



CS 11 python track: lecture 2

- Today:
 - Odds and ends
 - Introduction to object-oriented programming
 - Exception handling



Odds and ends

- 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"
```

```
s1 = """this is a  
multiline string."""
```

```
s12 = '''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



Introduction to OOP

- OOP = **O**bject-**O**riented **P**rogramming
- OOP is very simple in python
 - but also powerful
- What is an object?
 - data structure, and
 - functions (methods) that operate on it



OOP terminology

- **class** -- a template for building objects
- **instance** -- an object created from the template (an instance of the class)
- **method** -- a function that is part of the object and acts on instances directly
- **constructor** -- special "method" that creates new instances of a particular class



Defining a class

```
class Thingy:
```

```
    """This class stores an arbitrary object."""
```

```
    def __init__(self, value):
```

```
        """Initialize a Thingy."""
```

```
        self.value = value
```

constructor

```
    def showme(self):
```

```
        """Print this object to stdout."""
```

```
        print "value = %s" % self.value
```

method



Using a class (1)

```
t = Thingy(10)    # calls __init__ method
t.showme()        # prints "value = 10"
```

- `t` is an **instance** of class `Thingy`
- `showme` is a **method** of class `Thingy`
- `__init__` is the **constructor method** of class `Thingy`
 - when a `Thingy` is created, the `__init__` method is called
- Methods starting and ending with `__` are "special" methods



Using a class (2)

```
print t.value # prints "10"
```

- `value` is a **field** of class **Thingy**

```
t.value = 20 # change the field value
```

```
print t.value # prints "20"
```



More fun stuff

- Can write `showme` a different way:

```
def repr(self):  
    return str(self.value)
```

- Now can do:

```
print t # prints "10"
```

```
print "thingy: %s" % t # prints "thingy: 10"
```

- repr converts object to string



"Special" methods

- All start and end with `__` (two underscores)
- Most are used to emulate functionality of built-in types in user-defined classes
- e.g. operator overloading
 - `__add__`, `__sub__`, `__mult__`, ...
 - see python docs for more information



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"
```



try/raise/except (2)

```
try:
```

```
    a = 1 / 0
```

```
    # this raises ZeroDivisionError
```

```
except:    # no exception specified
```

```
    # catches ANY exception
```

```
    print "something bad happened"
```

```
    # Not usually a good idea;
```

```
    # better to catch specific exceptions
```



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:  
        # do whatever is appropriate
```



Raising exceptions (3)

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



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



Summing up

- Use classes where possible
- Use exceptions to deal with error situations
- Use docstrings for documentation
- In two weeks: more OOP (inheritance)