshell#133>", line 1, in <module>
    print(x + 3)
TypeError: Can't convert 'int' object to str implicitly
>>> print(x + str(3))
hello3
```

Lab: Variables/Typing

In the Python shell (or IDLE), type in the following and be you understand dynamic typing and strong typing:

```
i = 1
f = 1.4
b = True
s = 'True'
b = 5
s + b
```

Some Built-in Python functions



(and an introduction to 2to3)

str/int

Develop Intelligence

- `str()` returns a string containing a nicely printable representation of the object passed as its argument
- `int()` returns an integer object constructed from its argument–will be an error if not a number!

```
>>> str(304)
'304'
>>> str(True)
'True'
>>> str(1.33e14)
'133000000000000.0'
>>> str('x')
'x'
```

```
>>> x = '503'
>>> int(x)
503
>>> x += 'a'
>>> x
'503a'
```

type

Develop Intelligence

- `type()` returns the type of the object passed as its argument

```
>>> x = 1
>>> type(x)
<class 'int'>
>>> type('hello')
<class 'str'>
>>> x += 0.35
>>> type(x)
<class 'float'>
>>> type(True)
<class 'bool'>
```

print

Develop Intelligence

- `print` was a Python statement in Python 2
- …but it's a function in Python 3

```
>>> x = 'hello'
>>> print x
hello
```

Python 2

```
>>> x = 'hello'
>>> print(x)
hello
>>> print x
SyntaxError: Missing parentheses in call to 'print'
```

Lab: str/int/type

use the Python interpreter (or IDLE) to type the following

```
str(53.3)
str(False)
str(false)
int('300')
int('30x')
type(False)
type('False')
type(3.5)
```

Python Arithmetic

Python arithmetic

Develop Intelligence

- Python interpreter can perform arithmetic
- similar to other languages, except /, which performed integer division in Python 2

```
>>> 3 / 2
1
```

- use // in Python 3 to get the same behavior

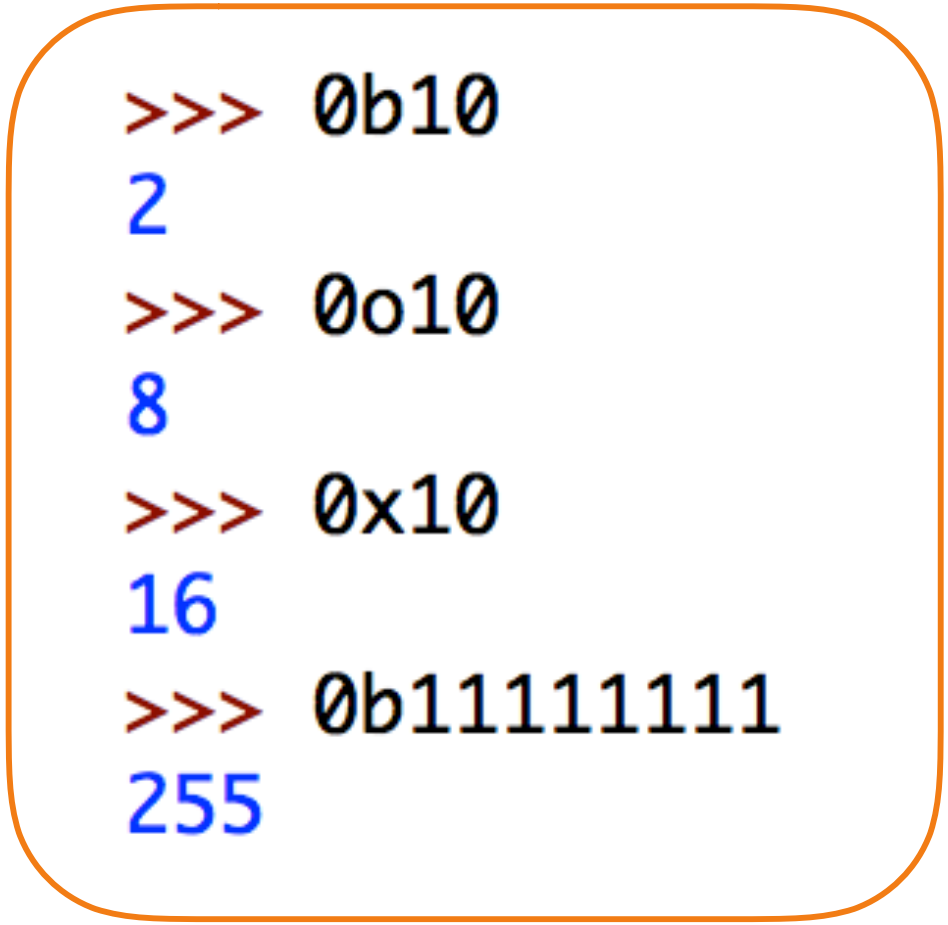
```
>>> 3 / 2
1.5
>>> 3 // 2
1
```

div/mod/divmod

Develop Intelligence

```
>>> 9 // 5
1
>>> 9 % 5
4
>>> divmod(9, 5)
(1, 4)
>>> q, r = divmod(9, 5)
>>> q
1
>>> r
4
```

Non-decimal bases



Size of an int

Develop Intelligence

- `int` limited to 32-bit in Python 2
- `long was limited to 64-bits and was designated by a trailing 'L'`

```
>>> 2 ** 32
4294967296
>>> 2 ** 64
18446744073709551616L
```

- `int` is unlimited in Python 3

```
>>> 2 ** 64
18446744073709551616
>>> 10 ** 100
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000
```

Lab: arithmetic

- try using Python interpreter as a simple calculator
- be sure to note the difference between / and //, and make use of ** for exponentiation
- be sure to try some very large numbers

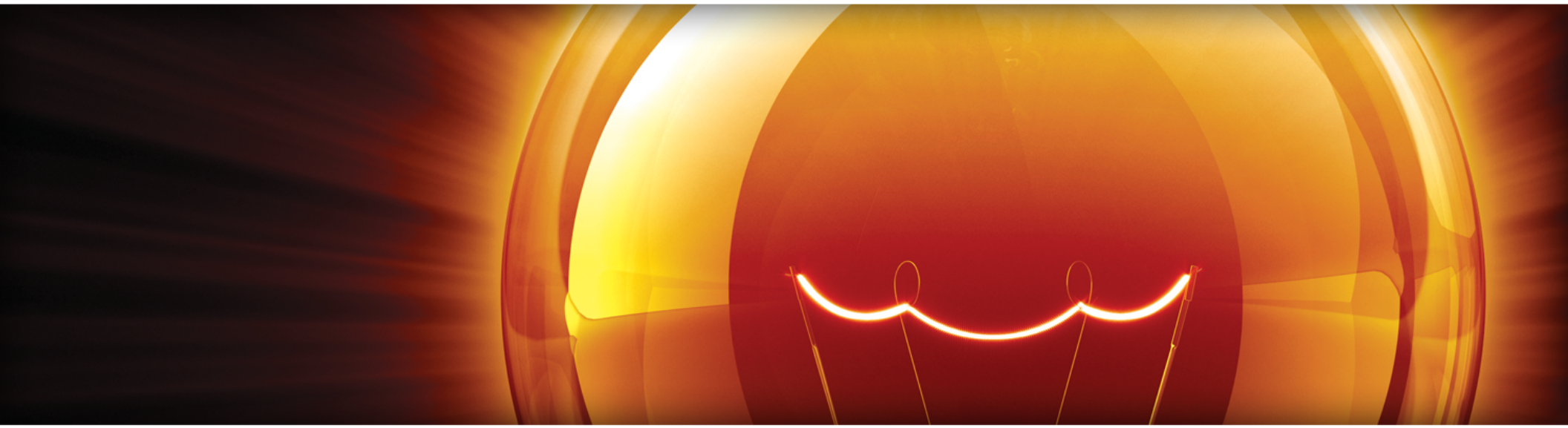
floating point numbers (floats)

Develop Intelligence

- similar to floating point numbers in other languages
- use `float()` to convert to float, just as `int()` converts to integer

```
>>> x = 1
>>> float(x)
1.0
>>> float(True)
1.0
>>> float('hello')
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    float('hello')
ValueError: could not convert string to float: 'hello'
```

Strings



strings

Develop Intelligence

- use single or double quotes
- \ lets you *escape* the next character, i.e., avoid it's usual meaning

```
>>> s1 = "This string isn't a problem"
>>> s1
"This string isn't a problem"
>>> s2 = 'This string is a "good" example'
>>> s2
'This string is a "good" example'
>>> s3 = "This string is \"more difficult\" to read"
>>> s3
'This string is "more difficult" to read'
```

strings (cont'd)

Develop Intelligence

```
>>> palindrome = 'A man,\nA plan,\nA canal:\nPanama.'
>>> palindrome
'A man,\nA plan,\nA canal:\nPanama.'
>>> print(palindrome)
A man,
A plan,
A canal:
Panama.
>>> print('\n')


>>> print('\\n')
\n
```

the '\' escapes the '\' that follows it, resulting in a literal '\' printed to the screen

strings (cont'd)

Develop Intelligence

- + is concatenation operator, * = duplication

```
>>> s, t = 'hello', "bye"
>>> print(s + t)
hellobye
>>> print(s, t)
hello bye
>>> print(s * 4)
hellohellohellohello
>>> u = "Isn't this cool?"
>>> u
"Isn't this cool?"
>>>
```

multi-line strings

Develop Intelligence

- triple quotes allow easy multi-line strings

```
>>> s = '''
this is a mult-line
string
'''
>>> s
'\nthis is a mult-line\nstring\n'
>>> print(s)

this is a mult-line
string

>>>
```

Lab: strings

Develop Intelligence

- be sure you understand what these do (use IDLE or Python shell if you're not sure

```
a, b, o, p = 'b', 'a', 'p', 'o'
o + p + o
a * 3 + b
a + p * 2 + 'k' * 2 + 'e' * 2 + o + 'er'
```

len

Develop Intelligence

- returns length of a string

```
>>> a = 'hello'
>>> len(a)
5
>>> len('')
0
>>> len(a * 5)
25
```

[]

- access a single character via its offset
- easier to think of offset as opposed to index
- negative offsets count from end of string

```
>>> alphabet = 'abcdefghijklmnopqrstuvwxyz'
>>> alphabet[0]
'a'
>>> alphabet[25]
'z'
>>> alphabet[-1]
'z'
>>> alphabet[-26]
'a'
```

strings: Python 2 vs. 3

- in Python 2, strings were just sequences of bytes (characters)
- strings had to be prefaced with 'u' in order to be treated as Unicode

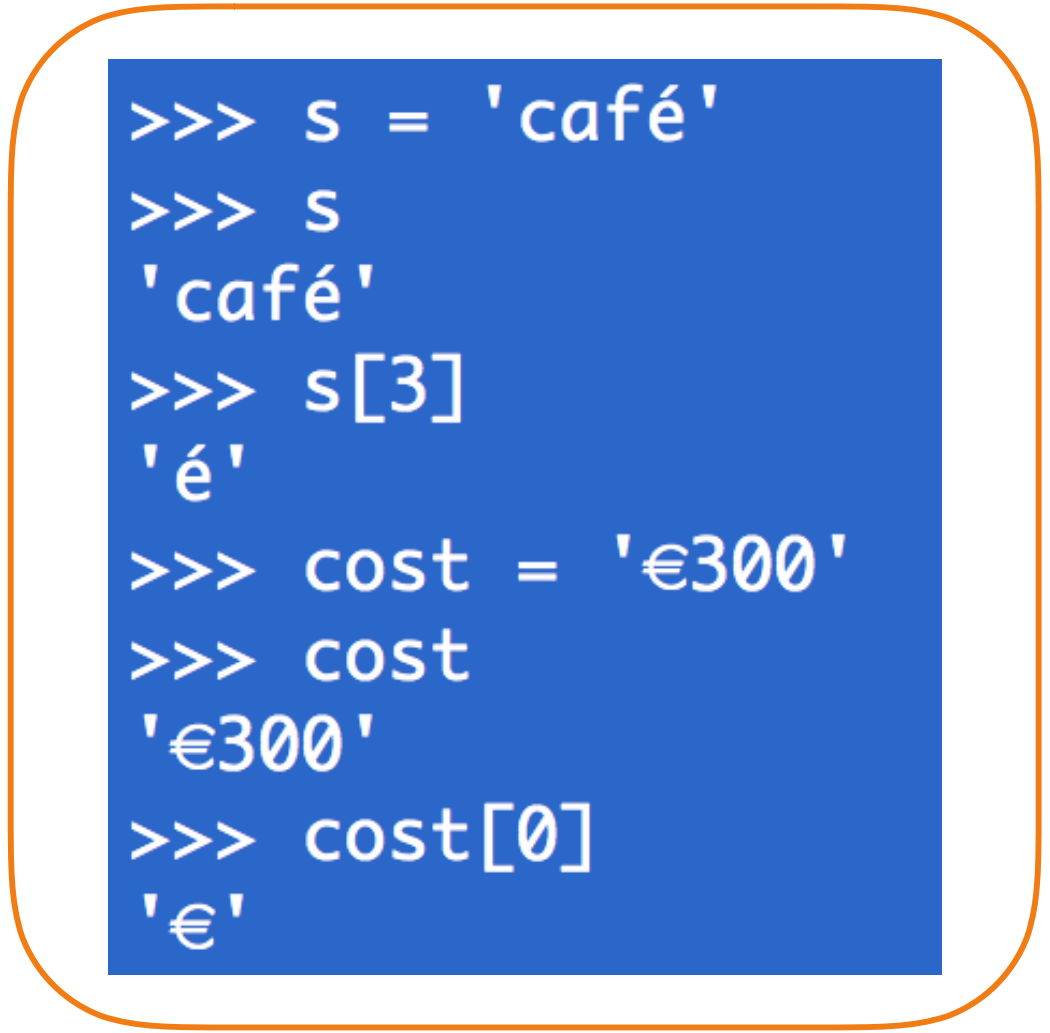
```
>>> s = 'café'
>>> s[3]
'\xc3'
>>> s = u'café'
>>> s[3]
u'\xe9'
```

strings: Python 2 vs. 3 (cont'd)

Develop Intelligence

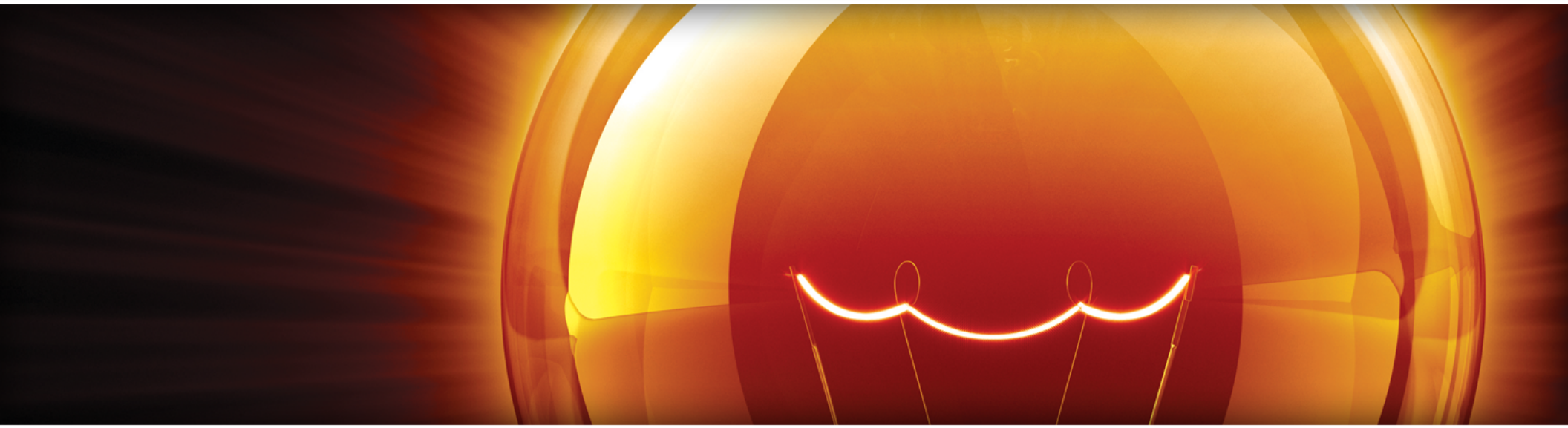
- in Python 3, all strings are Unicode by default
  - the identity of a character—its code point—is a number from 0 to 1,114,111 shown in the Unicode standard as 4-6 hex digits with a “U+” prefix, e.g.,
    - `A = U+0041`
    - `€ = U+20AC`
  - actual bytes that represent a character depend on the encoding in use, e.g.,
    - `€ = \xe2\x82\xac` in UTF-8, but `\xac\x20` in UTF-16LE

strings: Python 2 vs. 3 (cont'd)



```
>>> s = 'café'
>>> s
'café'
>>> s[3]
'é'
>>> cost = '€300'
>>> cost
'€300'
>>> cost[0]
'€'
```

Let's start writing some code!



(we will revisit strings and transition to more complicated datatypes soon)

lab: first standalone program

Develop Intelligence

- in IDLE, go to `File => New File`
- enter some Python code and save it
- execute it by going to `Run => Run Module`
- output will appear in Python shell window

first.py - /Users/dws/Python/first.py (3.5.0)

```
name = input("Enter your name: ")
print("You entered", name)
```

```
========== RESTART: /Users/dws/Python/first.py ==========
Enter your name: Dave
You entered Dave
```

lab: running from command line

Develop Intelligence

1. open a `Terminal` window and navigate to the file you saved via IDLE, e.g., `prog1.py`
2. to execute it, type `python prog1.py`
   (if you have multiple Python installations, you may have to type `python3 prog1.py`)

lab: shebang

1. using IDLE or a text editor such as `vi/vim`, `nano`, `sublime`, etc., add the following as the first line of your Python program

```
#!/usr/bin/env python3
```

2. open a `Terminal` window (if you aren't already in one) and navigate to directory containing the file `prog1.py` and type

```
chmod +x prog1.py
```

3. to run it, type

```
./prog1.py
```

2to3

Develop Intelligence

- automatically convert Python 2 code to Python 3

```
DWS-MacBook:py2 dws$ cat py2.py
print "Hello, world!"
DWS-MacBook:py2 dws$ 2to3 -w py2.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored py2.py
--- py2.py      (original)
+++ py2.py      (refactored)
@@ -1 +1 @@
-print "Hello, world!"
+print("Hello, world!")
RefactoringTool: Files that were modified:
RefactoringTool: py2.py
DWS-MacBook:py2 dws$ cat py2.py
print("Hello, world!")
```

lab: 2to3

1. put the following Python 2 code into a file

```
s = raw_input("Enter a string: ")
print "you entered", s
```

2. run it by passing it to the Python 2 interpreter
3. convert it to Python 3 using `2to3`
4. run it by passing it to the Python 3 interpreter

indentation

Develop Intelligence

- colons and indentation delineate blocks
  - no braces! (try typing **`from __future__ import braces`**)
- this will trip you up at first but once you're used to it, you'll love it

```
if x == 1:
    print("Hey, x is 1")
else:
    print("x is something other than 1")
```

indentation (cont'd)

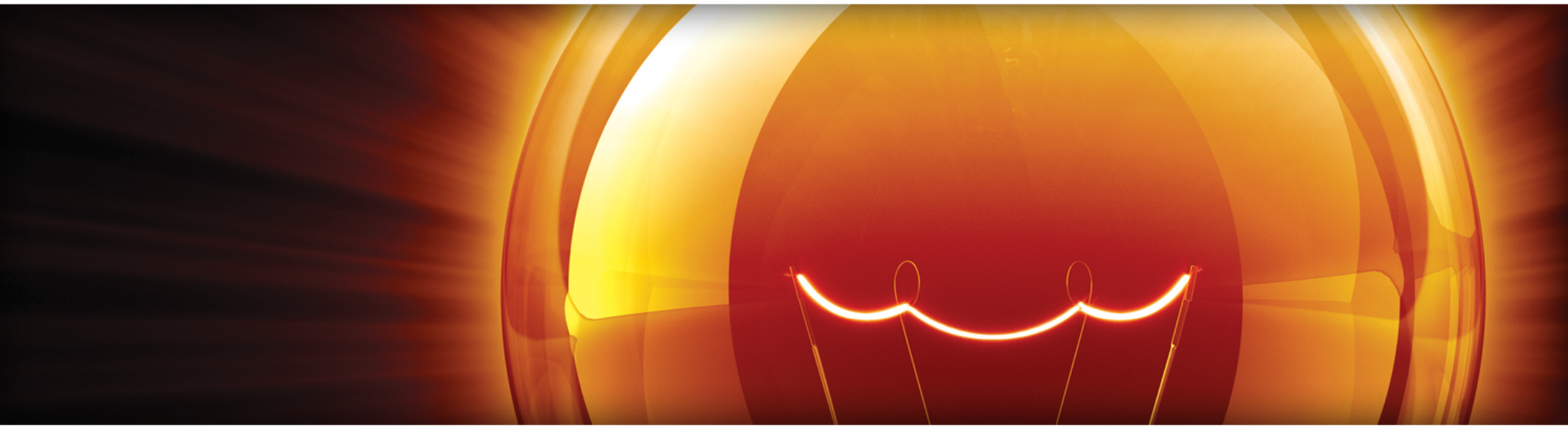
Develop Intelligence

- indentation must be consistent throughout the block

```
>>> if x == 1:
...      print("x is 1")
...     print("same block")
  File "<stdin>", line 3
    print("same block")
                      ^
IndentationError: unindent does not match any outer indentation level
```

- you can use any indentation you want as long as it's 4 spaces (PEP-8)
  https://www.python.org/dev/peps/pep-0008/

`if` statements



if statements

Develop Intelligence

- similar to `if` statements in other languages
- no parens needed
- `elif` = else if

```
my_number = 37
print("Enter your guess: ", end='')
guess = int(input())

if guess > my_number:
    print("Guess was too high")
elif guess < my_number:
    print("Guess was too low")
else:
    print("You got it!")
```

comparison operators

equality	`==`
inequality	`!=`
less than	`<`
less than or equal	`<=`
greater than	`>`
greater than or equal	`>=`
membership	`in ...`

comparison operators (cont'd)

Develop Intelligence

```
>>> x = 7
>>> 5 < x
True
>>> x < 10
True
>>> 5 < x and x < 10
True
>>> (5 < x) and (x < 10)
True
>>> 5 < x < 10
True
```

Loops

while loops

Develop Intelligence

```
# import random module ("batteries included")
import random
# generate a random int between 1 and 100
my_number = random.randint(1, 100)
guess = 0

# keep looping until user guesses the number or gives up
while guess != my_number:
    guess = int(input("Your guess (0 to give up)? "))
    if guess == 0:
        print("Sorry that you're giving up!")
        break
    elif guess > my_number:
        print("Guess was too high")
    elif guess < my_number:
        print("Guess was too low")
else:
    print("Congratulations. You guessed it!")
```

for loops

Develop Intelligence

- typically used to iterate through an *iterable* (list, string, and others we haven't learned yet) one element at a time

```
>>> for letter in 'string':
        print(letter)

s
t
r
i
n
g
```

for loops (cont'd)

Develop Intelligence

- sequences are also iterable

```
>>> for num in range(0, 5):
        print(num, end=', ')


0, 1, 2, 3, 4,
```

- `range()` is an immutable sequence of numbers in Python 3,
  - was a function which created a list in Python 2
  - now does what `xrange()` used to do in Python 2

for loops (cont'd)


```
>>> for num in range(0, 100, 3):
        print(num, end=', ')


0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48,
51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96,
99,
```

```
>>> for num in range(-5, 6):
        if num == 0:
            continue
        print(1 / num, end=', ')


-0.2, -0.25, -0.3333333333333333, -0.5, -1.0, 1.0, 0.5,
0.3333333333333333, 0.25, 0.2,
```

Lab: loops

1. Loop through the number from 2 to 25 and print out which numbers are prime, and for those number which are not prime numbers, you should print them as a product of two factors
2. Remember that prime = no divisors other than 1 and itself
3. Don't worry about efficiency, but if you're interested, check out `math.sqrt()`

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
20 equals 2 * 10
21 equals 3 * 7
22 equals 2 * 11
23 is a prime number
24 equals 2 * 12
25 equals 5 * 5
```

Lab: loops (solution)


```
import math

for num in range(2, 26):
    for i in range(2, round(math.sqrt(num)) + 1):
        if num % i == 0:
            j = num / i
            print('%d equals %d * %d' % (num, i, j))
            break
    else:
        print(num, 'is a prime number')
```

Lab: loops/strings

1. Have the user enter a string, then loop through the string to generate a new string which every character is duplicated, e.g., "hello" => "hheelllloo"

Lab: loops/strings (solution)

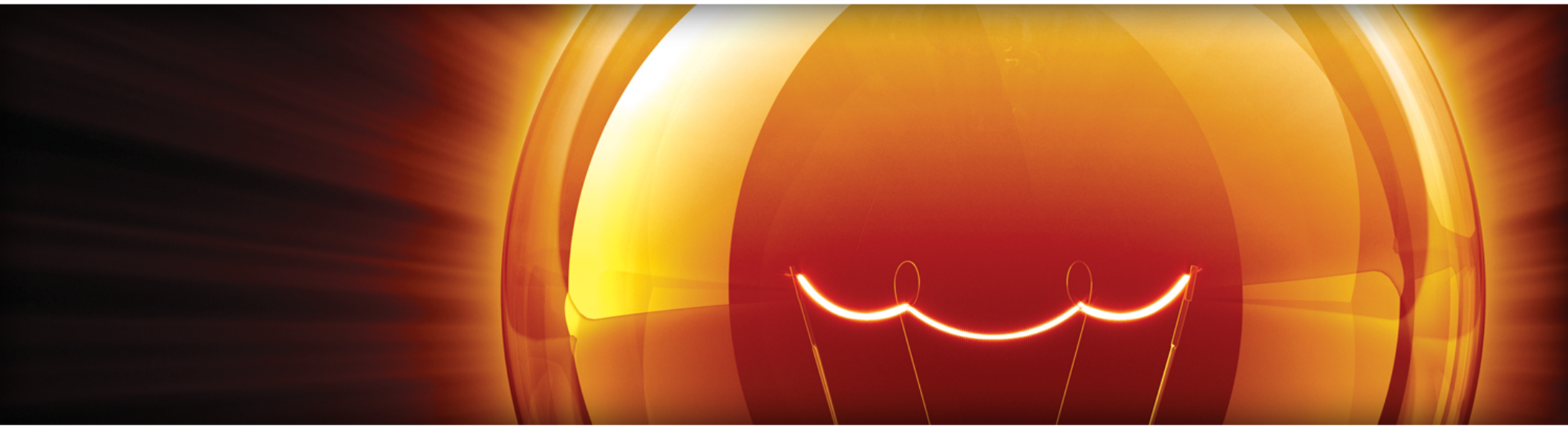

```
s = input("Enter a string: ")
t = ""

for c in s:
    t += c + c

print(t)
```

loops recap

Develop Intelligence

- `for` loop is more common
- `break` exits the loop immediately
- `continue` skips rest of loop and starts next iteration
- `else` is executed if loops terminates normally (i.e., no `break`)

Revisiting Strings



slices

Develop Intelligence

- [start:end:step]
  - extracts substring from the `start` offset to the `end` offset **minus 1**, skipping characters by `step`

```
>>> alphabet[10:15]
'klmno'
>>> alphabet[23:]
'xyz'
>>> alphabet[:5]
'abcde'
>>> alphabet[3:23:3]
'dgjmpsv'
>>> alphabet[::-1]
'zyxwvutsrqponmlkjihgfedcba'
>>> alphabet[-3:]
'xyz'
```

split/join


```
>>> "Now is the time".split()
['Now', 'is', 'the', 'time']
>>> "eggs, bread, milk, yogurt".split(', ')
['eggs', 'bread', 'milk', 'yogurt']
>>> ''.join(['anti', 'dis', 'establish', 'men',
'tarian', 'ism'])
'antidisestablishmentarianism'
>>> ', '.join(['Anne', 'Robert', 'Nancy'])
'Anne, Robert, Nancy'
>>>
```

more string functions


```
>>> poem = '''Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

Then took the other, as just as fair,
And having perhaps the better claim,
Because it was grassy and wanted wear;
Though as for that the passing there
Had worn them really about the same,

And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day!
Yet knowing how way leads on to way,
I doubted if I should ever come back.

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I-
I took the one less traveled by,
And that has made all the difference.'''
```

more string functions (cont'd)

Develop Intelligence

```
>>> poem[:17]
'Two roads diverge'
>>> len(poem)
729
>>> poem.startswith('Two')
True
>>> poem.endswith('And miles to go before I sleep.')
False
>>> poem.find('the')
163
>>> poem[163:175]
'the undergro'
>>> poem.rfind('the')
714
>>> poem.count('the')
12
>>> poem.isalnum()
False
```

strip()

Develop Intelligence

```
>>> s = '  Now is the time       '
>>> s.strip()
'Now is the time'
>>> s
'  Now is the time       '
>>> s = '.' + s.strip() + '...'
>>> s
'.Now is the time...'
>>> s.strip('.')
'Now is the time'
```

still more string functions

Develop Intelligence

```
>>> s = 'now is the time'
>>> s.capitalize()
'Now is the time'
>>> s.title()
'Now Is The Time'
>>> s.upper()
'NOW IS THE TIME'
>>> s.swapcase()
'NOW IS THE TIME'
>>> s.replace('the', 'not the')
'now is not the time'
>>> s.replace('t', 'T')
'now is The Time'
```

Lab: string functions

Develop Intelligence

- write a Python program which prompts the user for a string and a stride (increment), and alternately makes the string upper case and lower case, *stride* characters at a time, e.g.,

```
==================== RESTART: /Users/dws/Python/uplow.py ====================
Enter a string: abcdefghijklmnopqrstuvwxyz
Enter a stride: 1
AbCdEfGhIjKlMnOpQrStUvWxYz
>>>
==================== RESTART: /Users/dws/Python/uplow.py ====================
Enter a string: abcdefghijklmnopqrstuvwxyz
Enter a stride: 4
ABCDefghIJKLmnopQRSTuvwxYZ
>>>
==================== RESTART: /Users/dws/Python/uplow.py ====================
Enter a string: abcdeFGHIJklmnoPQRSTuvwxyZ
Enter a stride: 5
ABCDEfghijKLMNOpqrstUVWXYz
```

Lab: string functions (solution)

Develop Intelligence

```
s = input("Enter a string: ")
stride = int(input("Enter a stride: "))

up = True

for i in range(0, len(s), stride):
    if up:
        s = s[0:i] + s[i:i + stride].upper() + s[i + stride:]
        up = False
    else:
        s = s[0:i] + s[i:i + stride].lower() + s[i + stride:]
        up = True

print(s)
```

Lab: strings functions

1. Have the user enter two strings and indicate whether either string appears at the end of the other string, e.g.,

   "Bigelow", "low" => YES

   "mart", "Minimart" => YES

2. Extra…modify your solution for #1 so that case is ignored, i.e.,

   "Mart", "Minimart" => YES

Lab: string functions (solution)

Develop Intelligence

```
s1 = input("Enter a string: ")
s2 = input("Enter another string: ")

s1 = s1.lower()
s2 = s2.lower()

if s1.endswith(s2):
    print(s1, "ends with", s2)
elif s2.endswith(s1):
    print(s2, "ends with", s1)
```

Day 2

Day 2 Agenda

- lists
- tuples
- dictionaries
- sets
- list comprehensions
- sets
- file I/O
- functions
- scope global vs. local
- exceptions

Lists

lists

Develop Intelligence

- usually homogeneous, but can contain any objects
- duplicates allowed
- list() function creates a list from another sequence

```
>>> l = [1, 3, 5, 7, 5, 3, 1]
>>> l
[1, 3, 5, 7, 5, 3, 1]
>>> days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
>>> days
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

lists (cont'd)

Develop Intelligence

```
>>> today = '01/18/2016'
>>> today.split('/')
['01', '18', '2016']
>>> 'a/b//c/d//e//f'.split('/')
['a', 'b', '', 'c', 'd', '', 'e', '', 'f']
>>> i = input()
This is a test of the input function
>>> i.split()
['This', 'is', 'a', 'test', 'of', 'the', 'input', 'function']
```

lists (cont'd)


```
>>> stooges = [ 'Larry', 'Moe', 'Curly' ]
>>> stooges[1]
'Moe'
>>> stooges[-1]
'Curly'
>>> people = [stooges, 'Groucho']
>>> people
[['Larry', 'Moe', 'Curly'], 'Groucho']
>>> people[0]
['Larry', 'Moe', 'Curly']
>>> people[0][2]
'Curly'
>>> stooges[2] = 'Curley'
>>> stooges
['Larry', 'Moe', 'Curley']
>>> stooges[0:2]
['Larry', 'Moe']
>>> stooges[::2]
['Larry', 'Curley']
>>> stooges[::-1]
['Curley', 'Moe', 'Larry']
```

looping through a list

Develop Intelligence

```
>>> count = 0
>>> while count < len(stooges):
        print(stooges[count])
        count += 1

Larry
Moe
Curly
```

```
>>> for stooge in stooges:
        print(stooge)

Larry
Moe
Curly
```

lists: append/insert/extend

- append(): add an item to a list
- insert(): add an item by offset
- extend(), +=: add a list to a list

```
>>> stooges.append('Shemp')
>>> stooges
['Larry', 'Moe', 'Curley', 'Shemp']
>>> stooges.insert(3, 'Iggy')
>>> stooges
['Larry', 'Moe', 'Curley', 'Iggy', 'Shemp']
>>> others = ['Joe', 'Joe']
>>> stooges += others
>>> stooges
['Larry', 'Moe', 'Curley', 'Iggy', 'Shemp', 'Joe', 'Joe']
>>> stooges.append(others)
>>> stooges
['Larry', 'Moe', 'Curley', 'Iggy', 'Shemp', 'Joe', 'Joe', ['Joe', 'Joe']]
```

lists: del/remove/pop

- `del`: delete by position
- `remove(item)`: remove by value
- `pop()`: remove last item (or specified item)

```
>>> stooges
['Larry', 'Moe', 'Curley', 'Iggy', 'Shemp', 'Joe', 'Joe', ['Joe', 'Joe']]
>>> del stooges[-1]
>>> stooges
['Larry', 'Moe', 'Curley', 'Iggy', 'Shemp', 'Joe', 'Joe']
>>> stooges.remove('Iggy')
>>> stooges
['Larry', 'Moe', 'Curley', 'Shemp', 'Joe', 'Joe']
>>> stooges.remove('Joe')
>>> stooges
['Larry', 'Moe', 'Curley', 'Shemp', 'Joe']
>>> stooges.pop()
'Joe'
>>> stooges
['Larry', 'Moe', 'Curley', 'Shemp']
>>> stooges.pop(0)
'Larry'
>>> stooges
['Moe', 'Curley', 'Shemp']
```

lists: index/in/count

Develop Intelligence

- `index(item)`: return position of item
- `in`: test for membership
- `count(item)`: count occurrences of item

```
>>> stooges
['Larry', 'Moe', 'Curley', 'Shemp']
>>> stooges.index('Moe')
1
>>> 'Curly' in stooges
False
>>> 'Curley' in stooges
True
>>> stooges.append('Joe')
>>> stooges.append('Joe')
>>> stooges.count('Joe')
2
>>> stooges
['Larry', 'Moe', 'Curley', 'Shemp', 'Joe', 'Joe']
```

lists: join

Develop Intelligence

- combine list (or iterable sequence of strings) into string
- opposite of `split()`

```
>>> stooges
['Larry', 'Moe', 'Curley', 'Shemp', 'Joe', 'Joe']
>>> joined = ', '.join(stooges)
>>> joined
'Larry, Moe, Curley, Shemp, Joe, Joe'
>>> unjoined = joined.split(', ')
>>> unjoined
['Larry', 'Moe', 'Curley', 'Shemp', 'Joe', 'Joe']
>>> stooges == unjoined
True
```

lists: sort/sorted/len

Develop Intelligence

- `sort()`: sort a list in place
- `sorted()`: return a sorted copy of a list
- `len()`: return length of a list

```
>>> stooges
['Larry', 'Moe', 'Curley', 'Shemp', 'Joe', 'Joe']
>>> sorted(stooges)
['Curley', 'Joe', 'Joe', 'Larry', 'Moe', 'Shemp']
>>> stooges
['Larry', 'Moe', 'Curley', 'Shemp', 'Joe', 'Joe']
>>> stooges.sort()
>>> stooges
['Curley', 'Joe', 'Joe', 'Larry', 'Moe', 'Shemp']
>>> stooges.sort(reverse=True)
>>> stooges
['Shemp', 'Moe', 'Larry', 'Joe', 'Joe', 'Curley']
>>> len(stooges)
6
```

lists: assign vs. copy/list/slice

Develop Intelligence

- list assignment does NOT copy the values!
- use `copy()`, `list()` or `[:]` for copying

```
>>> stooges
['Shemp', 'Moe', 'Larry', 'Joe', 'Joe', 'Curley']
>>> s = stooges
>>> s[5] = 'Curly'
>>> stooges
['Shemp', 'Moe', 'Larry', 'Joe', 'Joe', 'Curly']
>>> s = stooges.copy()
>>> s
['Shemp', 'Moe', 'Larry', 'Joe', 'Joe', 'Curly']
>>> s[5] = 'Curley'
>>> stooges
['Shemp', 'Moe', 'Larry', 'Joe', 'Joe', 'Curly']
>>> s
['Shemp', 'Moe', 'Larry', 'Joe', 'Joe', 'Curley']
>>> s = list(stooges)
>>> s
['Shemp', 'Moe', 'Larry', 'Joe', 'Joe', 'Curly']
>>> s = stooges[:]
>>> s
['Shemp', 'Moe', 'Larry', 'Joe', 'Joe', 'Curly']
```

lab: lists

Develop Intelligence

- write a Python program to maintain two lists and loop until the user wants to quit
- your program should offer the user the following options:
  - add an item to list 1 or 2
  - remove an item from list 1 or 2 by value or index
  - reverse list 1 or list 2
  - display both lists
- EXTRA: add an option to check if lists are equal, even if contents are not in the same order (i.e, if list 1 is `[3, 2, 'apple', 4]` and list 2 is `[2, 3, 4, 'apple']`, you should indicate they are the same)

lab: lists (solution)


```
'''A list of lists. Only using lists[1] and lists[2]
so that list index matches what user calls it.'''
lists = [[], [], []]

while True:
    print("s) Show both lists")
    print("a) Add an item to a list")
    print("i) Remove an item from a list by index")
    print("v) Remove an item from a list by value")
    print("r) Reverse a list")
    print("e) Check for list equality")
    print("q) Quit")

    choice = input()

    if choice == '': # try again if user types nothi
        continue
    if choice == 'q':
        break
    if choice == 's':
        for i in range(1, 3):
            print("\nList", i, ":\n", lists[i])
        print("\n")
```

```
elif choice[0] == 'e':
    for i in range(1, 3):
        print("\nlist", i, lists[i])

    print("\nComparing the two lists...", end='')
    if sorted(lists[1]) != sorted(lists[2]):
        print("NOT ", end='')
    print("equal\n")
else:
    '''all other commands need a list number so
    we grab it once and store it in num'''
    num = int(choice[1])
    if choice[0] == 'a':
        lists[num].append(choice[2:])
    elif choice[0] == 'i':
        lists[num].pop(int(choice[2:]))
    elif choice[0] == 'v':
        lists[num].remove(choice[2:])
    elif choice[0] == 'r':
        lists[num] = lists[num][::-1]
```

"pythonic"

Develop Intelligence

```
>>> mylist = ['Larry', 'Moe', 'Curly']
>>>
>>> i = 0
>>> for val in mylist:
        print('index', i, 'is', val)
        i += 1


index 0 is Larry
index 1 is Moe
index 2 is Curly
```

NOT pythonic

enumerate

Develop Intelligence

- `enumerate()` is a builtin function which returns an *enumerate* object from any *iterable*

```
>>> for idx, val in enumerate(mylist):
        print("item", idx, "=", val)


item 0 = Larry
item 1 = Moe
item 2 = Curly
```

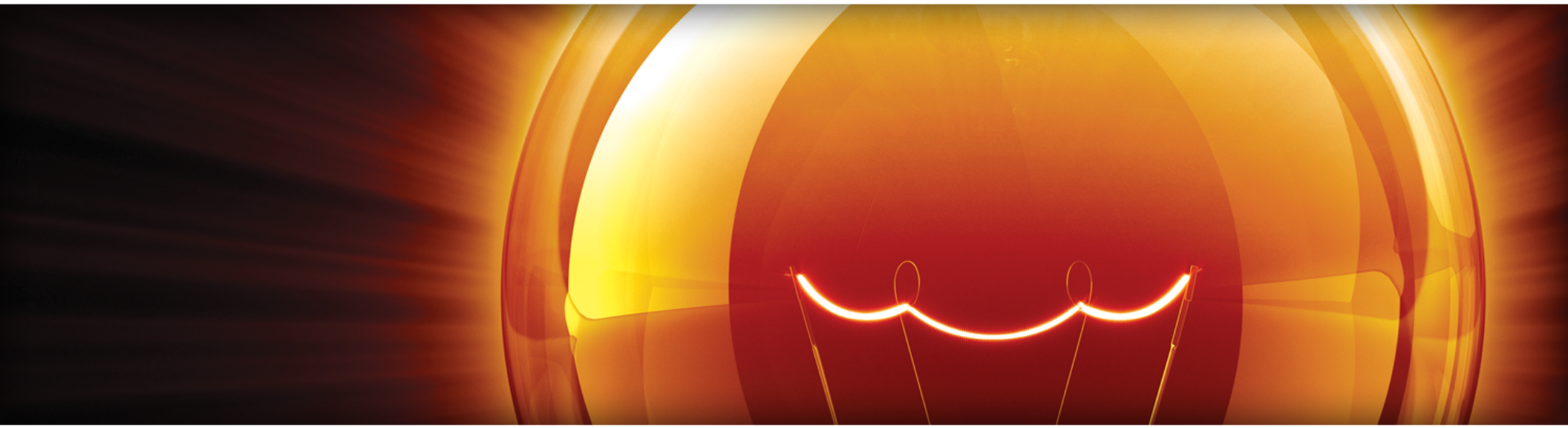
zip

Develop Intelligence

- `zip(*iterable)` is a builtin function which creates an iterator that aggregates elements from each iterable

```
>>> stooges = ['Larry', 'Moe', 'Curly']
>>> marxbros = ['Groucho', 'Harpo', 'Chico']
>>> for s, m in zip(stooges, marxbros):
        print(s, m)


Larry Groucho
Moe Harpo
Curly Chico
```

List Comprehensions



List Comprehensions ("listcomps")

Develop Intelligence

- quick way to build a list!
- more readable / faster
- which is easier to read?

```
>>> string = 'ABCabc*'
>>> ascii_codes = []
>>> for char in string:
        ascii_codes.append(ord(char))
```

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

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

- 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)

Develop Intelligence

- 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

1. Start with Cartesian product example (colors x sizes of t-shirts) and add a third list, `sleeves = ['short', 'long']` then write a new listcomp which generates the Cartesian product `colors` x `sizes` x `sleeves`. `tshirts` should look like this:

```
>>> 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']]
```

2. Use a list comprehension to create a list of the integers from 0 to 99.

Lab: listcomps (solution)


```
>>> tshirts = [[size, color, sleeve] for size in sizes
                                     for color in colors
                                     for sleeve in sleeves]
>>> tshirts
[['S', 'black', 'short'], ['S', 'black', 'long'], ['S', 'white', 'short'],
 ['S', 'white', 'long'], ['M', 'black', 'short'], ['M', 'black', 'long'],
['M', 'white', 'short'], ['M', 'white', 'long'], ['L', 'black', 'short'],
['L', 'black', 'long'], ['L', 'white', 'short'], ['L', 'white', 'long']]
>>>
>>> nums = [num for num in range(0, 100)]
>>> nums
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 3
9, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57,
 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 7
6, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94,
 95, 96, 97, 98, 99]
```

Tuples

tuples

Develop Intelligence

- immutable!
- typically heterogeneous (as opposed to lists)
- generally imply some structure

```
>>> t = ()
>>> t
()
>>> t = ('hello',)
>>> t
('hello',)
>>> t = 'Jones', 'Mary', 1023, True
>>> t
('Jones', 'Mary', 1023, True)
>>> last, first, employee_num, full_time = t # tuple unpacking
>>> employee_num
1023
>>> s = input()
Jones Mary 1023 True
>>> tuple(s.split())
('Jones', 'Mary', '1023', 'True')
```

singleton

parens are optional

tuples (cont'd)


```
>>> t = ('Gutzon Borglum', 'Idaho', 1867)
>>> t[1]
'Idaho'
>>> t[1] = 'Wyoming'
Traceback (most recent call last):
  File "<pyshell#258>", line 1, in <module>
    t[1] = 'Wyoming'
TypeError: 'tuple' object does not support item assignment
```

tuples (cont'd)

Develop Intelligence

can contain mutable objects

```
>>> t = ('Curie', 'Marie', 1867, [])
>>> t
('Curie', 'Marie', 1867, [])
>>> t[3].append("Jack Karen Dante Leanne".split())
>>> t
('Curie', 'Marie', 1867, [['Jack', 'Karen', 'Dante', 'Leanne']])
```

Lab: tuples

- Given a list of words, sort them by length of word, rather than alphabetically.
- To do this, first create a list of tuples of the form (len, word), where the first element is the length of the word.
- Next, sort the tuples.
- Finally, extract the words from the list of tuples into a new list which is now sorted by length of word. Try to use a list comprehension if you can.

Lab: tuples (solution)

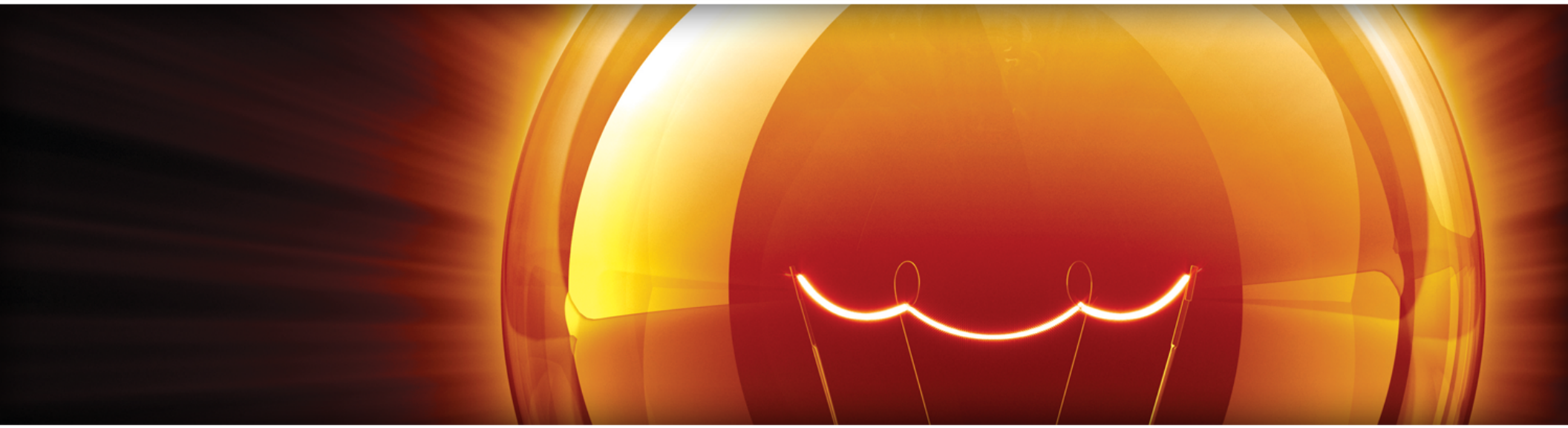

```
words = input("Enter a list of words: ")

'split string into a list of words'
wordlist = words.split()

'use a list comprehension to create a list of tuples (len, word)'
tuples = [(len(word), word) for word in wordlist]

'sort the list of tuples, which will sort by first element'
tuples = sorted(tuples)

'''
overwrite the wordlist with a new list created from the second
element of each tuple
'''
wordlist = [tuple[1] for tuple in tuples]

print("sorted by length of word: ", wordlist)
```

recap: tuples vs. lists

- not just "constant lists"
  - (see http://jtauber.com/blog/2006/04/15/python_tuples_are_not_just_constant_lists)
- remember that lists are (typically) ordered sequences of homogeneous values

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

Dictionaries



dictionaries

Develop Intelligence

- unordered list of key/value pairs
- "associative array"

```
>>> d = {} # empty dict
>>> d = { 'X' : 10, 'V' : 5, 'I' : 1 } # initialized
>>> d
{'X': 10, 'V': 5, 'I': 1}
>>> d['L'] = 50 # add an element later
>>> d
{'L': 50, 'X': 10, 'V': 5, 'I': 1}
>>> for numeral in d:
        print(numeral, end=' ')


L X V I
```

dictionaries (cont'd)

Develop Intelligence

```
>>> mydict = { 'tall': 12, 'grande': 16, 'venti': 20 }
>>> mydict
{'tall': 12, 'venti': 20, 'grande': 16}
>>> mydict.values()
dict_values([12, 20, 16])
>>> mydict.keys()
dict_keys(['tall', 'venti', 'grande'])
>>> total = 0
>>> for amount in mydict.values():
        total += amount


>>> total
48
```

dictionaries: keys/values/items

- `keys()`, `values()`, and `items()` are *view* objects
- unlike lists, they provide a dynamic window into dictionary

```
>>> k = mydict.keys()
>>> k
dict_keys(['tall', 'venti', 'grande'])
>>> mydict['trenta'] = 31
>>> k
dict_keys(['tall', 'venti', 'trenta', 'grande'])
>>>
>>> l = list(k)
>>> l
['tall', 'venti', 'trenta', 'grande']
>>> mydict.items()
dict_items([('tall', 12), ('venti', 20), ('trenta', 31), ('grande', 16)])
>>> l = list(mydict.items())
>>> l
[('tall', 12), ('venti', 20), ('trenta', 31), ('grande', 16)]
```

dictionaries: enumerate

Develop Intelligence

```
>>> for index, value in enumerate(mydict):
        print('index', index, '=', value)


index 0 = tall
index 1 = venti
index 2 = trenta
index 3 = grande
```

zip redux

Develop Intelligence

```
>>> for key, value in zip(mydict.keys(), mydict.values()):
        print(key, '=>', value)


tall => 12
venti => 20
trenta => 31
grande => 16
```

- above is same as `for item in mydict.items()`

dictionaries: sorted example

- `sorted(iterable[, key][, reverse])` is a builtin function which returns a sorted copy of an iterable
  - `key` = function specifying sort key, default = `None`

```
>>> for key in sorted(mydict, key=mydict.get):
        print(key, '=>', mydict[key])


tall => 12
grande => 16
venti => 20
trenta => 31
```

Lab: dictionaries

Develop Intelligence

- Write a Python program to maintain a dictionary. Your program should continuously loop while offering the user the following options:
  - Erase dictionary (i.e., start with an empty dict)
  - Add an item
  - Delete an item by key
  - Delete an item by value
- To delete an item by value you may iterate through the items, note when you find what you are looking for and `pop()`–we will learn a more Pythonic solution soon

Lab: dictionaries (solution)


```
mydict = {}

while True:
    print('''
e) erase dict
a) add item to dict ('key,value')
d) delete item by key
r) delete item by value''')
    i = input()
    if i == '': # if nothing entered, try again...
        continue
    if i[0] == 'e': # ...so we can ensure i[0] is valid
        mydict = {}
    elif i[0] == 'a':
        'Split input on comma and use results as key and value'
        key, val = i[1:].split(',')
        mydict[key] = val
    elif i[0] == 'd':
        '''Rest of input line is key, so grab it and delete
        item. Use get() function in case key does not exist'''
        if mydict.get(i[1:]):
            mydict.pop(i[1:])
        else:
            print("no key", i[1:])
    elif i[0] == 'r':
        '''Rest of input is val, so grab it and then search
        through dict looking for a key which matches value.
        Remeber mydict.items() is a *view* object, so we can't
        modify the dict during the loop. Other way to do it
        is to put items into a list,...list(mydict.items())
        and then it's OK to modify the dict in a loop.'''
        found = False
        for key, val in mydict.items():
            if val == i[1:]:
                found = True
                break
        '''If found is True then it means we found key whose
        value is that which we want to delete.'''
        if found:
            mydict.pop(key)

print("\n", mydict)
```

Sets

sets (cont'd)

Develop Intelligence

- unordered collection, no duplicates

```
>>> s = { 'Anne', 'Betty', 'Cathy' }
>>> s
{'Betty', 'Anne', 'Cathy'}
>>> s.add('Lola')
>>> s
{'Betty', 'Anne', 'Cathy', 'Lola'}
>>> s.add('Anne')
>>> s
{'Betty', 'Anne', 'Cathy', 'Lola'}
>>> if 'Anne' in s:
        print("Yep!")
Yep!
```

sets (cont'd)

Develop Intelligence

```
>>> even = set(range(2, 11, 2))
>>> odd = set(range(1, 10, 2))
>>> even
{8, 2, 10, 4, 6}
>>> odd
{1, 3, 5, 9, 7}
>>> prime = {2, 3, 5, 7}
>>> prime & odd
{3, 5, 7}
>>> prime & even
{2}
>>> odd | even
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
>>> prime - even
{3, 5, 7}
>>> prime ^ odd
{1, 2, 9}
```

intersection

union

difference

symmetric difference

sets + dicts

Develop Intelligence

```
>>> movies = {
        'Die Hard' : { 'Bruce Willis', 'Alan Rickman', 'Bonnie Bedelia' },
        'The Sixth Sense' : { 'Donnie Wahlberg', 'Toni Colette', 'Bruce Willis' },
        'Hunt for Red October' : { 'Sean Connery', 'Alec Baldwin' },
        'The Highlander' : { 'Christopher Lambert', 'Sean Connery'},
        '16 Blocks': { 'Bruce Willis', 'Yasiin Bey', 'David Morse' }
        }
```

```
>>> for title, stars in movies.items():
        if 'Bruce Willis' in stars:
            print(title)


Die Hard
The Sixth Sense
16 Blocks
```

sets + dicts (cont'd)

Develop Intelligence

```
>>> movies = {
        'Die Hard' : { 'Bruce Willis', 'Alan Rickman', 'Bonnie Bedelia' },
        'The Sixth Sense' : { 'Donnie Wahlberg', 'Toni Colette', 'Bruce Willis' },
        'Hunt for Red October' : { 'Sean Connery', 'Alec Baldwin' },
        'The Highlander' : { 'Christopher Lambert', 'Sean Connery'},
        '16 Blocks': { 'Bruce Willis', 'Yasiin Bey', 'David Morse' }
        }
```

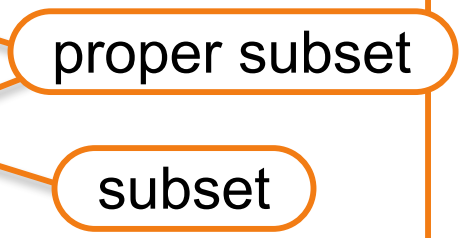
```
>>> for title, stars in movies.items():
        if stars & {'Alan Rickman', 'Sean Connery'}:
            print(title)


Die Hard
The Highlander
Hunt for Red October
```

(sub)sets

Develop Intelligence

- proper subset is a subset of that is not equal to

```
>>> set1 = { 1, 2, 3 }
>>> set2 = { 1, 2, 3, 5, 7, 9 }
>>> set1 < set1
False
>>> set1 <= set1
True
>>> set1 < set2
True
>>> set1.issubset(set1)
True
>>> set1.issubset(set2)
True
```



Lab: sets

Develop Intelligence

- Write a Python program which will display this menu:
  1. Add a boy name
  2. Add a girl name
  3. Print union
  4. Print intersection
  5. Check for membership
  6. Quit
- The program should maintain two lists, a list of boy names and a list of girl names
- Enter some names in both lists (e.g., "Lee", "Morgan")
- Exercise all 6 options
- Some Python constructs you should use: `input()`, sets, set operators, `while` loop, `break`

Lab: sets (solution)


```
boys, girls = set(), set()

while True:
    print('''
b) Add a boy name
g) Add a girl name
u) Print union
i) Print intersection
m) Check for membership
q) Quit''')
    i = input()
    if i == '':
        continue
    if i[0] == 'q':
        break
    elif i[0] == 'b':
        boys.add(i[1:]) # add rest of input line
    elif i[0] == 'g':
        girls.add(i[1:]) # add rest of input line
    elif i[0] == 'u':
        print(boys | girls) # union
    elif i[0] == 'i':
        print(boys & girls) # intersection
    elif i[0] == 'm':
        if i[1] == 'b':
            print(i[2:] in boys) # membership
        else:
            print(i[2:] in girls) # membership

print("\nGirls:", girls)
print("Boys:", boys)
```

sets recap

- unordered
- no duplicates
- operators `&`, `|`, `-`, `^`
- use `in` to test for membership
- subset vs. proper subset

File I/O

File I/O

Develop Intelligence

- `fileobj = open(filename, mode)`
  - `mode` is one or two letters
    - `r = read`
    - w = write (create/overwrite)
    - x = write, but only if file does not already exist
    - a = append, if the file exists.
  - second letter =
    - t = text file (default)
    - b = binary
- `fileobj.close()`

File I/O: open/close


```
>>> f = open("/tmp/test.txt", "r")
Traceback (most recent call last):
  File "<pyshell#685>", line 1, in <module>
    f = open("/tmp/test.txt", "r")
FileNotFoundError: [Errno 2] No such file or directory: '/tmp/test.txt'
>>> f = open("/tmp/test.txt", "w")
>>> f.close()
>>> f = open("/tmp/test.txt", "x")
Traceback (most recent call last):
  File "<pyshell#688>", line 1, in <module>
    f = open("/tmp/test.txt", "x")
FileExistsError: [Errno 17] File exists: '/tmp/test.txt'
```

File I/O: read/write

Develop Intelligence

```
>>> len(poem)
730
>>> f = open("/tmp/poem.txt", "w")
>>> f.write(poem)
730
>>> f.close()
>>> f = open("/tmp/poem.txt", "r")
>>> poem2 = f.read()
>>> f.close()
>>> poem == poem2
True
```

File I/O: write vs. print


```
>>> len(poem)
730
>>> f = open("/tmp/poem.txt", "w")
>>> print(poem, file=f)
>>> f.close()
>>> f = open("/tmp/poem.txt", "r")
>>> poem2 = f.read()
>>> f.close()
>>> poem == poem2
False
>>> len(poem2)
731
```

File I/O: print

- `print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`
  - `sep` = separator (default is space)
  - `end` = what to print at end (default is newline)
- `file` = where to print, default is screen
- `flush` = whether to flush output buffer, default is no

File I/O: ways to read data

Develop Intelligence

- `read()` slurps up entire file at once
  - `read(x)` reads at most x bytes
- `readline()` reads a line at a time
- `readlines()` reads a line at a time and returns the lines as a list of strings
- or use an iterator…

```
>>> poem = ''
>>> f = open("/tmp/poem.txt", "r")
>>> for line in f:
        poem += line


>>> len(poem)
731
```

File I/O: with

- the `with` statement sets up a temporary "context "and closes the file automatically so we don't have to bother with closing it

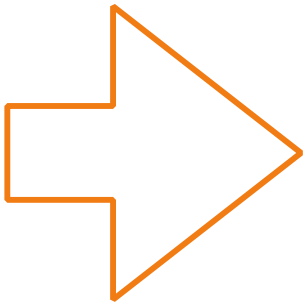
```
>>> with open("/tmp/poem.txt", "r") as f:
        poem2 = f.read()
        if not f.closed:
                print("in with, file is not yet closed")


in with, file is not yet closed
>>> f.closed
True
>>> poem == poem2
True
```

Lab: File I/O

Develop Intelligence

- write a Python program which prompts the user for a filename, then opens that file and writes the contents of the file to a new file, in reverse order, i.e.,

```
original file
==============
1 First line
2 Second line
3 Third line
4 Fourth line
```



```
reversed file
==============
4 Fourth line
3 Third line
2 Second line
1 First line
```

Lab: File I/O plus dicts

- write a Python program to read a file and count the number of occurrences of each word in the file
- use a dict, indexed by word, to count the occurrences
- if your dict is d, then `d.get(key)` will return None if there is no such key in the dict (`d[key]` will throw an exception)
- your program should treat 'The' and 'the' as the same word for counting purposes
- print out the words and their counts, from least common to most common
- Road Not Taken is available at https://github.com/davewadestein/Python-Core

Lab: File I/O (solution)


```
with open("/tmp/test.txt", "r") as f1:
    with open("/tmp/test2.txt", "w") as f2:
        lines = f1.readlines()

        for line in lines[::-1]:
            f2.write(line)

'''
or

        for line in reversed(lines):
            f2.write(line)
'''
```

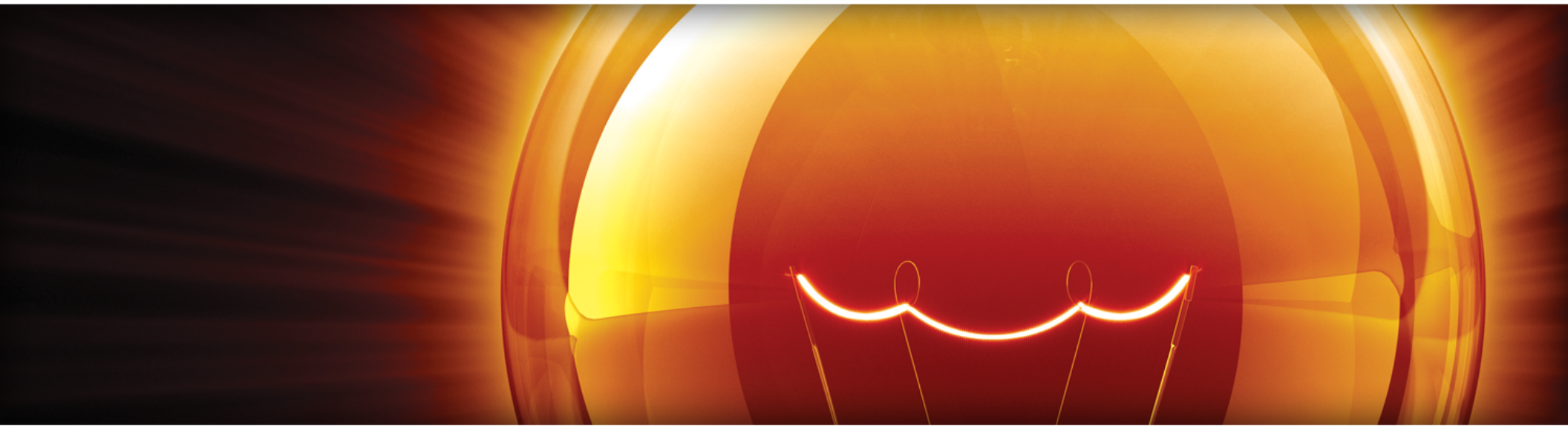
Lab: File I/O (solution)


```
with open("/tmp/test.txt", "r") as f1:
    with open("/tmp/test2.txt", "w") as f2:
        lines = f1.readlines()

        for line in lines[::-1]:
            f2.write(line)

'''
or

        for line in reversed(lines):
            f2.write(line)
'''
```

Lab: File I/O (solution)


```
import string

ex = set(string.punctuation)
count = {}

for line in open('poem.txt'):
        line = line.lower()
        line = ''.join(ch for ch in line if ch not in ex)
        for word in line.split():
            if count.get(word):
                count[word] += 1
            else:
                count[word] = 1

for key in sorted(count, key=count.get, reverse=True):
    print(key, count[key])
```

Day 3



Day 3 Agenda

- functions
- exceptions
- decorators
- command line arguments
- modules
- developer modules
  - os, sys, subprocess
  - shutil, glob
- StringIO
- regular expressions
- OO programming: classes/class decorators

functions

Develop Intelligence

- `def` introduces a function, followed by function name, parenthesized list of args and then a colon
- body of function is indented

```
>>> def noop():
        pass

>>> noop()
>>> noop(1)
Traceback (most recent call last):
  File "<pyshell#836>", line 1, in <module>
    noop(1)
TypeError: noop() takes 0 positional arguments but 1 was given
```

do nothing

functions (cont'd)

Develop Intelligence

```
>>> 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(-1)
hey, x is less than 10 and not 1
```

functions (cont'd)

Develop Intelligence

```
def rounder25(amount):
    '''
    Return amount rounded UP to nearest quarter dollar
        ...$1.89 becomes $2.00
        ...but $1.00/$1.25/$1.75/etc. remain unchanged
    '''
    dollars = int(amount)
    cents = round((amount - dollars) * 100)
    quarters = cents // 25
    if cents % 25:
        quarters += 1
    amount = dollars + 0.25 * quarters

    return amount
```

docstring

functions (cont'd)

- `help(func)` prints out formatted docstring
- `func.__doc__` prints out raw docstring

```
>>> help(rounder25)
Help on function rounder25 in module __main__:

rounder25(amount)
    Return amount rounded UP to nearest quarter dollar
        ...$1.89 becomes $2.00
        ...but $1.00/$1.25/$1.75/etc. remain unchanged

>>> rounder25.__doc__
'\n    Return amount rounded UP to nearest quarter dollar\n        ...$1.89 beco
mes $2.00\n        ...but $1.00/$1.25/$1.75/etc. remain unchanged\n    '
>>>
>>> rounder25(2.04)
2.25
>>> rounder25(2.26)
2.5
>>> rounder25(2.91)
3.0
```

functions (cont'd)

Develop Intelligence

- if function doesn't call return explicitly, the special value `None` is returned
- `None` is like `NULL` in other languages
- not the same as `False`

```
>>> def noop():
        pass
```

```
>>> thing = noop()
>>> print(thing)
None
>>> if thing:
        print("some thing")
else:
        print("no thing")


no thing
>>> if thing is True:
        print("True")
elif thing is False:
        print("False")
elif thing is None:
        print("None")


None
```

functions: positional arguments

Develop Intelligence

- arguments are passed to functions in order written
- downside: you must remember meaning of each position

```
>>> def menu(wine, entree, dessert):
        return {'wine': wine, 'entree': entree, 'dessert': dessert}

>>> menu(
        (wine, entree, dessert)
```

- outside an IDE, it can be difficult to remember
- if you pass args in wrong order, bad things can happen!

```
>>> menu('chianti', 'tartufo', 'polenta')
{'entree': 'tartufo', 'wine': 'chianti', 'dessert': 'polenta'}
```

functions: keyword arguments

Develop Intelligence

- you may specify arguments by name, in any order
- once you specify a keyword argument, all arguments following it must be keyword arguments

```
>>> menu('chianti', dessert='tartufo', entree='polenta')
{'entree': 'polenta', 'wine': 'chianti', 'dessert': 'tartufo'}
```

```
>>> menu(wine='chianti', dessert='tartufo', 'polenta')
SyntaxError: positional argument follows keyword argument
```

functions: default arguments


```
>>> def menu(wine, entree, dessert='tartufo'):
        return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

```
>>> menu('chardonnay', 'braised tofu')
{'entree': 'braised tofu', 'wine': 'chardonnay', 'dessert': 'tartufo'}
>>> menu('chardonnay', dessert='cannoli', entree='fagioli')
{'entree': 'fagioli', 'wine': 'chardonnay', 'dessert': 'cannoli'}
```

lab: functions

1. modify the `rounder25()` function to take an additional argument which specifies the increment to round up to (e.g., `rounder(1.37, 10)` would round $1.37 up to the next dime, or $1.40)
2. write a function `calculate` which is passed two operands and an operator and returns the calculated result, e.g. `calculate(2, 4, '+')` would return 6

lab: functions (solution)

Develop Intelligence

```
def rounder(amount, inc):
    '''
    Return amount rounded UP to nearest
    increment.
        ...$1.89 becomes $2.00
        ...but $1.XX/$1.XX/$1.XX, where
           XX is a multiple of the increment
           remain unchanged.
    '''
    dollars = int(amount)
    cents = round((amount - dollars) * 100)
    coins = cents // inc
    if cents % inc:
        coins += 1
    amount = dollars + (inc / 100) * coins

    return amount
```

variable positional arguments

- sometimes we want to a function which takes a variable number of arguments (e.g., builtin `print()` function)

```
>>> def func(*args):
        print("the args are", args)
```

```
>>> func()
the args are ()
>>> func(1, 2, 3)
the args are (1, 2, 3)
>>> func([1, 2, 3], "hello", True)
the args are ([1, 2, 3], 'hello', True)
>>> func(1)
the args are (1,)
>>> func('this is a test'.split())
the args 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 args. 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
```

Lab: variable positional arguments (solution)


```
def product(*args):
    '''Return the product of the args passed in'''
    result = 1
    for term in args:
        result *= term
    return result
```

variable keyword arguments

- what if a function needs a bunch of configuration options, having default values which typically aren't overridden?
  - one way to do this would be to have the function accept a dict in which these value(s) can be specified
- better way is to use variable keywords arguments

```
>>> def vka(**kwargs):
        for key in kwargs:
            print(key, "=", kwargs[key])
```

```
>>> vka(debug=True, x=5, color='red')
x = 5
color = red
debug = True
```

Lab: variable keyword arguments

- modify your `calculate` function by adding variable keywords arguments to it and checking whether `float = True`, and if so, the calculation should be done as floating point, rather than integer (of course this could be done with a default argument value, but don't do that)

```
>>> calculate(2, 4, '+')
6
>>> calculate(2, 4, '+', float=True)
6.0
```

Lab: variable keyword arguments (solution)


```
def calculate(operand1, operand2, operator, **kwargs):
    float = False

    if kwargs.get('float'):
        operand1 = float(operand1)
        float = True

    if operator == '+':
        return operand1 + operand2
    elif operator == '-':
        return operand1 - operand2
    elif operator == '*':
        return operand1 * operand2
    elif operator == '/':
        if float:
            return operand1 / operand2
        else:
            return operand1 // operand2
```

functions: scope/global

Develop Intelligence

```
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?

functions: recap

- Python encourages functions which support lots of arguments with default values
- ***"Explicit is better than implicit"***
  - arguments can be passed out of order ONLY if they're passed by keyword
  - keywords are more explicit than positions because the function call documents the purpose of its arguments
- variable positional args (*args)
- variable keyword args (**kwargs)

exceptions

- errors detected during execution are called *exceptions*
- exceptions are "thrown" and either "caught" by an exception handler, or propagated upward
- "…exceptions create hidden control-flow paths that are difficult for programmers to reason about" –Weimer & Necula, "Exceptional Situations and Program Reliability"

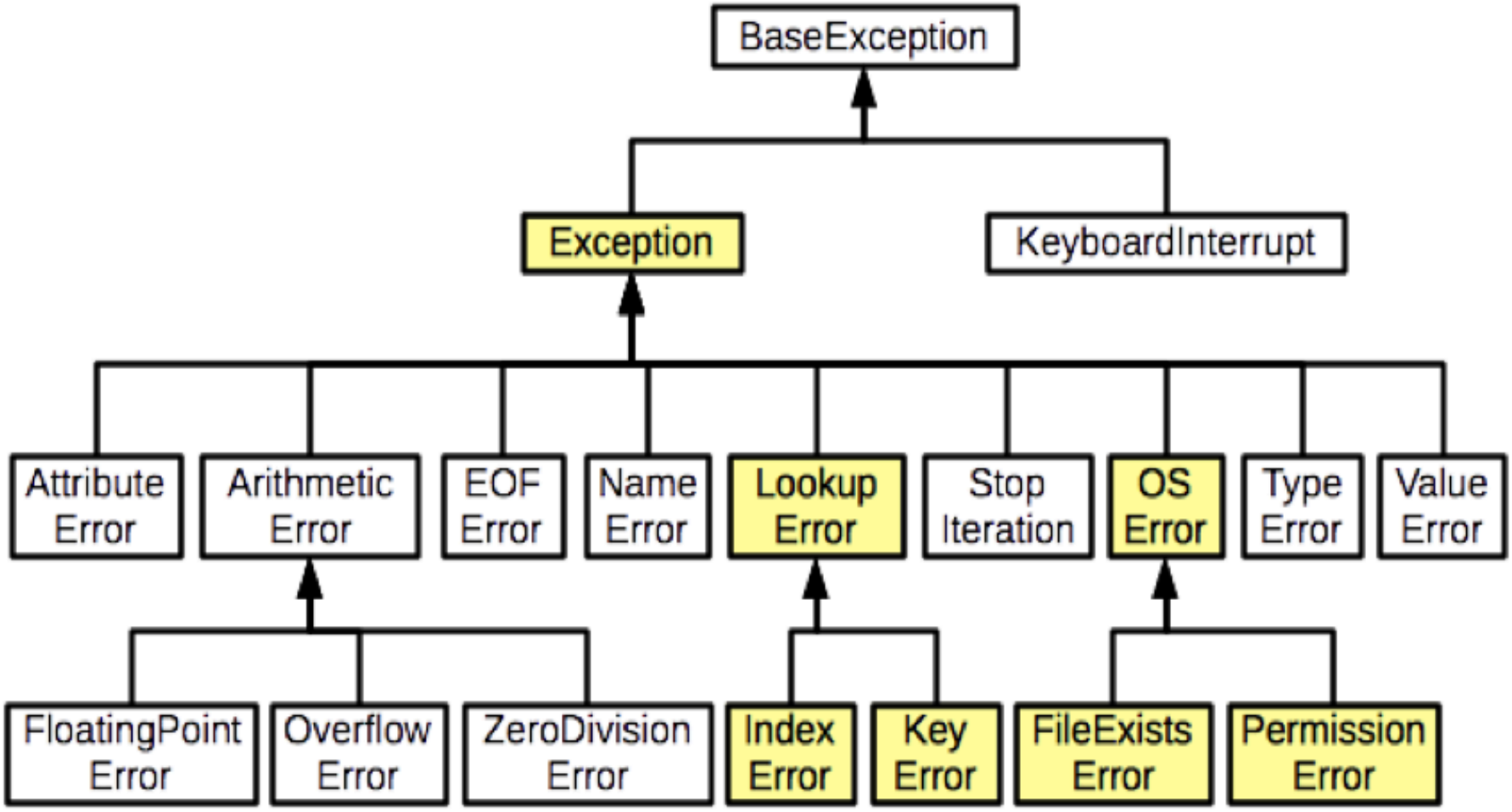
exceptions (cont'd)


```
>>> mylist = [1, 5, 14]
>>> mylist[0]
1
>>> mylist[5]
Traceback (most recent call last):
  File "<pyshell#119>", line 1, in <module>
    mylist[5]
IndexError: list index out of range
```

```
>>> int('13.5')
Traceback (most recent call last):
  File "<pyshell#124>", line 1, in <module>
    int('13.5')
ValueError: invalid literal for int() with base 10: '13.5'
```

exceptions (cont'd)

Develop Intelligence



exceptions: try/except

- `try` block wraps code which may throw an exception, and `except` block catches exception

```
>>> try:
        mylist[5]
except:
        print("oops, there is no element at offset 5")


oops, there is no element at offset 5
```

- problem? above example catches ALL exceptions, not just `IndexError` we are expecting

exceptions: try/except (cont'd)

Develop Intelligence

- best practice is to catch expected exceptions and let unexpected ones through, so as to avoid hidden errors

```
>>> mylist = [1, 5, 14]
>>> try:
        mylist[1]
        int('a')
except IndexError:
        print('Bad index, try again!')
except Exception as uhoh:
        print('Some other exception:', uhoh)


5
Some other exception: invalid literal for int() with base 10: 'a'
```

exceptions: try/except (cont'd)

Develop Intelligence

```
short_list = [1, 2, 3]

while True:
    value = input('Position [q to quit]? ')

    if value == 'q':
        break
    try:
        position = int(value)
        print(short_list[position])
    except IndexError:
        print('Bad index:', position)
    except Exception as other:
         print('Something else broke:', ot
```

```
Position [q to quit]? 0
1
Position [q to quit]? 2
3
Position [q to quit]? 3
Bad index: 3
Position [q to quit]? two
Something else broke: invalid literal
for int() with base 10: 'two'
Position [q to quit]? q
>>>
```

lab: exceptions

Develop Intelligence

- modify your `calculate` function to catch the `ZeroDivisionError` exception and print an informative message if the user tries to divide by zero, e.g.,

```
>>> calculate(4, 2, '/')
2
>>> calculate(4, 0, '/')
You cannot divide by zero!
0
```

lab: exceptions (solution)

Develop Intelligence

```
def calculate(operand1, operand2, operator):
    if operator == '+':
        return operand1 + operand2
    elif operator == '-':
        return operand1 - operand2
    elif operator == '*':
        return operand1 * operand2
    else:
        try:
            return operand1 / operand2
        except ZeroDivisionError:
            print("No divide by zero!")
            return 0
```

exceptions: (cont'd)

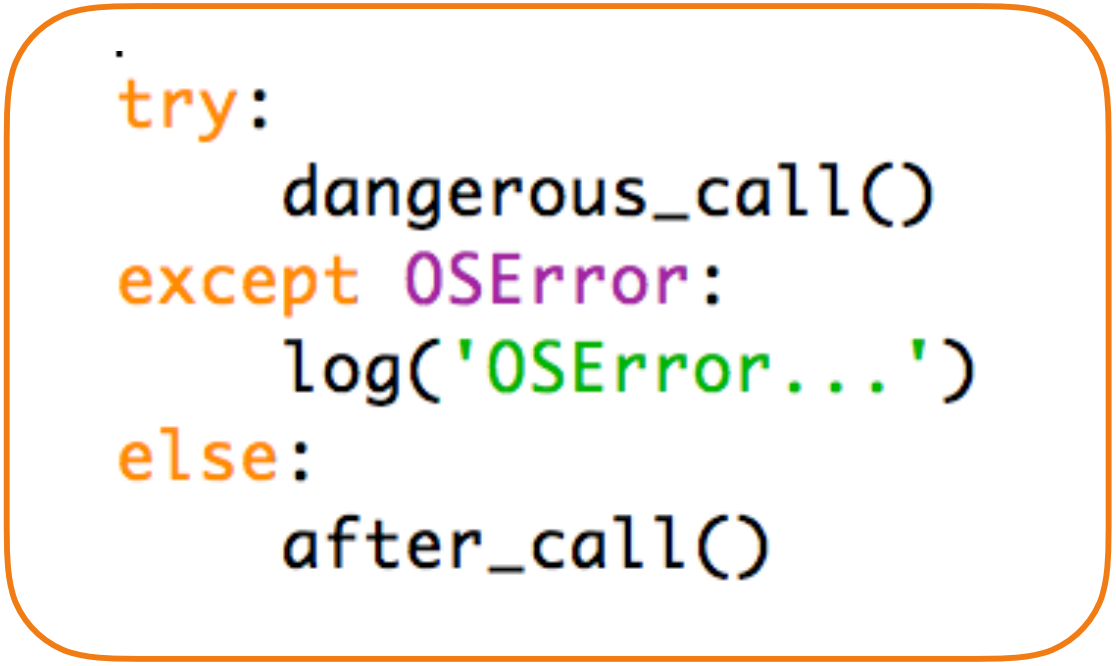
Develop Intelligence

- important to minimize size of try block

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

- `after_call()` will only run if `dangerous_call()` doesn't throw an exception…So what's the problem?

try/else (cont'd)




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

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

lab: exceptions

Develop Intelligence

- modify the exception handler in your `calculate` function to include an `else` block and move code to the `else` block except code which may throw an exception
- extend your calculator to include a log() function where the second argument is the base, i.e,. `calculate(49.0, 7, 'log')` = $\log_7(49.0) = 2.0$
- be sure you have a `try/except/else` block for your `log()` function
- (remember that $\log_b(x) = \log_a(x)/\log_a(b)$)

lab: exceptions (solution)


```
def calculate(operand1, operand2, operator):
    if operator == '+':
        return operand1 + operand2
    elif operator == '-':
        return operand1 - operand2
    elif operator == '*':
        return operand1 * operand2
    else:
        try:
            result = operand1 / operand2
        except ZeroDivisionError:
            print("No divide by zero!")
            return 0
        else:
            return result
```

lab: exceptions (solution 2)

Develop Intelligence

```
elif operator == 'log':
    from math import log
    base = operand2
    try:
        return log(operand1) / log(base)
    except ZeroDivisionError:
        print("There is no such thing as base 1!")
    except ValueError:
        print("Can't take log(0.0)!")

    return 0.0
```

variables in Python

Develop Intelligence


```
>>> x = 1
>>> id(x)
4297537952
>>> x = 2
>>> id(x)
4297537984
>>> id(1)
4297537952
>>> id(2)
4297537984
```

pass by assignment

Develop Intelligence

```
>>> def func(x):
        x.append('new')
        x = [4, 5, 6]
        print(x)


>>> l = [1, 2, 3]
>>> func(l)
[4, 5, 6]
>>> l
[1, 2, 3, 'new']
```

Decorators

Decorators

- 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
- boilerplate code that needs to be executed before or after what the function actually does

Decorators (cont'd)

- to discuss 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)

Develop Intelligence

```
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)

Develop Intelligence

```
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

- create a decorator called `square_it()` which squares the result
- since you can have more than one decorator for a function, apply both the `square_it()` and the `document_it()` decorators to the `add_ints()` function
- run `add_ints()`, then change the order of decorators and run again…the end result remains the same, but the intermediate result should be different.
- which decorator runs first?

Lab: Decorators (solution)


```
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 square_it(func):
    def inner_func(*args):
        result = func(*args)
        return result ** 2
    return inner_func

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

print(add_ints(2, 3))
```

Command Line Arguments

command line arguments

- when we run Python programs directly, we often want to pass arguments to the program

```
import sys
print('Program arguments:', sys.argv)
```

```
DWS-MacBook:Python dws$ python3 args.py
Program arguments: ['args.py']
DWS-MacBook:Python dws$
DWS-MacBook:Python dws$ python3 args.py hi there 1 2 3
Program arguments: ['args.py', 'hi', 'there', '1', '2', '3']
```

command line arguments


```
import sys

for idx, arg in enumerate(sys.argv):
    print("arg %d is %s" % (idx, arg))
```

```
DWS-MacBook:Python dws$ python args2.py hello 1 2 there
arg 0 is args2.py
arg 1 is hello
arg 2 is 1
arg 3 is 2
arg 4 is there
```

lab: command line arguments

Develop Intelligence

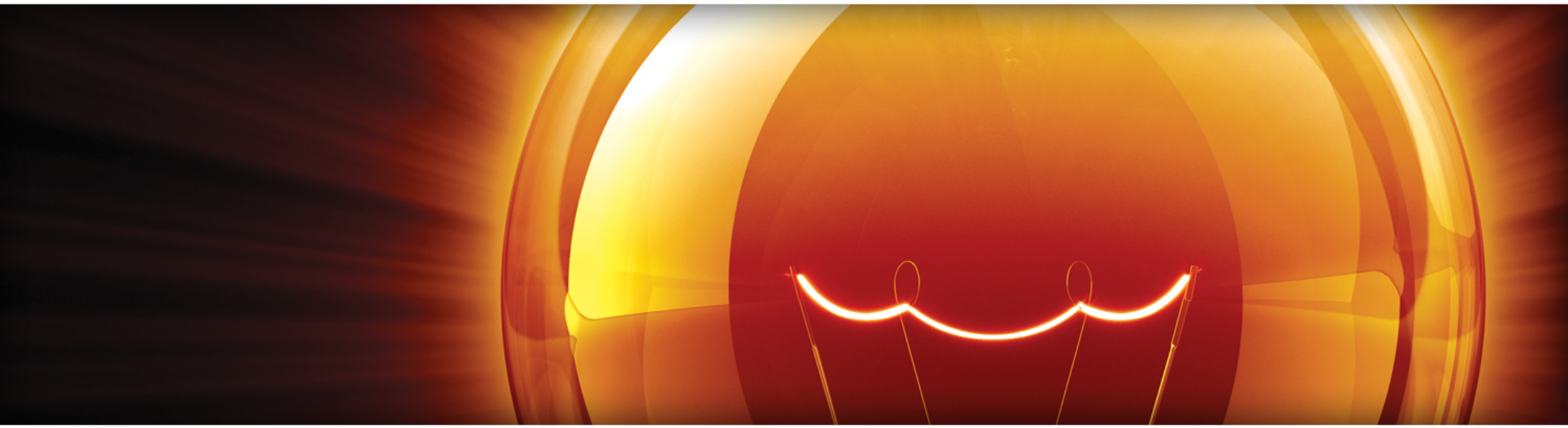
- turn your `calculate()` function into a standalone program which takes 3 command line arguments and invokes `calculate()` with those arguments

```
DWS-MacBook:Python dws$ python3 calculate.py 2 4 +
6
DWS-MacBook:Python dws$
DWS-MacBook:Python dws$ python3 calculate.py 8 7 /
1.1428571428571428
```

lab: command line arguments (solution)


```
def calculate(operand1, operand2, operator, **kwargs):
    float = False

    if kwargs.get('float'):
        operand1 = float(operand1)
        operand2 = float(operand2)
        float = True
    else:
        operand1 = int(operand1)
        operand2 = int(operand2)

    if operator == '+':
        return operand1 + operand2
    elif operator == '-':
        return operand1 - operand2
    elif operator == '*':
        return operand1 * operand2
    elif operator == '/':
        if float:
            return operand1 / operand2
        else:
            return operand1 // operand2

if __name__ == '__main__':
    import sys
    print(calculate(sys.argv[1], sys.argv[2], sys.argv[3]))
```

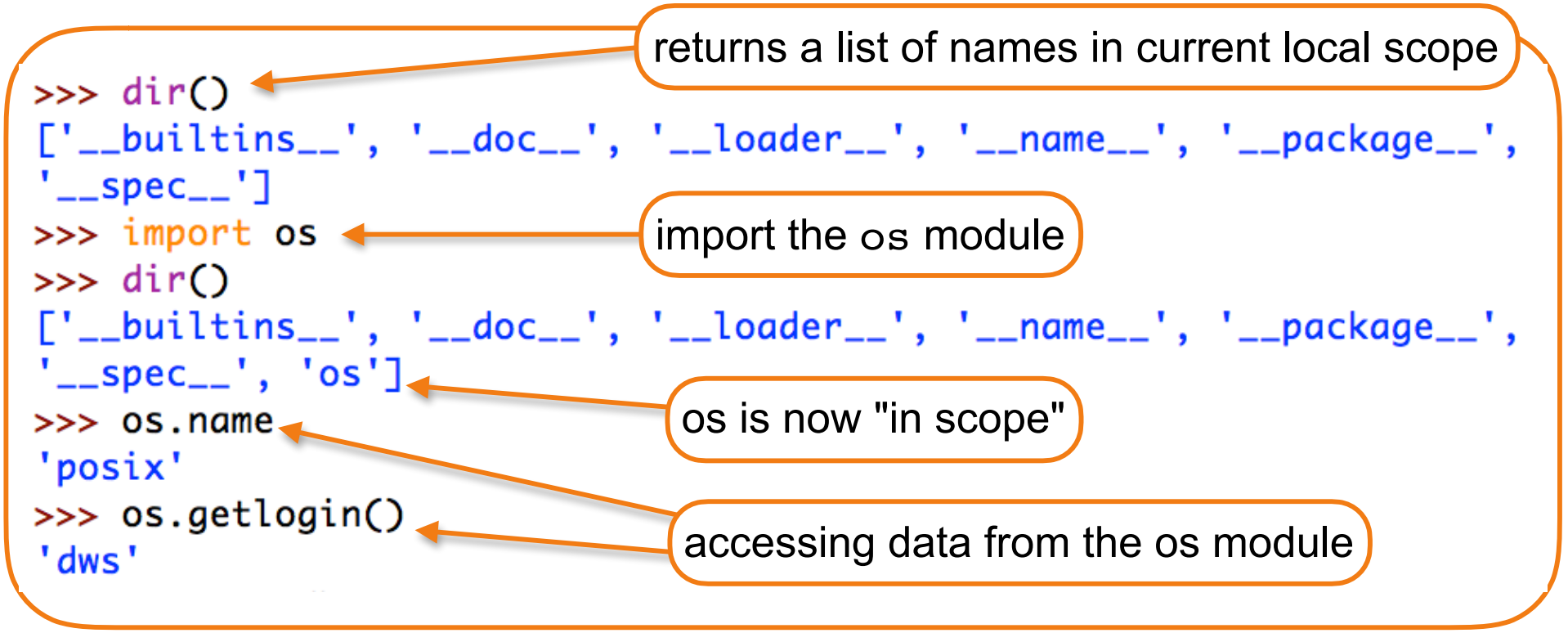
Modules



modules

Develop Intelligence

- files of Python code which "expose" functions, data, and classes (we'll be working with classes shortly)



modules (cont'd)

- two ways to import
  1. `import module`
  2. `from module import something`

     `from module import *`
- imported stuff can be renamed

```
import numpy as np
from sys import argv as foo
```

modules: from vs. import


```
def dummy():
    print("in dummy function")

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

print("in mymodule")
```

private data is prefaced with an underscore (_)

modules: from vs. import


```
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
>>> from mymodule import *
in mymodule
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'dummy', 'public_data']
>>> dummy()
in dummy function
>>> _private_data
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    _private_data
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!

modules: from vs. import

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

```
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
>>> import mymodule
in mymodule
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
 'mymodule']
>>> mymodule.public_data
'public stuff!'
>>> mymodule._private_data
'private stuff!'
```

private data is accessible

lab: modules

Develop Intelligence

1. create your own module, `mymodule.py` (or any name you choose) and import it from IDLE or the Python shell using both `from` and `import` syntax
   - be sure you are understand how to access variables/ data from your imported modules and the difference between `from mymodule` and `import mymodule`
2. take your `calculate.py` program and split it into two files: a module which contains the `calculate` function, and a main program which imports the `calculate` module

lab: modules (solution)

Develop Intelligence

```
from calculate_module import calculate

if __name__ == '__main__':
    import sys
    print(calculate(sys.argv[1], sys.argv[2], sys.argv[3]))
```

…and `calculate_module.py`, in the same directory, contains the calculate function

modules: search path

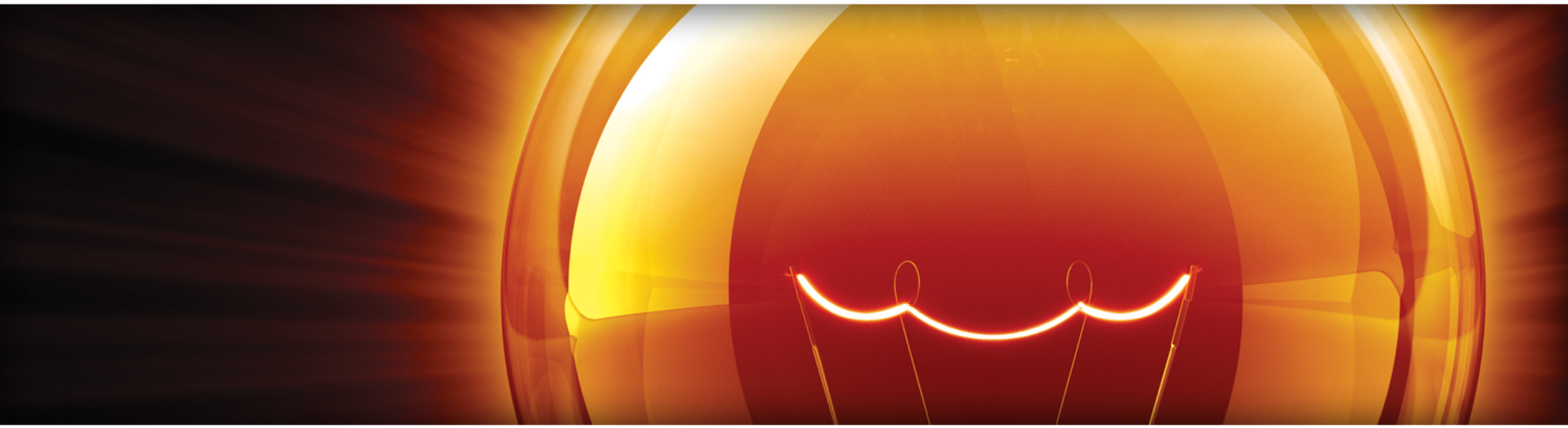
- where does Python look for modules?

```
>>> import sys
>>> for place in sys.path:
        print('"%s"' % place)


""
"/Users/dws/Python"
"/Library/Frameworks/Python.framework/Versions/3.5/bin"
"/Library/Frameworks/Python.framework/Versions/3.5/lib/python35.zip"
"/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5"
"/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/plat-darwin"
"/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/lib-dynload"
"/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages"
```

modules: recap

- modules are just files of Python code
- two ways to import: `from module import stuff` and `import module`
- private data is not really private!
- packages are directories containing one or more Python modules

StringIO



StringIO

Develop Intelligence

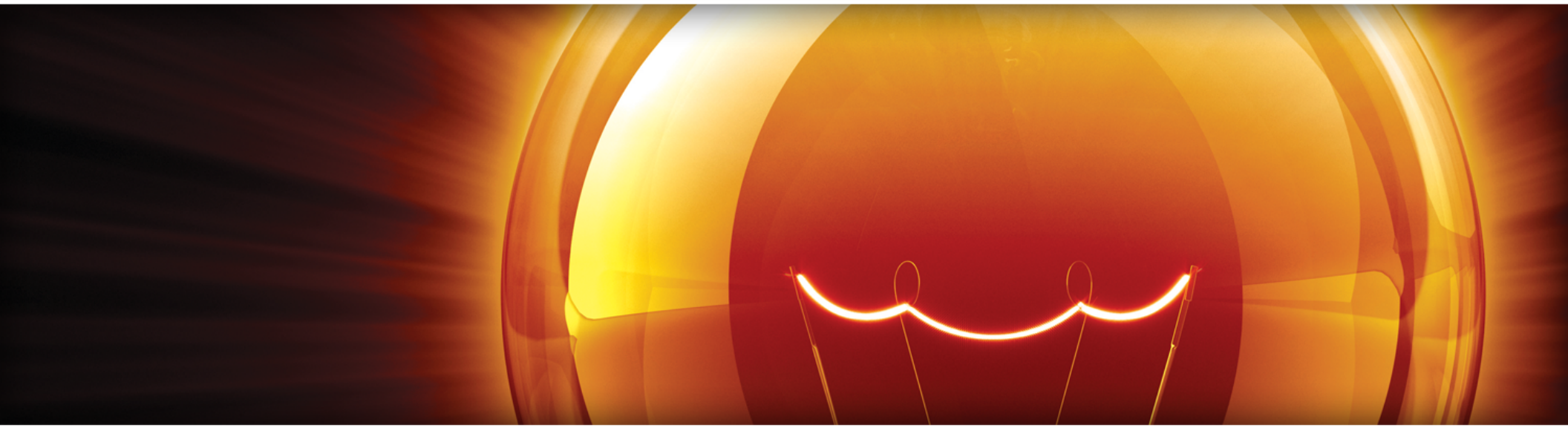
- a way to use file I/O in memory, rather than to a file, i.e., you can use `read()/write()` to a memory buffer–why would you want to do this?
- in Python 2, `StringIO` was in the `StringIO` module
- Python 3 did away with the `StringIO` module and the functionality is now in the `io` module
- let's handle both Python 2 and Python 3…

```
try:
    from StringIO import StringIO
except ImportError:
    from io import StringIO
```

StringIO (cont'd)


```
>>> from io import StringIO
>>> buffer = StringIO()
>>> buffer.write('Two roads diverged in a yellow wood')
35
>>> buffer.getvalue()
'Two roads diverged in a yellow wood'
>>> buffer.close()
>>> buffer = StringIO('Two roads diverged in a yellow wood')
>>> buffer.read()
'Two roads diverged in a yellow wood'
>>> buffer.close()
```

Regular Expressions



regular expressions

Develop Intelligence

- special sequence of characters that helps you find specific text sequences in strings, files, etc.
- "wildcard" characters take the place of a group of characters

```
>>> import re
>>> re.match("alpha", "alphabet")
<_sre.SRE_Match object; span=(0, 5), match='alpha'>
>>> re.match("bet", "alphabet")
>>>
>>> re.search("bet", "alphabet")
<_sre.SRE_Match object; span=(5, 8), match='bet'>
>>> re.search("bets", "alphabet")
>>>
```

re special characters

`.` = any character except newline

`^` = beginning of line/string

`$` = end of line/string

`*` = 0+ of the preceding RE

`+` = 1+ of the preceding RE

`?` = 0 or 1 instances of preceding RE

`{n}` = exactly n instances of the preceding RE

`[]` = match character set or range, e.g., `[aeiou]`, `[a-z]`, etc.

`(…)` = matches the RE inside the parens, and creates a *group*

Let's try some of these using regex101.com

regular expressions (cont'd)

Develop Intelligence

```
>>> import re
>>> re.match('a.*b', 'alphabet')
<_sre.SRE_Match object; span=(0, 6), match='alphab'>
>>> if re.match('a.*b', 'alphabet'):
...    print("match found!")
...
match found!
>>>
>>> if re.match('l.*b', 'alphabet'):
...    print("match found!")
...
>>>
```

need to `import re` in order to use regular expressions

look for 'a' followed by 0+ characters, followed by 'b' in the string 'alphabet'

look for 'l' followed by 0+ characters, followed by 'b' in the string 'alphabet'

why not found?

regular expressions (cont'd)

Develop Intelligence

```
>>> 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'
```

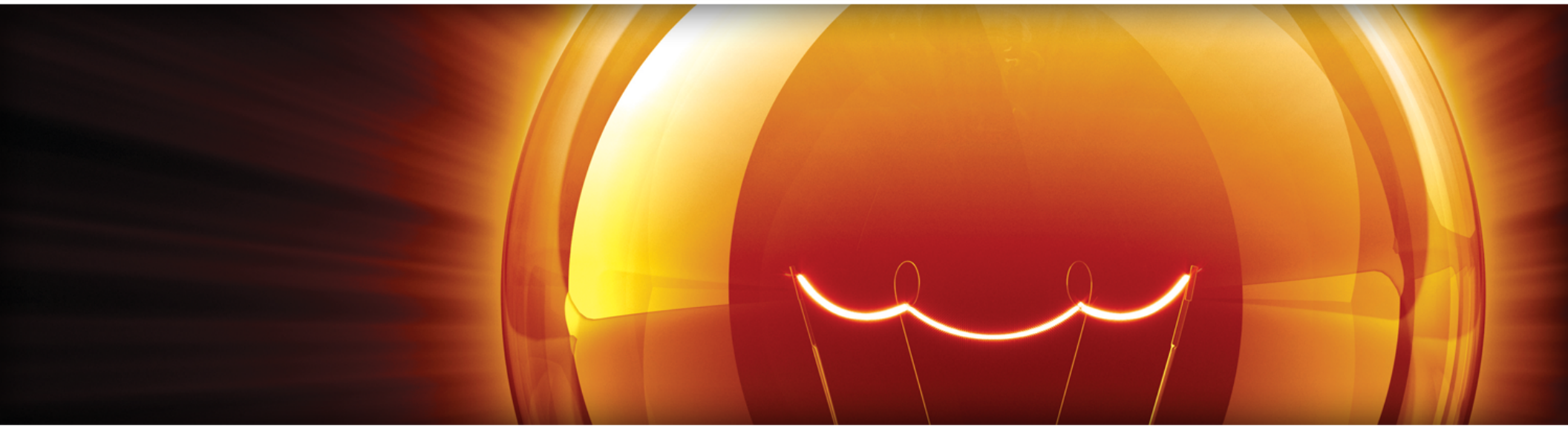
`re.match()` anchors the search to the beginning of the string, whereas `re.search()` will find a pattern anywhere

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

Lab: regular expressions

- write a Python program which takes two command line arguments, a filename and a regex pattern
- your program should act like `grep` in that it should search for the pattern in each line of the file
- if the pattern matches a given line, print out the line

Developer Modules



os module

Develop Intelligence

- operating system specific module
- dealing with files, directories, etc.
- running commands outside of Python
- let's see and try a few examples…

```
>>> import os
>>> os.system('ls')
0
>>> os.system('touch newfile')
0
>>> os.system('ls')
newfile
0
>>> os.getcwd()
'/Users/dws/Python/os'
>>> os.path.exists('newfile')
True
>>> os.mkdir('newdir')
>>> os.path.isfile('newdir')
False
>>> os.path.isdir('newdir')
True
```

sys module

- system-specific parameters and functions
- we've already seen some examples, `argv` and `path`

```
>>> import sys
>>> sys.path
['', '/Library/Frameworks/Python.framework/Versions/3.5/lib/python35.zip', '/Library/F
rameworks/Python.framework/Versions/3.5/lib/python3.5', '/Library/Frameworks/Python.fr
amework/Versions/3.5/lib/python3.5/plat-darwin', '/Library/Frameworks/Python.framework
/Versions/3.5/lib/python3.5/lib-dynload', '/Library/Frameworks/Python.framework/Versio
ns/3.5/lib/python3.5/site-packages']
>>> sys.maxsize
9223372036854775807
>>> 2 ** 63 - 1
9223372036854775807
>>> sys.exit(34)
DWS-MacBook:os dws$ echo $?
34
```

shutil module

Develop Intelligence

- used for high-level file operations

```
>>> import shutil
>>> import os
>>> os.system('ls -F')
newdir/          newfile
0
>>> shutil.copy('newfile', 'newfileCOPY')
'newfileCOPY'
>>> os.system('ls -F')
newdir/          newfile          newfileCOPY
0
>>> shutil.move('newfileCOPY', 'newerfile')
'newerfile'
>>> os.system('ls -F')
newdir/          newerfile        newfile
0
```

glob module

Develop Intelligence

- `glob()` function matches file or directory names using Linux shell rules rather than regular expression syntax

```
>>> import glob
>>> glob.glob('n*')
['newdir', 'newerfile', 'newfile']
>>> glob.glob('*e')
['newerfile', 'newfile']
>>> glob.glob('???')
[]
>>> import os
>>> os.system('touch abc')
0
>>> glob.glob('???')
['abc']
```

subprocess module

Develop Intelligence

- supplants os.system()/os.spawn(), both of which used to be standard way to run programs outside of Python

```
>>> import subprocess
>>> ret = subprocess.getoutput('date')
>>> ret
'Wed Jan  6 18:00:29 MST 2016'
>>> ret = subprocess.getoutput('ls')
>>> ret
'abc\nnewdir\nnewerfile\nnewfile'
>>> print(ret)
abc
newdir
newerfile
newfile
```

argparse module

- replacement for optparse module, which was deprecated in Python 2.7
- for Python 3, use argparse
- not surprisingly, it parses command line arguments and options (arguments which begin with - or --)

argparse example

Develop Intelligence

```
import argparse

parser = argparse.ArgumentParser(description='salesforce Core Python')

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)
parser.add_argument('-v', action='version', version='%(prog)s 1.0')

args = parser.parse_args(['-a', '-bsomething', '-c', '3'])

'''parse args from command line
args = parser.parse_args()
'''

print(args)

if args.a:
    print("-a was passed")
if args.b:
    print("-b", args.b, "was passed")
if args.c:
    print("-c", args.c, "was passed (int)")
```

create an arg parser, description used in help text

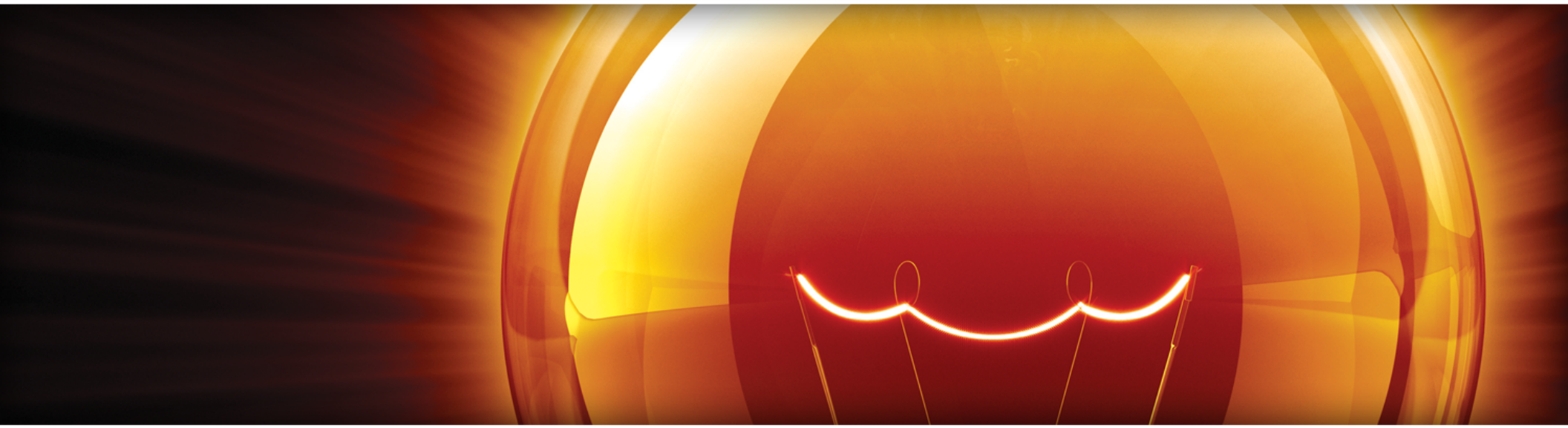
add arguments you want to be able to parse

parse command line args from a list for testing

Lab: argparse

- modify the previous lab to use -f to specify the filename and -p to specify the pattern
- also add -v or --version as an option
- if you have time add a -c ("context") option which will print the preceding and following line for each line that matches

OO Programming/Classes



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

classes

Develop Intelligence

- so far we've looked at built-in types; now we're doing to define a new type
- class = programmer-defined type

```
>>> class Person():
        pass

>>> Person
<class '__main__.Person'>
>>> somebody = Person()
>>> somebody
<__main__.Person object at 0x105a62c88>
```

defining the class creates a class object

to create a Person, we call it as if it were a function

- creating a new object is called *instantiation*
- the object is an instance of the class

classes: "magic" methods

Develop Intelligence

- __init__ is a special initialization method that is invoked when the object is instantiated
  - `self` is an argument which represents the object itself, can be any name, but `self` is customary
- __str__ returns a string representation of the object (i.e., for humans)
- __repr__ return unambiguous representation of the object

```
>>> import datetime
>>> today = datetime.datetime.now()
>>> str(today)
'2012-03-14 09:21:58.130922'
>>> repr(today)
'datetime.datetime(2012, 3, 14, 9, 21, 58, 130922)'
```

classes (cont'd)

Develop Intelligence

```
>>> class Person():
        def __init__(self, name):
            self.name = name


>>> stooge = Person("Shemp Howard")
>>> stooge
<__main__.Person object at 0x105a62e48>
>>> stooge.name
'Shemp Howard'
```

classes: properties

Develop Intelligence

```
class Duck():
    '''getters/setters example using properties'''
    def __init__(self, input_name):
        self.hidden_name = input_name

    def get_name(self):
        'getter for name attribute'
        print('inside the getter')
        return self.hidden_name

    def set_name(self, input_name):
        'setter for name attribute'
        print('inside the setter')
        self.hidden_name = input_name

    name = property(get_name, set_name)
```

traditional getter/setter

defined to be properties of the name attribute

classes: name attribute


```
>>> fowl = Duck('Donald')
>>> fowl.name
inside the getter
'Donald'
>>> fowl.get_name()
inside the getter
'Donald'
>>> fowl.name = 'Daffy'
inside the setter
>>> fowl.name
inside the getter
'Daffy'
```

classes: properties/decorators

Develop Intelligence

```
class Duck():
    '''getters/setters example using decorators'''
    def __init__(self, input_name):
        self.hidden_name = input_name

    @property
    def name(self):
        'getter for name attribute'
        print('inside the getter')
        return self.hidden_name

    @name.setter
    def name(self, input_name):
        'setter for name attribute'
        print('inside the setter')
        self.hidden_name = input_name
```

define getter/setter properties with decorators

classes: not hidden!

Develop Intelligence

```
>>> fowl = Duck('Donald')
>>> fowl.name
inside the getter
'Donald'
>>> fowl.name = 'Daffy'
inside the setter
>>> fowl.name
inside the getter
'Daffy'
>>> fowl.hidden_name
'Daffy'
```

we no longer have visible get_name() and set_name() methods

…but hidden_name can still be accessed from outside

classes: computed values

Develop Intelligence

```
class Circle():
    def __init__(self, radius):
        self.radius = radius

    @property
    def diameter(self):
        return 2 * self.radius
```

```
>>> c.radius = 10
>>> c.diameter
20
>>> c.diameter = 30
Traceback (most recent call last):
  File "<pyshell#167>", line 1, in <module>
    c.diameter = 30
AttributeError: can't set attribute
```

but you can't set an attribute if no setter has been defined

classes: name mangling


```
class Duck():
    '''getters/setters example using decorators'''
    def __init__(self, input_name):
        self.__name = input_name

    @property
    def name(self):
        'getter for name attribute'
        print('inside the getter')
        return self.__name

    @name.setter
    def name(self, input_name):
        'setter for name attribute'
        print('inside the setter')
        self.__name = input_name
```

data which is intended to be private should be preceded with a double underscore

classes: name mangling


```
>>> d = Duck('Donald')
>>> d.name
inside the getter
'Donald'
>>> d.name = 'Daffy'
inside the setter
>>> d.name
inside the getter
'Daffy'
>>> d.__name
Traceback (most recent call last):
  File "<pyshell#175>", line 1, in <module>
    d.__name
AttributeError: 'Duck' object has no attribute '__name'
>>> d._Duck__name
'Daffy'
```

__name is "mangled" so it can't be accessed outside the class…

except by its mangled name

Lab: Calculator class

- Create a class `Calculator` which acts like a calculator
- Your class should have methods `add()`, `sub()`, `mult()`, `div()`, `pow()`, `log()`, and `log10()`, but you can add more if you wish
- Each of the above methods (except `log`/`log10`) should take 1 or 2 arguments–for 1 argument, e.g., `add(1)`, your method should add to the running total. For 2 arguments, your method should act on those 2 arguments to create the new running total
  - e.g., `add(2, 4)` should produce 6, and then when followed by `multiply(5)`, it should produce 30
- All calculations should be stored, and should be accessible to the caller via the `showcalc()` method.
- You should also have an `ac()` "all clear" method which clears the running total and the list of calculations (i.e., `showcalc()` should produce no output, or "0.0" when preceded by `ac()`).

Lab: Calculator class (solution p. 1)

Develop Intelligence

```
import math

class Calc():
    def __init__(self):
        'Same as ac()--see below'
        self.ac()

    def __str__(self):
        'The string representation of Calc() is the list of calculations.'
        return self.showcalc()

    def __add__(self, other):
        'Calc1 + Calc2 combines the totals and the lists of caclulations.'
        self.calculations.extend(other.calculations)
        self.total = other.total
        return self

    def notecalc(self, op, op1, op2):
        'Add latest calculation to the running list.'
        calc = ''
        if op == 'log':
            op += op2
        else:
            calc = str(op2) + " "
        calc += str(op) + ' ' + str(op1) + ' = ' + str(self.total)
        self.calculations.append(calc)
```

Lab: Calculator class (solution p. 2)

Develop Intelligence

```
def add(self, op1, op2 = None):
    'Add two numbers. If only one number supplied, add to running total.'
    if not op2:
        op2 = self.total
    self.total = op1 + op2
    self.notecalc('+', op1, op2)
    return self.total

def sub(self, op1, op2 = None):
    '''Subtract two numbers. If only one number supplied, subtract from
    running total.'''

    if not op2:
        op2 = op1
        op1 = self.total
    self.total = op1 - op2
    self.notecalc('-', op1, op2)
    return self.total

def mult(self, op1, op2 = None):
    '''Multiply two numbers. If only one number supplied, multiply
    running total.'''
    if not op2:
        op2 = self.total
    self.total = op1 * op2
    self.notecalc('*', op1, op2)

def pow(self, op1, op2 = None):
    '''Raise a number to a power. If no number supplied, raise the
    running total to the power.'''
    if not op2:
        op2 = op1
        op1 = self.total
    self.total = math.pow(op1, op2)
    self.notecalc('**', op1, op2)
    return self.total
```

Lab: Calculator class (solution p. 3)

Develop Intelligence

```
def dolog(self, logtype, op1 = None):
    '''Take log/log10 of a number. If no number supplied, take log
    of the running total.'''
    if not op1:
        op1 = self.total
    if logtype == 'log':
        self.total = math.log(op1)
        self.notecalc('log', op1, '')
    else:
        self.total = math.log10(op1)
        self.notecalc('log', op1, '10')
    return self.total

def log(self, op1 = None):
    return self.dolog('log', op1)

def log10(self, op1 = None):
    return self.dolog('log10', op1)

def showcalc(self):
    'Show current list of calculations.'
    return '\n'.join(self.calculations)

def ac(self):
    'All clear--set total to 0 and erase the list of calculations.'
    self.calculations = []
    self.total = 0.0
```

classes: inheritance


```
class Polygon:
    def __init__(self, num_sides):
        self.num_sides = num_sides
        self.sides = [0 for i in range(num_sides)]

    def __repr__(self):
        return ", ".join([str(i) for i in self.sides])

    def inputSides(self):
        self.sides = [float(input("Enter side "+ str(i + 1) + ": "))
                      for i in range(self.num_sides)]

    def area(self):
        print("Can't compute area of unknown polygon!")
        raise ValueError
```

classes: inheritance

overriding `__init__` method from base class

```
class Triangle(Polygon):
    def __init__(self):
        '''
        use super() to call __init__ in base class and
        be sure we have 3 sides
        '''
        super().__init__(3)

    def area(self):
        import math
        a, b, c = self.sides
        'compute semi-perimeter'
        s = sum(self.sides) / 2
        "compute area using Heron's formula"
        area = math.sqrt((s * (s - a) * (s - b) * (s - c)))
        return area
```

overriding `area` method from base class

classes: inheritance


```
class Square(Polygon):
    def __init__(self):
        super().__init__(4)

    def inputSides(self):
        'only need one side length for a square'
        s = float(input("Enter length of side: "))
        'only need to store one side'
        self.sides = [s]

    def area(self):
        return self.sides[0] ** 2
```

Lab: class vars vs. instance vars

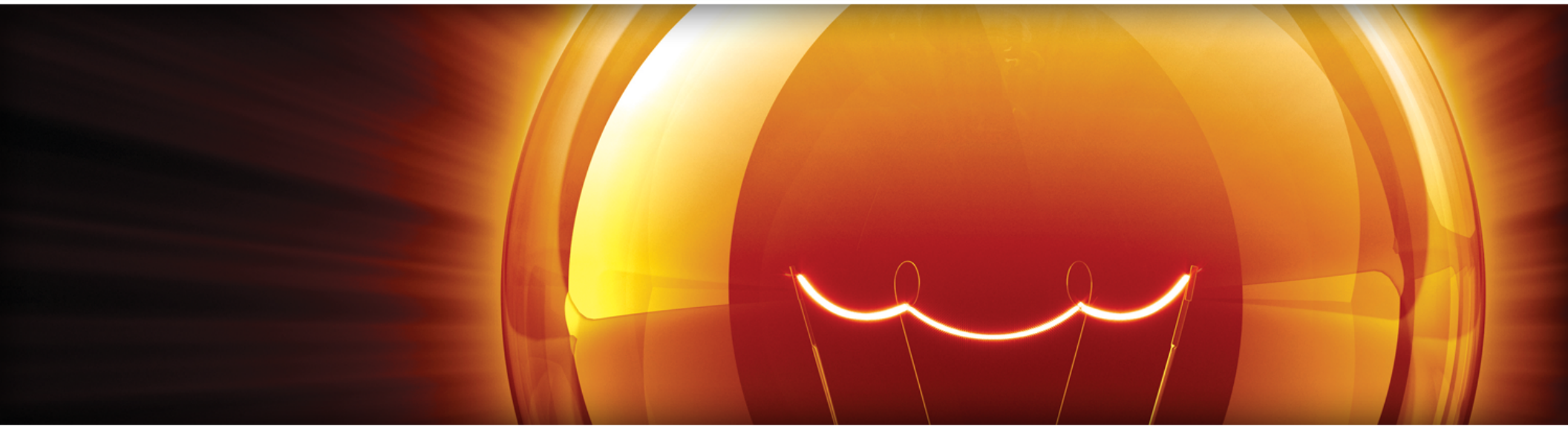
- Variable set outside __init__ belong to the class and are shared by all instances of the class. Can be accessed via ClassName.var and classInstance.var.
- Variables created inside __init__ (and all other method functions) and prefaced with `self.` belong to the object instance and cannot be accessed 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

Lab: class vars vs. instance vars (solution)


```
class Test():
    class_var = 'This var is for every instance'

    def __init__(self, val):
        self.instance_var = val

    def __repr__(self):
        return 'instance var = ' + \
               self.instance_var + \
               '\nclass var = '  + \
               self.class_var
```

```
>>> a = Test('a')
>>> b = Test('b')
>>> a
instance var = a
class var = This var is for every instance
>>> b
instance var = b
class var = This var is for every instance
>>> a.instance_var = 'no longer a'
>>> a
instance var = no longer a
class var = This var is for every instance
>>> b
instance var = b
class var = This var is for every instance
>>> Test.class_var = 'Presto change-o!'
>>> a
instance var = no longer a
class var = Presto change-o!
>>> b
instance var = b
class var = Presto change-o!
```

Unit Testing



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)

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

unittest module

Develop Intelligence

```
import unittest

class TestStringMethods(unittest.TestCase):

  def test_upper(self):
      self.assertEqual('foo'.upper(), 'FOO')

  def test_isupper(self):
      self.assertTrue('FOO'.isupper())
      self.assertFalse('Foo'.isupper())

  def test_split(self):
      s = 'hello world'
      self.assertEqual(s.split(), ['hello', 'world'])
      # check that s.split fails when the separator is not a string
      with self.assertRaises(TypeError):
          s.split(2)

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

subclass unittest.TestCase

method names begin with test_

prevents code from being run if module is imported

unittest module (cont'd)


```
def rounder25(amount):
    '''
    Return amount rounded UP to nearest
    quarter dollar.
        ...$1.89 becomes $2.00
        ...but $1.00/$1.25/$1.75/etc.
           remain unchanged
    '''
    dollars = int(amount)
    cents = round((amount - dollars) * 100)
    quarters = cents // 25
    if cents % 25:
        quarters += 1
    amount = dollars + 0.25 * quarters

    return amount
```

```
import unittest

class TestRounder25(unittest.TestCase):

  def setUp(self):
      print("\nSetting up a test.")

  def tearDown(self):
      print("Tearing down a test.")

  def test_roundup(self):
      self.assertEqual(rounder25(1.03), 1.25)
      self.assertEqual(rounder25(1.89), 2.00)

  def test_noround(self):
      self.assertEqual(rounder25(1.00), 1.00)
      self.assertEqual(rounder25(1.25), 1.25)
      self.assertEqual(rounder25(1.50), 1.50)
      self.assertEqual(rounder25(1.75), 1.75)

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