

# Class extension and Exception handling

Genome 559

# Review - classes

## 1) Class constructors -

```
class myClass:
    def __init__(self, arg1, arg2):
        self.var1 = arg1
        self.var2 = arg2
foo = myClass('student', 'teacher')
```

## 2) `__str__()` method and how Python print works

## 3) Other core Python class functions -

```
__add__()      # used for the '+' operator
__mul__()     # used for the '*' operator
__sub__()     # used for the '-' operator
etc.
```

# Extending Classes

Suppose you want to build on a class that Python (or someone else) has already written for you?

- get their code and copy it into your own file for modification
- OR use the "extension" mechanism provided for you by Python

# Extension formalism - much like biological classification

```
class Eukaryote
```

```
    class Animal (extends Eukaryote)  
        add class method movementRate ()
```

```
    class Insecta (extends Animal)  
        add class method numberOfWings ()
```

```
    class Drosophila (extends Insecta)  
        add class method preferredFruit ()
```

What methods are available for an object of type **Drosophila**?

**Drosophila** is an **Insecta** so it has all the **Insecta** data structures and methods.

**Drosophila** is also an **Animal** so it has all the **Animal** data structures and methods (and **Eukaryote**, though we didn't define any).

# Writing a new Class by extension

Writing a class (review):

```
class Date:  
    def __init__(self, day, month, year):  
        <assign arguments to class variables>
```

Extending an existing class:

```
class HotDate(Date):  
    def __init__(self, day, month, year, toothbrush):  
        super(day, month, year)  
        self.bringToothbrush = toothbrush
```

class to extend

super - call the constructor for Date

ALL OF THE DATA TYPES AND METHODS WRITTEN FOR **Date** ARE NOW AVAILABLE!

# Class hierarchy

class **Eukaryote**

super is Eukaryote

class **Animal** (extends **Eukaryote**)  
add class method **movementRate()**

super is Animal

class **Insecta** (extends **Animal**)  
add class method **numberOfWings()**

class **Drosophila** (extends **Insecta**)  
add class method **preferredFruit()**

super is Insecta

The next class up the hierarchy is the superclass (there can only be one).  
Each class down one level in the hierarchy (there can be more than one)  
is a subclass.

# Exception Handling

What if you want to enforce that a `Date` has integer values for day, month, and year?

```
class Date:
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year
```

```
myDate = Date("Ides", "March", "XLIV BC")
```

Does this code crash?

# Checking command line arguments

```
import sys
```

```
intval = int(sys.argv[1])
```

How could you check that the user entered a valid argument?

```
import sys
```

```
try:
```

```
    intval = int(sys.argv[1])
```

```
except:
```

```
    print "first argument must be parseable as an int value"  
    sys.exit()
```

two new reserved key  
words - try and except

You can put `try-except` clauses anywhere.

Python provides several kinds of exceptions (each of which is of course a class!). Some common exception classes:

`ZeroDivisionError` # when you try to divide by zero

`NameError` # when a variable name can't be found

`MemoryError` # when program runs out of memory

`ValueError` # when `int()` or `float()` can't parse a value

`IndexError` # when a list or string index is out of range

`KeyError` # when a dictionary key isn't found

`ImportError` # when a module import fails

`SyntaxError` # when the code syntax is uninterpretable

(note - each of these is actually an extension of the base `Exception` class - any code shared by all of them can be written once for the `Exception` class!)

# Example - enforcing format in the `Date` class

```
class Date:
    def __init__(self, day, month, year):
        try:
            self.day = int(day)
        except ValueError:
            print 'Date constructor: day must be an int value'
        try:
            self.month = int(month)
        except ValueError:
            print 'Date constructor: month must be an int value'
        try:
            self.year = int(year)
        except ValueError:
            print 'Date constructor: year must be an int value'
```



indicates only catches this type of exception

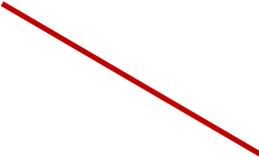
FYI, if there are other types of exceptions, they will be reported by the default Python exception handler, with output you are very familiar with, e.g.:

```
Traceback (most recent call last):
  File <pathname>, line X, in <module>
    <code line>
<default exception report>
```

You may even want to force a program exit with information about the offending line of code:

```
import traceback
import sys

class Date:
    def __init__(self, day, month, year):
        try:
            self.day = int(day)
        except ValueError:
            print 'Date constructor: day must be an int value'
            traceback.print_exc()
            sys.exit()
```



special traceback function  
that prints other information  
for the exception

# Create your own `Exception` class

```
import exceptions

class DayFormatException(exceptions.Exception):
    def __str__(self):
        print 'Day must be parseable as an int value'
```

## What does this mean?

`DayFormat` extends the Python defined `Exception` class

Remember that the `__str__()` function is what `print` calls when you try to print an object.

# Using your own Exceptions

```
class Date:
    def __init__(self, day, month, year):
        try:
            self.day = int(day)
        except:
            raise DayFormatException
```

`raise` is a new reserved key word - it raises an exception. The `DayFormatException` will get returned to wherever the constructor was called - there it can be "caught"

```
try:
    myDate = Date("Ides", "March", "IXIV")
except:
    <do something>
```

catch the exception raised  
by the Date constructor

# Exceptions - when to use

- Any software that will be given to someone else, especially if they don't know Python.
- Private software that is complex enough to warrant.
- Just as with code comments, exceptions are a useful way of reminding yourself of what the program expects.
- They have NO computational cost (if no exception is thrown, nothing at all is computed).

Imagine some poor schmuck's frustration when they try to use your program:

```
import sys
val = int(sys.argv[1])
```

```
> parse_int.py hello
Traceback (most recent call last):
  File "C:\Documents and Settings\jht\My Documents\parse_int.py", line 3, in <module>
    val = int(sys.argv[1])
ValueError: invalid literal for int() with base 10: 'hello'
```

what the @!#&\$!

by the way, notice that the  
ValueError is an exception class

```
import sys
try:
    val = int(sys.argv[1])
except ValueError:
    print "first argument '" + sys.argv[1] + "' is not a valid integer"
except IndexError:
    print "one integer argument required"
```

```
> parse_int.py
one integer argument required
> parse_int.py hello
first argument 'hello' is not a valid integer
```

hey, nice feedback!

# Exercise 1

Write a program `check_args.py` that gets two command line arguments and checks that the first represents a valid int number and that the second one represents a valid float number. Make useful feedback if they are not.

```
> python check_args.py 3 help!  
help! is not a valid second argument, expected a  
float value  
> python check_args.py I_need_somebody 3.756453  
I_need_somebody is not a valid first argument,  
expected an int value
```

```
import sys

try:
    arg1 = int(sys.argv[1])
except ValueError:
    print "'sys.argv[1]' is not a valid first argument,
expected an int value"
    sys.exit()
try:
    arg2 = int(sys.argv[2])
except ValueError:
    print "'sys.argv[2]' is not a valid second argument,
expected a float value"
    sys.exit()

<do something with the arguments>
```

# Exercise 2

Write a class `fastaDNA` that represents a fasta DNA sequence, with the name and the sequence itself stored as class variables (members). In the constructor, use exception handling to check that the sequence is valid (consists only of the characters 'A', 'C', 'G', 'T', or 'N' (either upper or lower case)). Provide useful feedback if not.

```
import sys
import re
class fastaDNA:
    __init__(self, name, sequence):
        self.name = name
        match = re.match('[^ACGTNacgtn]')
        if match != None:
            print sequence, 'is an invalid DNA sequence'
            sys.exit()
        self.sequence = sequence
```

# Challenge Exercises

Rewrite Exercise 2 using proper Exception handling.

Change your class definition from Exercise 2 so that it provides useful traceback information so that you can find where in your code you went wrong.