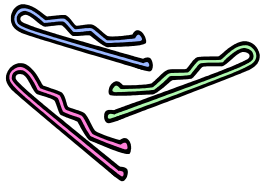
# CS 1
# Introduction to Computer Programming

*Lecture 21*: November 27, 2012

## Odds and Ends, part 2








# Last time

- Class inheritance
- Raising exceptions using `raise`
- Creating your own exception classes

# Today

- A variety of small topics that don't fit neatly into any category:
  - Method chaining
  - Writing to files
  - `print >> sys.stderr`
  - The `type()` function
  - The `None` type and value
  - Default arguments to functions
  - Functions with arbitrary numbers of arguments
  - Defining functions inside functions

# Today

- (continued)
  - `continue`
  - List slice assignment
  - List stride notation
  - The `//` operator
  - Short-circuiting behavior of `and` and `or`

# Method chaining

- If you have a method call on an object which returns another object, you can "chain" another method call onto the result

```
>>> 'foo'.upper()
'FOO'
>>> '      foo      '.strip()
'foo'
>>> '      foo      '.strip().upper()
'FOO'
```

"Chained" method calls

# Writing to files

- We've seen how to read from files using the **`readline()`** method on files *e.g.*

```
myfile = open('temps.txt', 'r')
while True:
    line = myfile.readline()
    if not line:
        break
    # do something with line...
myfile.close()
```

# Writing to files

- We've also seen how to read from files using the `for` statement, *e.g.*

```
myfile = open('temps.txt', 'r')
for line in myfile:
    # do something with line
myfile.close()
```

# Writing to files

- Writing to files can also be done using a method: the **`write()`** method on file objects
- Before doing this, the file to be written to has to be opened for *writing* or *appending*

```
# writing
myfile = open('results.txt', 'w')
# appending
myfile = open('results.txt', 'a')
```

# Writing vs. appending

- Opening a file using the `'w'` argument to the `open()` function creates a new file
  - removing any previous file with that name!
- Opening a file using the `'a'` argument to the `open()` function creates a new file only if no such file already exists
  - If it already exists, writing to the file just appends new lines to the end of the existing file

# The `write()` method

- Writing to a file is done using the **`write()`** method on file objects

```
myfile = open('results.txt', 'w')
myfile.write('this is line 1\n')
myfile.write('this is line 2\n')
myfile.write('this is line 3\n')
myfile.close()
```

# The `write()` method

- This will result in a new file called **`results.txt`** being created in the current directory with the lines:

```
this is line 1
this is line 2
this is line 3
```

# The `write()` method

- Note that you must put newlines in explicitly when you use the `write()` method:

```
myfile = open('results.txt', 'w')
myfile.write('this is line 1\n')
myfile.write('this is line 2\n')
myfile.write('this is line 3\n')
myfile.close()
```

# The `write()` method

- If you forget to do this *e.g.*

```
myfile = open('results.txt', 'w')
myfile.write('this is line 1')
myfile.write('this is line 2')
myfile.write('this is line 3')
myfile.close()
```

# The `write()` method

- This will result in:

```
this is line 1this is line 2this is line 3
```

- (common beginner's mistake)

# print >> file

- There is another way to write to files using the `print` statement:

```
myfile = open('results.txt', 'w')
print >> myfile, 'this is line 1'
print >> myfile, 'this is line 2'
print >> myfile, 'this is line 2'
myfile.close()
```

# print >> file

- There is another way to write to files using the `print` statement:

```
print >> myfile, 'this is line 1'
```

- This will print exactly what a `print` statement would print, but to the file `myfile` instead of to the terminal
- The newline (`'\n'`) character is added to the end of the line automatically

# `print >> sys.stderr`

- There is a special file object in the **`sys`** module called **`stderr`**
- It's not a "real" file
  - printing to **`sys.stderr`** prints to the terminal
- **`stderr`** stands for "standard error"

# print >> sys.stderr

- `stderr` is a place where error messages are supposed to go
- When you want to print out an error message, you should always write

```
print >> sys.stderr, 'You messed up!'
```

- instead of just

```
print 'You messed up!'
```

# `print >> sys.stderr`

- Keeping regular **`print`** statements distinct from **`print`** statements which print error messages is useful
- It's possible to redirect where **`sys.stderr`** prints to when the program executes
  - for instance, can redirect all the error messages to a log file
- So if you have to print error messages, you should print them to **`sys.stderr`**

# The `type()` function

- Python allows you to check the type of any value using the `type()` function:

```
>>> type(1010)
<type 'int'>
>>> type('foobar')
<type 'str'>
>>> type(False)
<type 'bool'>
>>> type(3.14)
<type 'float'>
```

# The `type()` function

- The `type()` function returns a type description object, not a string:

```
>>> type(type(1010))
<type 'type'>
```

- The `type` object's string representation is

```
<type '<name of type>'>
```

- So we have `<type 'int'>`, `<type 'float'>`, etc.

# The `type()` function

- Python doesn't distinguish some things that you might consider to be distinct types:

```
>>> type([1, 2, 3])
<type 'list'>
>>> type([1.23, 4.56, 7.89])
<type 'list'>
>>> type([12, 45.1, 'foobar', False])
<type 'list'>
```

- All of these are just "lists" to Python

# The `type()` function

- The `type()` function is mainly useful for checking that the arguments to a function or method are correct
- If they aren't right, normally raise a `TypeError` exception with an error message
- Type errors are very common, so this is an easy way to make your functions more robust

# The `type()` function

- Example: factorials again!

```
def factorial(n):
    '''Returns the factorial of n.'''
    if type(n) != type(0):  # not an int
        raise TypeError('n must be an int')
    if n < 0:
        raise ValueError('n must be >= 0')
    # rest of code
```

# The `type()` function

- Writing `if type(n) != type(0):` is very ugly, so Python provides some shortcuts
- The `types` module contains values which represent most standard types
  - `types.IntType` → `int`
  - `types.FloatType` → `float`
  - `types.StringType` → `string`
  - `types.ListType` → `list`
- So we can write:

```
if type(n) != types.IntType: ...
```

# The `type()` function

- Writing `if type(n) != types.IntType:` is *still* very ugly, so Python provides some *more* shortcuts
- Common type names are built-in to Python

```
>>> int
<type 'int'>
>>> list
<type 'list'>
```

etc.

# The `type()` function

- Now we can write

```
if type(n) != int: ...
```

- There's an even nicer version:

```
if type(n) is not int: ...
```

- `is` is an operator in Python
  - means "is exactly the same thing as"
  - usually the same as `==`
- `is not` is an operator which is the opposite of `is`

# The `type()` function

- The "nice" way to write the example:

```
def factorial(n):
    '''Returns the factorial of n.'''
    if type(n) is not int:
        raise TypeError('n must be an int')
    if n < 0:
        raise ValueError('n must be >= 0')
    # rest of code
```

- This is the recommended way to check the type of arguments

# The `None` type and value

- There is a special type and value called **`None`**
- It represents a value of no significance

```
>>> None
```

[no output]

```
>>> type(None)
<type 'NoneType'>
```

- What's the point of this?

# The `None` type and value

- The `None` value is what is returned from a function which doesn't actually return a value:

```
def foo():
    '''Do nothing.'''
    return

>>> type(foo())
<type 'NoneType'>
```

# The `None` type and value

- The `None` value is what is returned from a function which doesn't actually return a value
- This makes Python more consistent:
  - *Every* function returns a value
  - Functions that don't explicitly return a value return `None`

# The `None` type and value

- The `None` value can also be used when you want to initialize something, but don't care what the value is
- Example: a list of 5 items which will be filled in later

```
>>> lst = [None, None, None, None, None]
```

- (or:)

```
>>> lst = [None] * 5
```

# Interlude

- TV clip: All about candy! ☺

# Default arguments to functions

- It's often the case that we want to define functions or methods with arguments that can have default values
- If a particular argument isn't supplied, use the default value instead
- Often, this default value will be `None`

# Default arguments to functions

- Example: defining an exception class
- Want the constructor to be able to take an error message as an argument:

```
class MyException:
    def __init__(self, msg):
        self.msg = msg
```

# Default arguments to functions

```
class MyException:
    def __init__(self, msg):
        self.msg = msg
```

- We can use it as follows:

```
>>> raise MyException("You're hosed!")
```

- However, if we try:

```
>>> raise MyException
```

- We get:

```
TypeError: __init__() takes exactly 2
  arguments (1 given)
```

# Default arguments to functions

- We can fix this by giving the constructor a *default argument* for **`msg`**:

```
class MyException:
    def __init__(self, msg=None):
        self.msg = msg
```

- Now, if we do this:

```
>>> raise MyException
```

- it works (**`msg`** is **`None`**)

# Default arguments to functions

- Note that the recommended way to define our own exception classes is not this:

```
class MyException:
    def __init__(self, msg=None):
        self.msg = msg
```

- but instead:

```
class MyException(Exception):
    pass
```

- which gives the desired behavior (and other useful things as well)

# Default arguments to functions

- If there are default arguments, they must come after normal arguments

```
def foo(x, y, z=10, w=20):  # OK
    # ...

def foo(x=10, y, z, w=20):  # not allowed
    # ...
```

- This prevents some ambiguities which could occur if this rule wasn't in place

# Default args vs keyword args

- Notice that default arguments to functions are specified when the function itself is *defined*:

```
def foo(x, y, opt=None):
    # whatever
```

- Don't confuse with keyword arguments, which are specified when a function is *called*:

```
def bar(x, y, **kw):
    # whatever
>>> bar(10, 20, arg=3.14)
```

# Functions with arbitrary numbers of arguments

- Python lets you define functions with arbitrary numbers of arguments

```
def func(x, y, *z):
    print x, y
    print z
>>> func(1, 2, 3, 4, 5)
1 2
(3, 4, 5)
```

# Functions with arbitrary numbers of arguments

- Python lets you define functions with arbitrary numbers of arguments

```
def func(x, y, *z):
    print x, y
    print z
```

- All arguments except **x** and **y** get put into a tuple called **z**
- Similar to keyword arguments, which use ****** instead of *

# Functions with arbitrary numbers of arguments

- This can be used to define functions like **`range()`** that can be called with many different numbers of arguments

```
def myRange(*nums):
    if len(nums) == 1:
        return range(nums[0])
    elif len(nums) == 2:
        return range(nums[0], nums[1])
    elif len(nums) == 3:
        return range(nums[0], nums[1], nums[2])
    else:
        raise ValueError('Wrong number of arguments')
```

# Defining functions inside functions

- Python allows you to define functions inside of other functions!
- The inner function can only be called inside of the function in which it's defined
- Sometimes useful to define small "helper functions" inside a larger function to evaluate expressions that are repeated several times
  - could always "lift" inner function to a top-level function, but sometimes it's clearer if you don't

# Defining functions inside functions

- Example:

```
def evaluate_list(lst, a, b, c):
    def evaluate_num(a, b, c, n):
        return (a * n ** 2 + b * n + c)
    for i, n in enumerate(lst):
        lst[i] = evaluate_num(a, b, c, n)
```

- Note: can't call `evaluate_num()` outside of this function:

```
>>> evaluate_num(10)
NameError: name 'evaluate_num' is not defined
```

# `continue`

- We've seen that we can use **`break`** to exit a loop immediately
- Sometimes we want to immediately go to the *next iteration* of the loop, but not exit the loop entirely
- There is a keyword to do this: **`continue`**

# continue

```
for i in range(100):
    if i % 2 == 0 or i % 3 == 0 or i % 5 == 0:
        # ignore these numbers, go to next number
        continue
    # print all other numbers
    print i
```

```
1
7
11
13
17
...
```

# `continue`

- **`continue`** doesn't enable you to do anything you couldn't do without it
  - but sometimes it's more convenient to use **`continue`** than to have an extra **`if`**/**`else`** statement

# List slice assignment

- We can take slices from lists:

```
>>> lst = [2, 4, 6, 8, 10]
>>> lst[2:4]
[6, 8]
```

- We can also *assign* to list slices:

```
>>> lst[2:4] = [1001, 42]
>>> lst
[2, 4, 1001, 42, 10]
```

# List slice assignment

- When assigning to list slices:
  - compute the part of the list represented by the slice
  - replace it by the items on the right of the `=` operator

```
>>> lst[2:4] = [1001, 42]
```

- This replaces `lst[2]` with `1001`, and `lst[3]` with `42`
- The rest of the list is left unchanged

# List slice assignment

- You use this trick to expand or shrink a list from within:

```
>>> lst = [1, 3, 5, 7, 9]
>>> lst[2:4] = [-1, -2, -3, -4]
>>> lst
[1, 3, -1, -2, -3, -4, 9]
>>> lst[2:6] = [22]
>>> lst
[1, 3, 22, 9]
```

# List stride notation

- We can also use slices to pick every N$^{th}$ element from a list:

```
>>> lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> lst[0:10:2]
[1, 3, 5, 7, 9]
>>> lst[:10:2]
[1, 3, 5, 7, 9]
>>> lst[::2]
[1, 3, 5, 7, 9]
```

# List stride notation

- A list "stride" is a list slice with an extra slice value

```
lst[0:10:2]
```

- means: `lst`, starting from index `0`, up to but not including index `10`, indices going up by `2`

```
lst[3:22:7]
```

- means: `lst`, starting from index `3`, up to but not including index `22`, indices going up by `7`

# List stride notation

- A cute trick with list strides:

```
lst[::-1]
```

- This returns the *reverse* of the list `lst`

# The // operator

- Division in Python uses the `/` operator
- This operator discards the remainder if operands are `int`s
- There is an alternative division operator called `//` which discards the remainder even if the operands are `float`s:

```
>>> 7.5 / 3.0
2.5
>>> 7.5 // 3.0
2.0
```

# Short-circuiting behavior of `and` and `or`

- We can combine boolean-valued expressions (expressions which evaluate to `True` or `False`) using the `and` and `or` operators
- Both `and` and `or` have a "short-circuiting" property:
  - if first operand of `and` is `False`, the entire expression is `False`
  - if first operand of `or` is `True`, the entire expression is `True`
  - Don't have to evaluate second operand in these cases

# Short-circuiting behavior of `and` and `or`

- Second operand cannot have any effect, so the result is the same:
  - `False and <anything>` → `False`
  - `True or <anything>` → `True`
- Not evaluating second operand can make code more efficient
  - especially if second operand takes a long time to evaluate

# Short-circuiting behavior of `and` and `or`

- For instance:
  - `<expression evaluating to False> and <expression which takes a long time to evaluate>` → `False`
  - `<expression evaluating to True> or <expression which takes a long time to evaluate>` → `True`
- This is standard behavior for `and` and `or` in most programming languages

# Next time

- Friday lecture!
- Final `tkinter` lecture: GUI "widgets"
- Learn to program using buttons, drop-down lists, text entry boxes, etc.