Jython

Introduction to Jython programming

Agenda

- Module 1 - Introduction to Jython
- Module 2 - Jython language and semantics
- Module 3 - Data types
- Module 4 - Regular expressions
- Module 5 - Functions, debugging, modules, and packages
- **Module 6 - Objects, classes and exceptions**
- Module 7 - Java integration
- Module 8 - Testing
- Module 9 - System programming
- Module 10 - Conclusion

Topics

- OOP concepts
- Classes
- Creating classes
- Exception syntax
- More exceptions
- Quiz
- Q & A

Object orientation

- Python is an O-O language
  - Everything is an object of some type
  - All objects eventually derived from
    - “object”
    - “type”

Objects

```
>>> list
<type 'list'>
>>> list.__class__
<type 'type'>
>>> list.__bases__
(<type 'object'>,)
>>> tuple.__class__, tuple.__bases__
(<type 'type'>, (<type 'object'>,))
```

built-in <type 'list'>

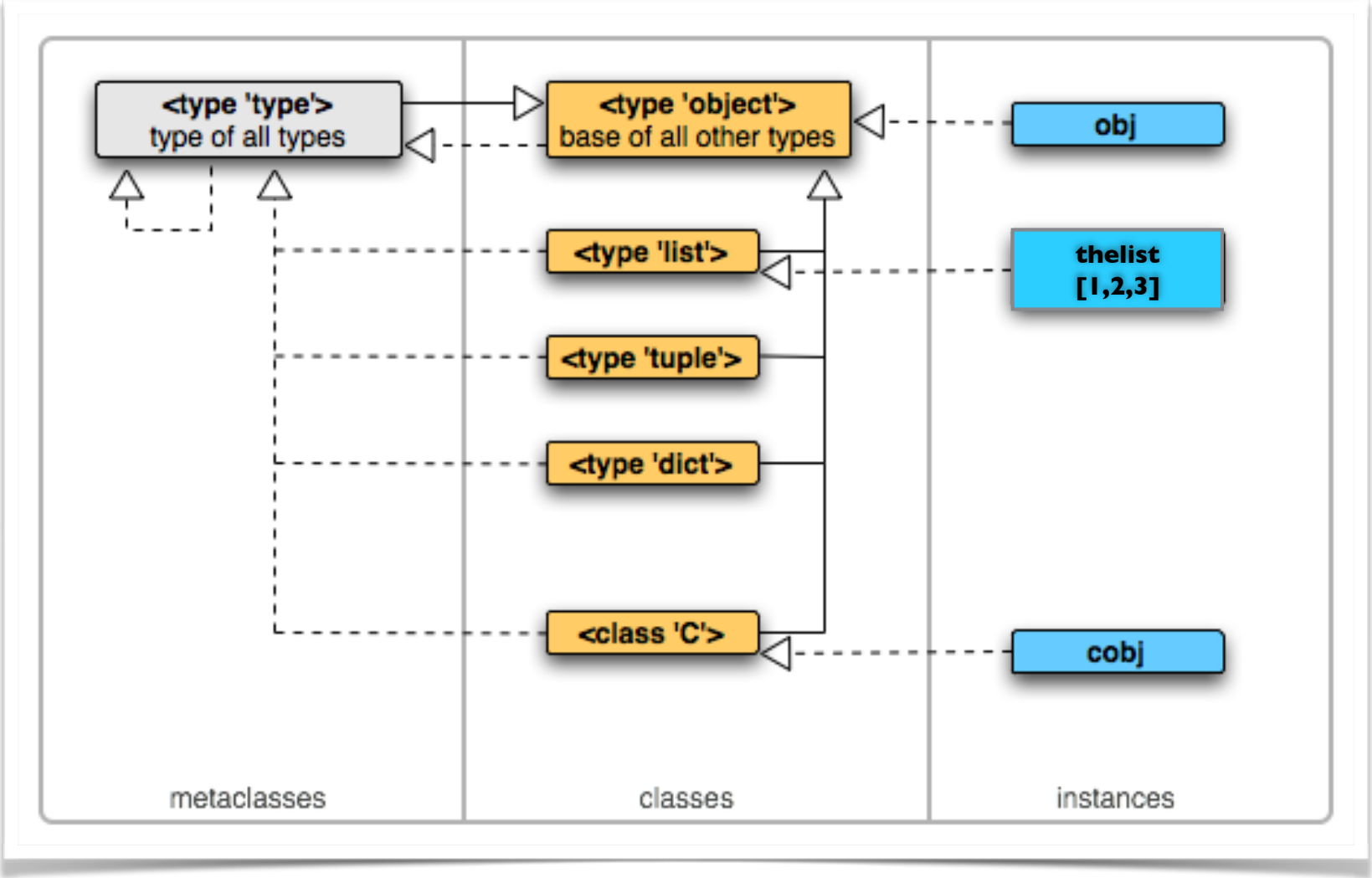
one superclass (base) - object

the "list" class is a "type"

```
>>> thelist = [1,2,3]
>>> thelist.__class__
<type 'list'>
>>>
```

Create a variable of type 'list'

Type map



Object testing syntax

- from types import type

```
>>> from types import *
>>> i = 1
>>> type(i) is IntType
True
>>>
```

deprecated method

- isinstance(variable, type)

```
>>> isinstance(i, int)
True
>>>
```

preferred method

OOP

- OOP is a design decision
  - How to use a class to model useful objects
- Core concepts
  - Inheritance - attribute lookup
  - Polymorphism - meaning of method depends on type of class
  - Encapsulation - methods and operators implement behavior; data hiding is a convention (default)

Reflection

Discover information about an object

- Everything is an object
  - Get information about the object in memory

Classes

Overview

- Compound statement
  - Indented statements
    - Names
      - Class methods (function names)
- Mixture of the class mechanism from C++ and Module-3
- Multiple base classes
- Derived class can override any methods of classes of all base classes
- Method can call a method of a base class with the same name
- Created at runtime
- Classes are aways nested within a module

Names and objects

- Objects have individuality
- Multiple names within scopes
  - Aliasing
- class statements are local scopes
  - Names in a class statement become attributes in a class object
- Class namespaces are the basis for inheritance

Inheritance

Inheritance customization

- Superclasses listed in parenthesis in the class header (declaration)

```
class D(B,C):
```

- Classes inherit attributes from superclasses
- Instances inherit attributes from all accessible classes
  - Each *object.attribute* reference invokes a new independent search

Name resolution

- When looking for a name that references a type
  - Check the instance of a class
  - Check the class
  - All superclasses

Attributes

- Any name following a "."
  - z.real - real is an attribute of the instance of z
- Can be used as class level constants
- Assignment of names inside the class statement creates attributes

Class methods

- Nested "*def*" statements within the class statement
  - Provide behavior for an instance of a class
- Work the same as a function in a module or other file
  - A methods first argument is receives the instance object
    - Implied subject of the method

```
class C(object):
    def __init__(self, arg=None):
        self.arg = arg
```

Private functions

- Private elements
  - Functions - cannot be called from outside the module
  - Class methods - cannot be called from outside the class
  - Attributes - cannot be accessed from outside the class
- Determined by name
  - Starts with "__" (double underscore)
    - *private*
  - Everything else is *public*

Name mangling

- Default is public access
  - Private can be created by using a leading "_"
  - Double "_" gets mangled with the class name

```
>>> class test():
...     def __init__(self):
...         self.var = 1
...         self._var = 2
...         self.__var = 3
...
>>> x = test()
>>> dir(x)
['__doc__', '__init__', '__module__', '_test__var', '_var', 'var']
>>>
```

Private

Mangled

Public

- All can be accessed

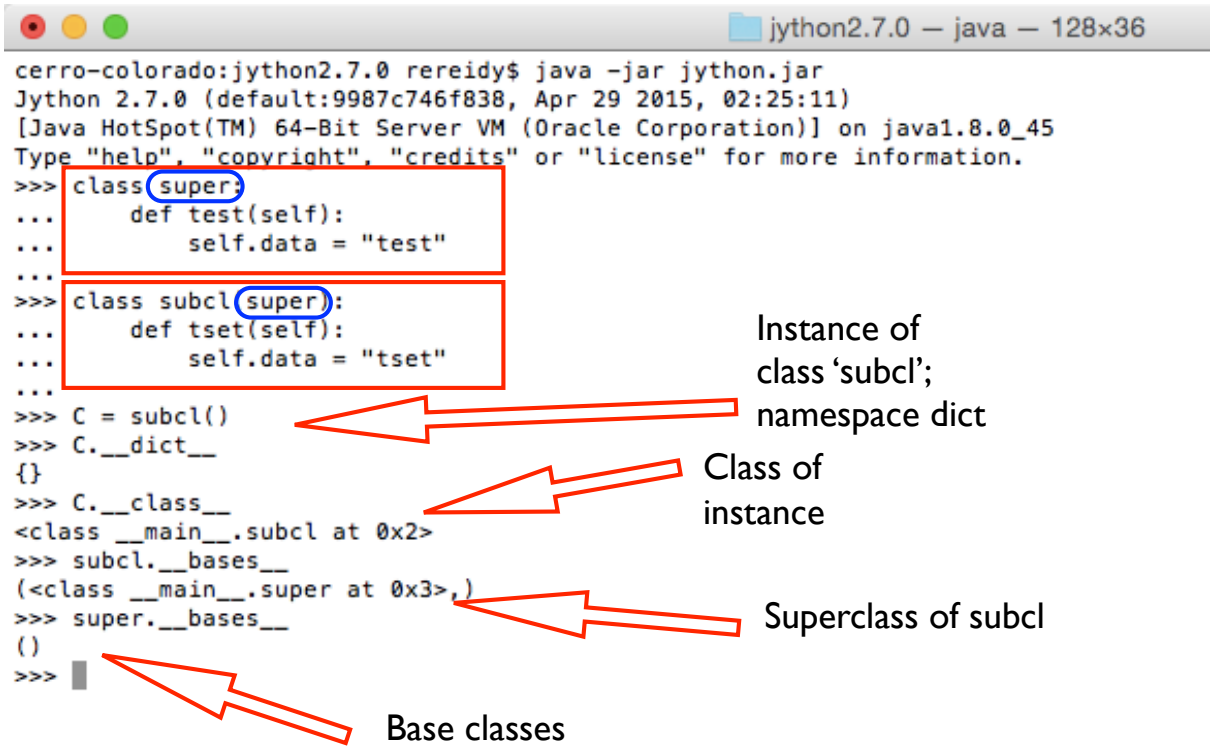
```
>>> print x.var, x._var, x.__var
1 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: test instance has no attribute '__var'
>>> print x.var, x._var, x._test__var
1 2 3
>>>
```

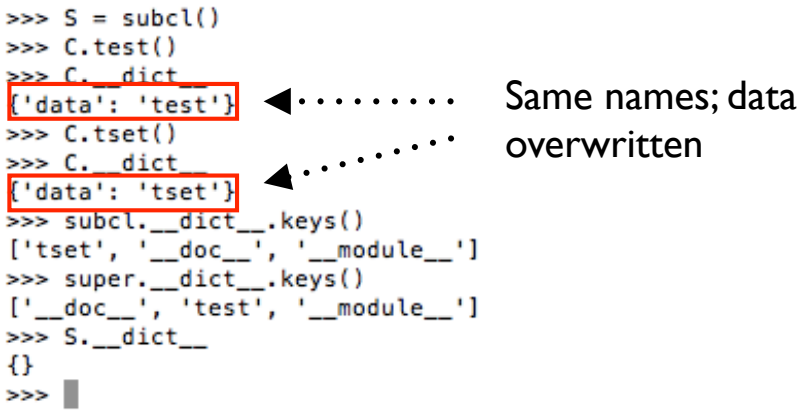
Access permitted

Classes vs. modules

- Modules
  - Data & logic packages
  - Created with Python files or Java classes
  - Used by being imported
- Classes
  - Implement new objects
  - Created by *class* statement
  - Used by being called
  - Always live within a module

Class level special variables





__doc__

- First statement of a module - docstring
- Necessity of docstring
  - All modules (files)
  - All functions and classes exported by a module
    - Public methods
  - Packages should include in `__init__.py`
- Used by help() in the interpreter

Two styles

- One line docstring

```
"""Return True when found"""
```

- Multi-line docstring

```
"""
Create a complex number
Args:
real - the real portion (default 0.0)
imag - the imaginary portion (default 0.0)
"""
```

https://www.python.org/dev/peps/pep-0257/ - conventions

https://www.python.org/dev/peps/pep-0258/ - style guide

Special methods

Jython

Characteristics

- Classes implement certain operations invoked by special syntax
  - Arithmetic
  - Subscripting/slicing
- How overloading is implemented

Notes

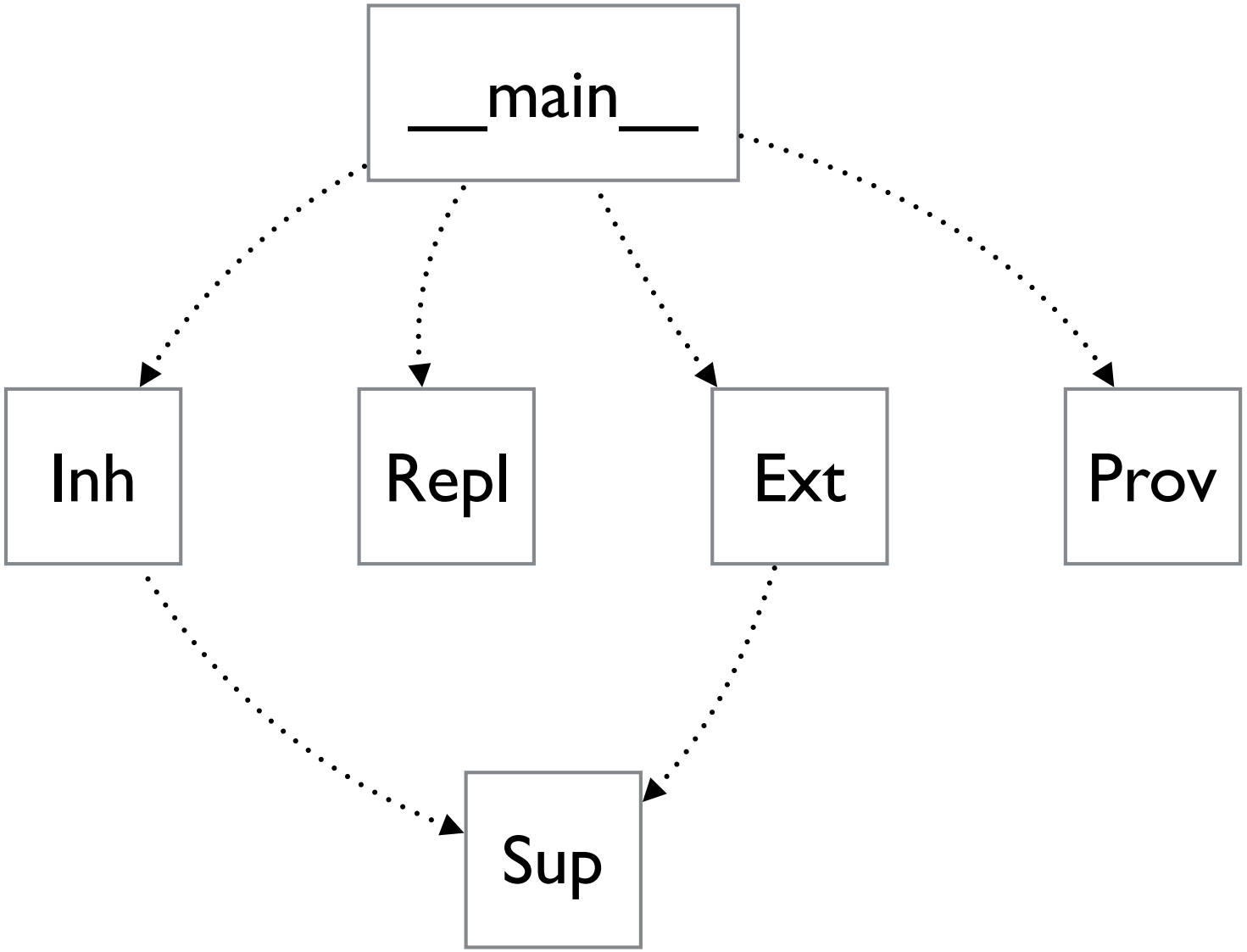
- There are many special methods implemented and available for customization in classes
  - Basic customization
  - Attribute access
  - Descriptors
  - Class creation
  - Emulation
    - Callable objects
    - Container types
    - Sequence types
    - Numeric types
  - Coercion rules
  - Context managers

https://docs.python.org/2/reference/datamodel.html

Inheritance

Namespace in a class

- Inheritance happens when an object attribute is qualified
  - Python searches namespaces from top to bottom



Abstract classes

- Abstract classes expect subclasses to define behavior
  - If not defined in the subclass
    - Raise UndefinedName

Operator overloading magic methods

Basics

- Intercepting built-in operations in class methods
  - Normal Python operations
  - Any Python expression operators
  - Any built-in operations (print, function calls, attribute access, etc.)
- Allows class instances to act more like a built-in type
- Implemented by providing specially named class methods

Classifications

- Constructor and initialization
- Comparison methods
- Numeric
- Class representation
- Attribute access control
- Reflection
- Callable
- Context management
- Descriptor objects
- Copying
- Serialization

Construction

__init__()

- Class constructor
- Called after the instance of the class is created, but before returned to the caller
- Arguments are passed to the class when created

Superclass constructors

- Methods normally called through an instance of the class
- Python calls only one constructor (__init__()) when instance constructed
- Subclasses need to guarantee super class constructors are called

__del__()

- Class destructor
- Called when the class instance is about to be destroyed
  - Does not implement behavior of "del x"
  - Defines behavior during garbage collection
- Exceptions that occur during invocation of __del__() are ignored
  - Warnings are printed so sys.stderr

__repr__()

- Compute the "official" string representation of an object
- Called by
  - *`repr()`* built-in function
  - String conversions
- Must return a string object
- Typically used for debugging
  - Information rich and unambiguous

__str__()

- Computer the "informal" string representation of the object
- Called by the built-in str() function
- Must return a string object
- If `__repr__()` is defined but not `__str__()`
  - `__repr__()` is also the "informal" string representation of the instance

Comparison methods

Comparison methods

- Called in preference of *__cmp__()* (next slide)
- object.__lt__(self, other)
  - x < y calls x.__lt__(y)
- object.__le__(self, other)
  - x <= y calls x.__le__(y)
- object.__eq__(self, other)
  - x == y calls x.__eq__(y)
- object.__ne__(self, other)
  - x != y calls x.__ne__(y)
- object.__gt__(self, other)
  - x > y calls x.__gt__(y)
- object.__ge__(self, other)
  - x >= y calls x.__ge__(y)

__cmp__()

- Called for comparison operations if rich comparison operators not defined
- Similar to strncmp() in C

```
if self < other:
  return -1
elif self == other:
  return 0
elif self > other:
  return 1
```

__cmp__()2

- If no *__cmp__()*, *__eq__()*, or *__ne__()* operations defined
  - Class instances compared by object identity (address)

Numeric methods

- Five sub categories of numeric methods
  - Unary operators
  - Normal arithmetic operators
  - Reflected arithmetic operators
  - Augmented assignment operators
  - Type conversion methods

http://www.rafekettler.com/magicmethods.html#numeric

Class representation methods

- Implement string representation of the class

http://www.rafekettler.com/magicmethods.html#representations

Attribute access control

- Get, set, and delete attributes

http://www.rafekettler.com/magicmethods.html#access

Reflection

- Control of how reflection is implemented in the class

http://www.rafekettler.com/magicmethods.html#reflection

Callable

- Allows instances of a class to behave like functions
  - Call as a function
  - Pass to a function as an argument

http://www.rafekettler.com/magicmethods.html#callable

Context management

- Allows setup and cleanup actions for objects
  - When wrapped with a “with” statement

http://www.rafekettler.com/magicmethods.html#context

Descriptor objects

- Descriptor objects are classes when accessed (get, set, or delete)
  - Also alter other objects

http://www.rafekettler.com/magicmethods.html#descriptor

Copying

- Copy an object
  - Make changes to the copy without affecting the original object

http://www.rafekettler.com/magicmethods.html#copying

Serialization

- Serialize data structures

http://www.rafekettler.com/magicmethods.html#pickling

Exception syntax

Jython

Note

- The Python VM can raise exceptions at
  - Compile time
  - Runtime
- Exceptions can be raised programmatically
- When raised
  - Execution is stopped
  - Control passed to the nearest exception handler in the program stack

Care must be taken when designing exceptions to perform proper clean up while also producing error messages that are relevant.

Implementing exceptions

Jython

- Syntax

```
try:
    statement
        ...
except NamedException [, alias]:
    statement
        ...
except NamedException [, alias]:
    statement
        ...
except [Exception [, alias]:
    statement
        ...
else:
    statement
        ...
finally:
    statement
        ...
```

Built in exceptions

- Many built-in exceptions
- Implemented in the exceptions module
  - Does not need to imported
    - Provided in the built-in namespace
- Can be implemented by
  - Interpreter
  - Built-in functions
  - User code (and classes)
- Can be subclassed to define new exceptions
  - Use the Exception class or a subclass
    - Do not subclass BaseException

https://docs.python.org/2/library/exceptions.html

Exception hierarchy

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
```

Rules for exceptions

- Exception handlers should not be used for processing tasks
  - Exceptions are for errors
- `try/except` blocks can be nested
- Exceptions can be re-reaised
  - raise statement raises the exception to the block enclosing the `try/except` block

Module: traceback

- Extract, format, print stack traces of python programs
- Mimics the Python interpreter exactly

https://docs.python.org/2/library/traceback.html

User defined exceptions

- Create custom exceptions by creating a new exception class
- Exception classes can be defined which do anything any other class can do
  - Keep it simple
  - Only define attributes that return information about the error to be extracted by the exception handler

Quiz

Jython

1. When is the __init__() method called?

A. Just after the object is created

2. What is the purpose of __init__?

A. Initialize the module package

Jython

3. What is the relationship between classes and modules?

A. Classes are nested within modules

Exercises

1. Create a program with a custom exception class. In the program, create a try/except block and raise the custom exception.

2. Modify the program in #1, adding else and finally to the exception block.

Jython

3. Create a class which will count the lines in a text file. The class should have a method to retrieve the line count.

   Do not use the os.system() call or other methods to create a sub shell to do this task.

Jython

4. Modify the program in #3 to return the actual number of bytes in the file.

   Create a new method for this information.