Before starting OOP…

- Today:
  - Review Useful tips and concepts

(based on CS 11 Python track, CALTECH)

Useful coding idioms

- "Idiom"
  - Standard ways of accomplishing a common task
- Using standard idioms won't make your code more correct, but
  - more concise
  - more readable
  - better designed (sometimes)

Trivial stuff (1)

- The **`None`** type and value:
- Sometimes, need a way to express the notion of a value which has no significance
  - often a placeholder for something which will be added later, or for an optional argument
- Use **`None`** for this
  - **`None`** is both a value and a type

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

Trivial stuff (2)

- Can use the `return` keyword with no argument:

```
def foo(x):
    print x
    return  # no argument!
```

- Here, not needed; function will return automatically once it gets to the end
  - needed if you have to return in the middle of a function
- Can use `return` with no argument if you want to exit the function before the end
- `return` with no argument returns a `None` value

Trivial stuff (3)

- Can write more than one statement on a line, separated by semicolons:

```
>>> a = 1; b = 2
>>> a
1
>>> b
2
```

- Not recommended; makes code harder to read

Trivial stuff (4)

- Can write one-line conditionals:

```
if i > 0: break
```

  - Sometimes convenient
- Or one-line loops:

```
while True: print "hello!"
```

  - Not sure why you'd want to do this

Trivial stuff (5)

- Remember the short-cut operators:
  - `+= -= *= /=` etc.
- Use them where possible
  - more concise, readable
- Don't write

```
i = i + 1
```

- Instead, write

```
i += 1
```

Trivial stuff (6)

- Unary minus operator
- Sometimes have a variable a, want to get its negation
- Use the unary minus operator:

```
a = 10
b = -a
```

- Seems simple, but I often see
  - `b = 0 - a`
  - `b = a * (-1)`

Trivial stuff (6)

- The **`%g`** formatting operator
- Can use **`%f`** for formatting floating point numbers when printing
- Problem: **`%f`** prints lots of trailing zeros:

```
>>> print "%f" % 3.14
3.140000
```

- **`%g`** is like **`%f`**, but suppresses trailing zeros:

```
>>> print "%g" % 3.14
3.14
```

`print` (1)

- Recall that print always puts a newline after it prints something
- To suppress this, add a trailing comma:

```
>>> print "hello"; print "goodbye"
hello
goodbye
>>> print "hello", ; print "goodbye"
hello goodbye
>>>
```

- N.B. with the comma, `print` still separates with a space

`print` (2)

- To print something without a trailing newline or a space, need to use the **`write()`** method of file objects:

```
>>> import sys
>>> sys.stdout.write("hello"); sys.stdout.write("goodbye")
hellogoodbye>>>
```

`print` (3)

- To print a blank line, use `print` with no arguments:

```
>>> print
```

- Don't do this:

```
>>> print ""
```

- (It's just a waste of effort)

`print` (4)

- Can print multiple items with `print`:

```
>>> a = 10; b = "foobar"; c = [1, 2, 3]
>>> print a, b, c
10 foobar [1, 2, 3]
```

- `print` puts a space between each pair of items
- Usually better to use a format string
  - get more control over the appearance of the output

The `range()` function (1)

- The `range()` function can be called in many different ways:

```
range(5)            # [0, 1, 2, 3, 4]
range(3, 7)         # [3, 4, 5, 6]
range(3, 9, 2)      # [3, 5, 7]
range(5, 0, -1)     # [5, 4, 3, 2, 1]
```

The `range()` function (2)

- `range()` has at most three arguments:
  - starting point of range
  - end point (really, 1 past end point of range)
  - step size (can be negative)
- `range()` with one argument
  - starting point == 0
  - step size == 1
- `range()`with two arguments
  - step size == 1

Type checking (1)

- Often want to check whether an argument to a function is the correct type
- Several ways to do this (good and bad)
- Always use the **`type()`** built-in function

```
>>> type(10)
<type 'int'>
>>> type("foo")
<type 'str'>
```

Type checking (2)

- To check if a variable is an integer:
- Bad:

```
if type(x) == type(10): ...
```

- Better:

```
import types
if type(x) == types.IntType: ...
```

- Best:

```
if type(x) is int: ...
```

Type checking (3)

- Many types listed in the **`types`** module
- **`IntType`**, **`FloatType`**, **`ListType`**, ...
- Try this:

```
import types
dir(types)
```

  - (to get a full list)

```
>>> types.IntType
<type 'int'>
```

Type checking (4)

- Some type names are now built in to python:

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

- So we don't usually need to `import types` any more

Type checking (5)

- You could write

```
if type(x) == int: ...
```

- but this is preferred:

```
if type(x) is int: ...
```

- It looks better
- `is` is a rarely-used python operator
  - equivalent to `==` for types

Type conversions (1)

- Lots of built-in functions to do type conversions in python:

```
>>> float("42")
42.0
>>> float(42)
42.0
>>> int(42.5)
42
>>> int("42")
42
```

Type conversions (2)

- Converting to strings:

```
>>> str(1001)
'1001'
>>> str(3.14)
'3.14'
>>> str([1, 2, 3])
'[1, 2, 3]'
```

Type conversions (3)

- Different way to convert to strings:

```
>>> `1001`    # "back-tick" operator
'1001'
>>> a = 3.14
>>> `a`
'3.14'
>>> `[1, 2, 3]`
'[1, 2, 3]'
```

- Means the same thing as the `str` function

Type conversions (4)

- Converting to lists:

```
>>> list("foobar")
['f', 'o', 'o', 'b', 'a', 'r']
>>> list((1, 2, 3))
[1, 2, 3]
```

- Converting from list to tuple:

```
>>> tuple([1, 2, 3])
(1, 2, 3)
```

The "`in`" operator (1)

- The `in` operator is used in two ways:
  - 1) Iterating over some kind of sequence
  - 2) Testing for membership in a sequence
- Iteration form:

```
for item in sequence: ...
```

- Membership testing form:

```
item in sequence
```

(returns a boolean value)

The "`in`" operator (2)

- Iterating over some kind of sequence

```
for line in some_file: ...
    # line is bound to each
    # successive line in the file "some_file"

for item in [1, 2, 3, 4, 5]: ...
   # item is bound to numbers 1 to 5

for char in "foobar": ...
   # char is bound to 'f', then 'o', ...
```

The "`in`" operator (3)

- Testing for membership in a sequence

```
# Test that x is either -1, 0, or 1:
lst = [-1, 0, 1]
x = 0
if x in lst:
    print "x is a valid value!"
```

- Can test for membership in strings, tuples:

```
if c in "foobar": ...
if x in (-1, 0, 1): ...
```

The "`in`" operator (4)

- Testing for membership in a dictionary:

```
>>> d = { "foo" : 1, "bar" : 2 }
>>> "foo" in d
True
>>> 1 in d
False
```

- Iterating through a dictionary:

```
>>> for key in d: print key
foo
bar
```

More stuff about lists (1)

- Use `lst[-1]` to get the last element of a list `lst`
- Similarly, can use `lst[-2]` to get second-last element
  - though it won't wrap around if you go past the first element
- The `pop()` method on lists:
  - `lst.pop()` will remove the last element of list `lst` and return it
  - `lst.pop(0)` will remove the first element of list `lst` and return it
  - and so on for other values

More stuff about lists (2)

- To copy a list, use an empty slice:

```
copy_of_lst = lst[:]
```

- This is a *shallow copy*
  - If `lst` is a list of lists, the inner lists will not be copied
  - Will just get a copy of the reference to the inner list
  - *Very* common source of bugs!
- If you need a *deep copy* (full copy all the way down), can use the `copy.deepcopy` method (in the `copy` module)

More stuff about lists (3)

```
>>> lst = [[1, 2], [3, 4]]
>>> copy_of_lst = lst[:]
>>> lst[0][0] = 10
>>> lst
[[10, 2], [3, 4]]
>>> copy_of_lst
[[10, 2], [3, 4]]
```

- This is probably not what you expected

More stuff about lists (4)

- Often want to make a list containing many copies of the same thing
- A shorthand syntax exists for this:

```
>>> [0] * 10     # or 10 * [0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- Be careful! This is still a shallow copy!

```
>>> [[1, 2, 3]] * 2
[[1, 2, 3], [1, 2, 3]]
```

- Both elements are the *same* list!

More stuff about lists (5)

- The `sum()` function
- If a list is just numbers, can sum the list using the `sum()` function:

```
>>> lst = range(10)
>>> lst
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sum(lst)
45
```

More stuff about strings (1)

- If you need a string containing the letters from a to z, use the **`string`** module

```
>>> import string
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

- If you need the count of a particular character in a string, use **`string.count`** or the **`count`** method:

```
string.count("foobar", "o")  # 2
"foobar".count("o")  # also 2
```

More stuff about strings (2)

- Comparison operators work on strings
- Uses "lexicographic" (dictionary) order

```
>>> "foobar" < "foo"
False
>>> "foobar" < "goo"
True
```

More stuff about strings (3)

- Can "multiply" a string by a number:

```
>>> "foo" * 3
'foofoofoo'
>>> 4 * "bar"
'barbarbarbar'
>>> 'a' * 20
'aaaaaaaaaaaaaaaaaaaa'
```

- This is occasionally useful

More stuff about tuples (1)

- Tuples can be used to do an in-place swap of two variables:

```
>>> a = 10; b = 42
>>> (a, b) = (b, a)
>>> a
42
>>> b
10
```

More stuff about tuples (2)

- This can also be written without parentheses:

```
>>> a = 10; b = 42
>>> a, b = b, a
>>> a
42
>>> b
10
```

More stuff about tuples (3)

- Why this works:
  - In python, the right-hand side of the **`=`** (assignment) operator is always evaluated before the left-hand side
  - the **`(b, a)`** on the right hand side packs the current versions of **b** and **a** into a tuple
  - the **`(a, b) =`** on the left-hand side unpacks the two values so that the new **a** is the old **b** etc.
- This is called "tuple packing and unpacking"

Review (cont.)

- List slice notation
- Multiline strings
- Docstrings

List slices (1)

```
a = [1, 2, 3, 4, 5]
print a[0]  # 1
print a[4]  # 5
print a[5]  # error!
a[0] = 42
```

List slices (2)

```
a = [1, 2, 3, 4, 5]
a[1:3]  # [2, 3] (new list)
a[:]    # copy of a
a[-1]   # last element of a
a[:-1]  # all but last
a[1:]   # all but first
```

List slices (3)

```
a = [1, 2, 3, 4, 5]
a[1:3]  # [2, 3] (new list)
a[1:3] = [20, 30]
print a
[1, 20, 30, 4, 5]
```

Multiline strings

```
s = "this is a string"
s2 = 'this is too'
s3 = "so 'is' this"
sl = """this is a
multiline string."""
sl2 = '''this is also a
   multiline string'''
```

Docstrings (1)

- Multiline strings most useful for documentation strings aka "docstrings":

```
def foo(x):
    """Comment stating the purpose of
    the function 'foo'. """
    # code...
```

- Can retrieve as `foo.__doc__`

Docstrings (2)

- Use docstrings:
  - in functions/methods, to explain
    - what function does
    - what arguments mean
    - what return value represents
  - in classes, to describe purpose of class
  - at beginning of module
- Don’t use comments where docstrings are preferred

Exception handling

- What do we do when something goes wrong in code?
  - exit program (too drastic)
  - return an integer error code (clutters code)
- Exception handling is a cleaner way to deal with this
- Errors "raise" an exception
- Other code can "catch" an exception and deal with it

try/raise/except (1)

```
try:
    a = 1 / 0
    # this raises ZeroDivisionError
except ZeroDivisionError:
    # catch and handle the exception
    print "divide by zero"
    a = -1   # lame!
```

try/raise/except (2)

```
try:
    a = 1 / 0
    # this raises ZeroDivisionError
except:    # no exception specified
    # catches ANY exception
    print "something bad happened"
    # Don’t do this!
```

try/raise/except (3)

```
try:
     a = 1 / 0
     # this raises ZeroDivisionError
except:    # no exception specified
     # Reraise original exception:
    raise
     # This is even worse!
```

Backtraces

- Uncaught exceptions give rise to a stack backtrace:

```
# python bogus.py
Traceback (most recent call last):
  file "bogus.py", line 5, in ?
    foo()
  file "bogus.py", line 2, in foo
    a = 1 / 0
  ZeroDivisionError: integer division or modulo by
  zero
```

- Backtrace is better than catch-all exception handler

Exceptions are classes

```
class SomeException:
    def __init__(self, value=None):
        self.value = value
    def __repr__(self):
        return `self.value`
```

- The expression `` `self.value` `` is the same as `str(value)`
- *i.e.* converts object to string

Raising exceptions (1)

```
def some_function():
    if something_bad_happens():
        # SomeException leaves function
        raise SomeException("bad!")
    else:
        # do the normal thing
```

Raising exceptions (2)

```
def some_other_function():
    try:
        some_function()
    except SomeException, e:
        # e gets the exception that was caught
        print e.value
```

Raising exceptions (3)

```
# This is silly:
try:
    raise SomeException("bad!")
except SomeException, e:
    print e  # prints "bad!"
```

`try/finally` (1)

- We put code that can throw exceptions into a `try` block
- We catch exceptions inside `except` blocks
- We don't have to catch all exceptions
  - If we don't catch an exception, it will leave the function and go to the function that called that function, until it finds an `except` block or reaches the top level
- Sometimes, we need to do something regardless of whether or not an exception gets thrown
  - e.g. closing a file that was opened in a `try` block

try/finally (2)

```
try:
    # code goes here...
    if something_bad_happens():
        raise MyException("bad")
finally:
    # executes if MyException was not raised
    # executes and re-raises exception
    #   if MyException was raised
```

- Can have `finally` or `except` statements, not both (which is a bogus rule, but there you are)
  - This will change in future versions of python

try/finally (3)

- try/finally

```
try:
    myfile = file("foo") # open file "foo"
    if something_bad_happens():
        raise MyException("bad")
finally:
    # Close the file whether or not an
    # exception was thrown.
    myfile.close()
    # If an exception was thrown, reraise
    # it here.
```

Exception classes

- Exception classes, with arguments:

```
class MyException(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return ‘self.value‘
try:
    raise MyException(42)
except MyException, e:
    print "bad! value: %d" % e.value
```

More odds and ends

- assertions
- "`print >>`" syntax
- more on argument lists
- functional programming tools
- list comprehensions

Odds and ends (1)

- Assertions

```
# 'i' should be zero here:
assert i == 0
# If fail, exception raised.
```

- "print to" syntax

```
import sys
print >> sys.stderr, "bad!"
```

Note on error messages

- Error messages should always go to `sys.stderr`
- Two ways to do this:

```
import sys
print >> sys.stderr, "bad!"

sys.stderr.write("bad!\n")
```

- Either is fine
- Note that `write()` doesn't add newline at end

Odds and ends (2) – arg lists

- Default arguments, keyword arguments

```
def foo(val=10):
    print val
foo()          # prints 10
foo(20)        # prints 20
foo(val=30)    # prints 30
```

- Default args must be at end of argument list

Odds and ends (3) – arg lists

- Arbitrary number of arguments

```
def foo(x, y, *rest):
    print x, y
    # print tuple of the rest args:
    print rest
>>> foo(1, 2, 3, 4, 5)
1 2
(3, 4, 5)
```

Odds and ends (4) – arg lists

- Arbitrary number of regular/keyword args:

```
def foo(x, y, *rest, **kw):
    print x, y
    print rest
    print kw
>>> foo(1, 2, 3, 4, 5, bar=6, baz=7)
1 2
(3, 4, 5)
{ baz : 7, bar : 6 }
```

Functional programming tools (1)

- First-class functions:

```
def foo(x):
    return x * 2
>>> bar = foo
>>> bar(3)
6
```

Functional programming tools (2)

- **lambda**, **map**, **reduce**, **filter**:

```
>>> map(lambda x: x * 2, [1, 2, 3, 4, 5])
[2, 4, 6, 8, 10]
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
15
>>> sum([1, 2, 3, 4, 5])  # easier
15
>>> filter(lambda x: x % 2 == 1, range(10))
[1, 3, 5, 7, 9]
```

List comprehensions

```
>>> vec = [2, 4, 6]
>>> [3 * x for x in vec]
[6, 12, 18]
>>> [3 * x for x in vec if x > 3]
[12, 18]
>>> [3 * x for x in vec if x < 2]
[]
>>> [[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```