PYTHON

QUICK INTRODUCTION

- Python was conceived in the late 1980s and its implementation was started in December 1989 - **Guido Van Rossum**
- Name python comes from the TV show "Monty Python's flying circus" from 1969

LANGUAGE SUMMARY

- Python is an interpreted programming language.
- Object-oriented (Objects are first class citizens)
- Dynamic typing

HOW TO GET IT

- www.python.org/download/

PEP 8 - STYLE GUIDE FOR PYTHON CODE

- **http://legacy.python.org/dev/peps/pep-0008/**

PYTHON VERSIONS

- Currently in a transition between python2 and python3
- Python 3 is not backwards compatible
- Python 2.7.6 last version of python2 series (Nov 10, 2013)
- Python 3.3.5 last version of python3 series (March 9, 2014)

INTERACTIVE MODE

- Running python interpreter in a terminal
- Much cooler interactive mode called ipython

```
$ python3.3
Python 3.3 (default, Sep 24 2012, 09:25:04)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- Sentences are executed inmediatly
- If sentece result is an expression, it will be printed on screen
- Last printed expression is assigned to the variable _
- "print" used to be a keyword, it's a builtin function since python2.6
- help(object) displays documentation
- Closing the interpreter
  - ctrl-d, ctrl-z
  - raise SystemExit
  - import sys; sys.exit(code)
  - On execite a file, finish when it gets to the end

ADVANTAGES

- Variables are just names assigned to values
  - Operator (=) links a name with a value
  - Same name can be set to different value type
  - Type information is in the value, not in the name
- INDENTATION IS PYTHON'S WAY OF GROUPING STATEMENTS AND DEFINING BLOCKS
  - Also known as off-side rule

CONDITIONALS

```
if boolean_expression:
    block
elif another_boolean_expression:
    block
elif ...
    block
else:
    block
```

- no switch/case at all!

STRINGS

- Python strings cannot be changed — they are immutable
- Strings can be indexed, with the first character having index 0
- Slicing is also supported
- While indexing is used to obtain individual characters, slicing allows you to obtain substring [start:stop:step]

```
>>> a="hello world"
>>> a[0]
'h'
>>> a[0:3]
'hel'
>>> a[-1]
'd'
>>> a[::2]
'hlowrd'
>>> a[0]='b'
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

STRINGS(2)

- Python is more strongly typed than other languages like Perl or PHP. No automatic type promotion supported for strings
- Objects can be converted to string representations
- repr(obj) (string representation of object - serialization)
- str(obj) (value to be diplayed by print(obj) - intended to be more human-readable than `__repr__`)
- From python 2.6 format method, similar to C's printf

```
'{0} %s {1}'.format('foo', 'bar')
```

```
>>> print(a + ", how're you doin'")
hello world, how're you doin'
>>> a + 16
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects

>>> a + '16' + str(16)
'hello world1616'
```

LISTS

- Lists are a mutable type, it is possible to change their content
- Lists might contain items of different types
- Literal List construction is supported

```
>>> a = ['hello', 'world', 1, 2]
>>> a[0]
'hello'
>>> a[2:] # slicing returns a new list
[1, 2]
>>> a[1] = False
>>> a
['hello', False, 1, 2]
>>> a[4]="extra"
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

```
>>> a
['hello', False, 1, 2]
>>> a.append("extra")
>>> a
['hello', False, 1, 2, 'extra']
>>> a.pop(0)
'hello'
>>> a
[1, 2, 'extra']
>>> a + [3,4,5] # lists support concatenation
[1, 2, 'extra', 3, 4, 5]
>>> [] == list()
True
>>> a = [1,2,[0,1,2],"End"]
```

TUPLES

- Similar to Lists, but immutable

```
>>> t = ('hello', 1, 2, 3)
>>> t[0] = 10
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> t2 = 'hello', 1, 2, 3
>>> t2 == t
True
```

DICTIONARIES

- Also known as Maps, Hashes, key-value stores.
- Literal way: {key: value, ...}

```
import datetime
user = {
    'name': 'Walter',
    'company': 'OrderGroove',
    'start_date': datetime.date(2014, 03, 03)
}
>>> user
{'name': 'Walter', 'company': 'OrderGroove', 'start_date': datetime.date(2014, 03, 03)}
>>> user['name']
'Walter'
>>> user.keys()
['name', 'company', 'start_date']
>>> user.values()
['Walter', 'OrderGroove', datetime.date(2010, 5, 3)]
>>> user['language'] = ['python']
>>> user
{'name': 'Walter', 'company': 'OrderGroove', 'start_date': datetime.date(2010, 5, 3),
'language': ['python']}
>>> user['extra_data']
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
KeyError: 'extra_data'

>>> user.get('extra_data', [])
[]
```

SETS

- Unordered collections of unique elements
- Common uses include removing duplicates from a sequence and computing standard math operations on sets such as intersection, union, etc.

```
>>> s = set([1,2,3]) # no literal constrcution on 2.6
>>> s = {1,2,3} # since python 2.7+
>>> len(s)
3
>>> 2 in s and not (10 in s)
True
>>> s.remove(1)
>>> s.add(5)
>>> s
set([2, 3, 5])
>>> s.add(2)
>>> s
set([2, 3, 5])
>>> s.union({True, 'hello'})
set([True, 2, 3, 5, 'hello'])
```

LOOPS & ITERATIONS

- Most popular loop is the "for loop" (WhileLoop as alternative)
- Takes an iterable object (explained later) and execute the block for each iteration
- Sequences, as other types, allow iterations

```
>>>for i in [5,6]
...  for j in [10,18]:
...    print(gcd(i,j))
...
5
1
2
6
```

MORE EXAMPLES

```
>>> for c in "hello":
...   print(c)
...
h
e
l
l
o
```

```
>>> d = {'name': 'Pete', 'company': 'OrderGroove'}
>>> for k in d:
...   print(k,d[k])
...
('name', 'Pete')
('company', 'OrderGroove')
```

```
mounts = open('/etc/fstab', 'r')
for mount in mounts:
    print mount
```

LINES STRUCTURE

- Any sentence can end with a "end of line" or a semicolon (;)

```
spam = 1
eggs = 2
```

```
spam = 1; eggs = 2
```

```
# not commonly used
spam = 1;
eggs = 2;
```

INDENTATION

- Blocks are defined by the indentation level
- Sentences that end with (:) define a new block
- You can use spaces and/or tabs to indent. Avoid tabs
- 4 spaces indentation is commonly used (multiple of 4)
- Empty blocks are not allowed. Empty sentence "pass" should be used

```
if a:
    foo
    bar
elif b:
    pass
elif c:
    eggs
        spam    # error
else:
            ugly  # but valid!
```

COMMENTS & DOCUMENTATION

- Hush symbol (#) indicates the begining of a comment
- Comments do not have to be indented (but its better if they are)
- If the first sentence of a module, class or function is a literal string, compiler takes is the object documentation

```
>>> def with_doc(): # un comentario
...     "this is documentation"
...     pass
...
>>> help(with_doc)
Help on function with_doc in module __main__:

with_doc()
    this is documentation
```

LITERAL BOOLEANS

- True == bool(1)
- False == bool(0)
- They are not reserved words in python2!

```
>>> True
True
>>> type(True)
<type 'bool'>
>>> True = "nasty"
>>> True
nasty
>>> True = False #real nasty
>>> if not True:
        print("impossible")
impossible
>>> True = bool(1) #restore
```

```
>>> int(True)
1
>>> int(False)
0
>>> True + 2
3
>>> bool.__bases__
(<type 'int'>,)
```

LITERAL STRINGS

```
>>> print("Simple")
simple
>>> print("how're you doing")
how're you doing
>>> print('"Thank you"')
"Thank you"
>>> print("""Multi line
... string""")
Multi line\nstring
>>> print("Hello " 'world')
Hello world
```

FUNCTIONS

- The keyword "def" introduces a function definition
- It must be followed by the function name and the parenthesized list of formal parameters
- The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring
- Calling a function with wrong args number, will raise a TypeError
- If a return statement is not present, None value is returned

```
>>> def add(x,y):
...     return x+y
...
>>> add(10,20) == 30
True
```

```
>>> add()
TypeError: add() takes
exactly 2 arguments (0 given)
>>> add(1,2,3)
TypeError: add() takes
exactly 2 arguments (3 given)
```

```
def just_do_something(l):
    l.append(10)

>>> just_do_something([1]) == [1,10]
False
>>> just_do_something([1]) is None
True
```

DEFAULT PARAMETER VALUES

```
>>> def defaults(a,b=10,c=False):
...     print(a,b,c)
...
>>> defaults(5)
(5, 10, False)
>>> defaults()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: defaults() takes at least 1 argument (0 given)
>>> defaults(1,2,3)
(1, 2, 3)
>>> defaults(1,c=3)
(1, 10, 3)
```

- If a parameter has a default value, all following parameters must also have a default value
- Be careful when using a “mutable” object as a **default value**

VARIABLE-LENGTH ARGUMENTS

- You may need to process a function for more arguments than you specified
- By placing an asterisk (*) before the variable name
- If *args is defined, name args is asociated to a tuple (0+ arguments)
- Variable-length of keyword arguments is supported with (**) prefix
- If **kwargs is defined, variable kwargs is dictionary (key:value)
- You can combine `*args and **kwargs`

```
>>> def varargs(a, b=None, *args, **kwargs):
...    print(a,b,args, kwargs)
...
>>> varargs(10)
(10, None, (), {})
>>> varargs(10, 'next')
(10, 'next', (), {})
>>> varargs(1,2,3,4,5,6,another=7)
(1, 2, (3, 4, 5, 6), {'another': 7})
```

PASS BY REFERENCE VS VALUE

- All parameters (arguments) in the Python are passed by reference
- Is that so? Well, the parameter passed in is actually a reference to a variable (but the reference is passed by value) What??

Passing a mutable object into a method

```
def try_to_change_list_contents(the_list):
    print 'got', the_list
    the_list.append('four')
    print 'changed to', the_list

outer_list = ['one', 'two', 'three']

print 'before, outer_list =', outer_list
try_to_change_list_contents(outer_list)
print 'after, outer_list =', outer_list
```

Output:

```
before, outer_list = ['one', 'two', 'three']
got ['one', 'two', 'three']
changed to ['one', 'two', 'three', 'four']
after, outer_list = ['one', 'two', 'three', 'four']
```

Now let's see what happens when we try to change the reference that was passed in as a parameter:

```
def try_to_change_list_reference(the_list):
    print 'got', the_list
    the_list = ['and', 'we', 'can', 'not', 'lie']
    print 'set to', the_list

outer_list = ['we', 'like', 'proper', 'English']

print 'before, outer_list =', outer_list
try_to_change_list_reference(outer_list)
print 'after, outer_list =', outer_list
```

Output:

```
before, outer_list = ['we', 'like', 'proper', 'English']
got ['we', 'like', 'proper', 'English']
set to ['and', 'we', 'can', 'not', 'lie']
after, outer_list = ['we', 'like', 'proper', 'English']
```

"Since the the_list parameter was passed by value, assigning a new list to it had no effect that the code outside the method could see"

Passing a immutable object into a method

```
def try_to_change_string_reference(the_string):
    print 'got', the_string
    the_string = 'In a kingdom by the sea'
    print 'set to', the_string

outer_string = 'It was many and many a year ago'

print 'before, outer_string =', outer_string
try_to_change_string_reference(outer_string)
print 'after, outer_string =', outer_string
```

Output:

```
before, outer_string = It was many and many a year ago
got It was many and many a year ago
set to In a kingdom by the sea
after, outer_string = It was many and many a year ago
```

SCOPES

- At any time during execution, there are at least three nested scopes whose namespaces are directly accessible: local, global, built-in
- Built-in is provied by the interpreter (True, max(), object, type, etc)
- Python searches on the innermost scope first, module's global names next, and built-in names at last

SCOPE IN FUNCTIONS

- If a name is bound at the module level, it is a global variable
- If a name is bound in a block, it is a local variable of that block
- If the global statement occurs within a block, all uses of the name specified in the statement refer to the binding of that name in the top-level namespace.

```
>>> a=10
>>> def f():
...     a=20
...
>>> f()
>>> a
10
>>> def g():
...     global a
...     a=20
...
>>> g()
>>> a
20
```

FUNCTIONS AS FIRST-CLASS CITIZENS

- A function can be defined on almost any point in the code:
  - As a Method in a Class (most common case)
  - As a function within a module (second most common case)
  - As a nasted function (Closures, third most common case)
  - Sentence **def** is used for those case (we will see an alternative)

```
>>> def f1():
...     pass
...
>>> type(f)
<type 'function'>
>>> class withMethods(object):
...     def f2(self):
...         pass
...     f3 = f1
...
>>> withMethods.f2
<unbound method withMethods.f2>
>>> withMethods.f3
<unbound method withMethods.f1>
```

CLOSURES

- A closure occurs when a function has access to a local variable from an enclosing scope that has finished its execution

```
>>> def makeInc(x):
        def inc(y):
            # x is "closed" in the definition of inc
            return y + x

        return inc

>>> inc5 = makeInc(5)
>>> inc10 = makeInc(10)

>>> inc5(5)   # returns 10
>>> inc10(5)  # returns 15
```

CLOSURES (2)

- The nonlocal statement is a close cousin to global
- nonlocal applies to a name in an enclosing function's scope, not the global module scope

```
>>> def outer():
        x = 1
        def inner():
            nonlocal x
            x = 2
            print("inner:", x)
        inner()
        print("outer:", x)

>>> outer()
inner: 2
outer: 2
```

DECORATORS

- Based on the Decorator Pattern
- Decorators allow you to inject or modify code in functions or classes (sounds like Aspect-Oriented Programming (AOP) in Java, doesn't it?)
- The @ indicates the application of the decorator (don't confuse with java annotations)
- Decorators help reducing boilerplate code

```
>>> def makebold(fn):
        def wrapped():
            return "<b>" + fn() + "</b>"
        return wrapped

>>> def makeitalic(fn):
        def wrapped():
            return "<i>" + fn() + "</i>"
        return wrapped

    @makebold
    @makeitalic
    def hello():
        return "hello world"

>>> print hello() ## returns <b><i>hello world</i></b>
```

ITERATOR PROTOCOL AND THE FOR LOOP

- Iterator objects in python conform to the iterator protocol, which basically means they provide two methods: `__iter__()` and `next()`
- The `__iter__` returns the iterator object
- The next() method returns the next value
- If there are no further items, raise the StopIteration exception
- The For loop is based on this iterator protocol

```
some_list = [1,2,3,4,5]
def sum(l, default=0):
    total=default
    for x in l:
        total += x
    return total
def sum2(l, default=0):
    total=default
    iterable = iter(l)
    try:
        while True:
            x = next(iterable)
            total += x
    except StopIteration:
        pass
```

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()  # __next__ from Python 3
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    it.next()
StopIteration
```

```
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def next(self): # Python 3: def __next__(self)
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

for c in Counter(3, 8):
    print c
```

GENERATORS

- Generators are a simple and powerful tool for creating iterators
- Use the **yield** statement whenever they want to return data
- Using a **yield** expression in a function definition is sufficient to cause that definition to create a generator function instead of a normal function
- Each time next() is called, the generator resumes where it left-off (it remembers all the data values and which statement was last executed)
- What makes generators so compact is that the `__iter__()` and `next()` methods are created automatically
- Local variables and execution state are automatically saved between calls
- When generators terminate, they automatically raise StopIteration
- yield and return under the same def structure is not allowed!

```
def counter(low, high):
    current = low
    while current <= high:
        yield current
        current += 1

for c in counter(3, 8):
    print c
```

```
>>> def up_to(n, start=0):
...     current = start
...     while (current < n):
...         yield current
...         current+=1
...
...
>>> count = up_to(3)
>>> next(count)
0
>>> next(count)
1
>>> next(count)
2
>>> next(count)
StopIteration:     Traceback ...
```

PYTHON

DAY 2

CONSTRUCCIÓN DE CLASSES VIA **CLASS**

- el statement class crea un nuevo namespace
- el bloque de código dentro de un class tiene como scope local ese nuevo namespace
- el scope global es el módulo donde está definido
- el bloque se ejecuta al construirse la clase
- se pueden definir funciones y clases dentro del cuerpo de una clase
- al cerrarse el bloque, una clase queda compuesta por los valores enlazados en el scope local. (atributos de clase)
- las funciones referenciadas son promovidas a métodos

```
class Nombre(object):
    """docstring"""
    some_var=True
    another=0
    def f(self):
        pass

    for x in range(10): #unusual, valid
        another += x
```

INSTANCIAS E INICIALIZACIÓN

- Una clase es un objeto
- En particular, es un objeto que puede ser llamado (callable)
- Llamar a una clase construye una nueva instancia
- El método `__init__`, si está definido, se llama luego de creado el nuevo objeto

```
>>> class MyClass(object):
...     def __init__(self, some_number):
...         self.number = some_number
...         print('initializing')
...
>>>
>>> i = MyClass(10)
initializing
>>> i.number
10
>>>
```

ATRIBUTOS

- El namespace que se genera al construir una clase se implementa con un diccionario (`clase.__dict__`)
- MiClase.atributo se busca via `MiClase.__dict__['atributo']`
- Si un atributo no se encuentra en la clase, se busca en sus subclases (Method resolution order, C3)
- Las asignaciones siempre suceden en el diccionario de la clase, no en sus bases
- Hay una serie de hooks para interceptar el acceso a atributos (getattr, descriptors)
- Las instancias construyen su propio namespace
- La búsqueda de atributos de una instancia comienza por el diccionario de la instancia, sigue en su clase y bases

```
>>> class Attrs(object):
...     a=10
...     def method(self):
...         pass
...
>>> Attrs.a
10
>>> print(Attrs.__dict__)
{'a': 10, 'method': <function method at 0xb76fa7d4>, ...}
>>> instance = Attrs()
>>> instance.a
10
>>> instance.a=20
>>> instance.a
20
>>> Attrs.a
10
```

MÉTODOS

- Se construyen a partir de funciones
- El namespace de clase no está visible en el cuerpo de una función
- El primer argumento es provisto por el enlace entre un método y una instancia (instancemethod) o una clase (classmethod)
- Por defecto todas las funciones encontradas en un scope de clase se envuelven en instancemethods
- Un método puede estar bound o unbound
- StaticMethod es una función común anidada en una clase

```
>>> class Methods(object):
...     def method1(self, n):
...         print(n)
...         print(self)
...     @classmethod
...     def method2(cls, n):
...         print(cls)
...     @staticmethod
...     def method3(n):
...         print(n)
```

```
>>> m = Methods()
>>> m.method1(10)
10
<__main__.Methods object at 0xb7704d0c>
>>> m.method2(20)
<class '__main__.Methods'>
>>> m.method3(30)
30
>>> m.method1
<bound method Methods.method1 of
<__main__.Methods object at 0xb7704d0c>>
>>> m.method2
<bound method type.method2 of
<class '__main__.Methods'>>
>>> m.method3
<function method3 at 0xb76fae9c>
>>> Methods.method1
<unbound method Methods.method1>
>>> Methods.method2
<bound method type.method2 of
<class '__main__.Methods'>>
>>> Methods.method3
<function method3 at 0xb76fae9c>
```

HERENCIA

- Especificada como una tupla de bases
- La herencia extiende un espacio de nombres con el de las bases
- Los atributos se buscan en la jerarquía de clases antes de fallar con AttributeError
- `__mro__` muestra la secuencia de búsqueda para un objeto dado

```
>>> class base1(object):
...     a=10
>>> class base2(object):
...     a=20
...     b=30
>>> class inherit1(base1):
...      pass
...
>>> class inherit2(base1, base2):
...      pass
>>> inherit1.a
10
>>> inherit2.a
10
>>> inherit2.b
30
```

INTERCEPTAR ATRIBUTOS CON GETATTR, SETATTR

- Si una clase implementa `__getattr__` / `__setattr__` puede modificar el acceso a atributos
- Se llama a `__getattr__` luego de no encontrar el atributo por el método tradicional
- `__setattr__(self, name, value)` intercepta la asignación de atributos
- `__delattr__(self, name)` intercepta del object.attribute
- Alternativa al operador atributo (.) getattr(obj, attribute_name, [default])
- También `__setattr__(self, attr_name, value)`, hasattr(obj, attr_name)
- `__getattribute__(self, name)` es la forma incondicional de `__getattr__`

```
>>> class Intercept(object):
...     def __getattr__(self, attr):
...         return 'intercepted_' + attr
...     def __setattr__(self, attr, value):
...         print('refusing to set')
...
>>>
>>> i = Intercept()
>>> i.test
'intercepted_test'
>>> i.another
'intercepted_another'
>>> i.test=10
refusing to set
```

CLASSIC CLASSES Y NEW-STYLE CLASSES

- En python < 2.2, class construía un class object
- Las instancias de un class object tienen tipo class instances
- Distinción via atributo `__class__`
- En python2.2 se unificaron tipos y clases
- class MiClase(object) define un nuevo tipo
- Sus instancias son de tipo MiClase
- object pasó a ser la base de todos los objetos
- Los tipos builtin pueden ser heredados/extendidos
- No hay classic classes en Python3

```
>>> class classic:
...     pass

>>> class newstyle(object):
...     pass

>>> type(classic)
<type 'classobj'>
>>> type(newstyle)
<type 'type'>
>>> c = classic()
>>> n = newstyle()
>>> type(c)
<type 'instance'>
>>> type(n)
<class '__main__.newstyle'>
```

CLASS DECORATORS

- Equivalente a decoradores de funciones
- No están obligados a devolver una clase

```
>>> def add_attribute(name, value):
...     def wrap_cls(cls):
...         setattr(cls, name, value)
...         return cls
...     return wrap_cls
...
>>> @add_attribute('hello', 'world')
... class empty(object):
...     pass
...
>>> empty
<class '__main__.empty'>
>>> empty.hello
'world'
```

DESCRIPTORS

- Los descriptores permiten implementar atributos con acceso controlado
- Un descriptor implementa los accesos get, set y delete
- Deben ser new-style classes

```
class RevealAccess(object):
    """A data descriptor that sets and returns values
       normally and prints a message logging their access.
    """

    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, cls):
        print 'Retrieving', self.name
        return self.val

    def __set__(self, obj, val):
        print 'Updating', self.name
        self.val = val
```

DESCRIPTORS (2)

- Dos maneras típicas de acceder a un descriptor - Asociado a una instancia - Asociado a una clase

```
>>> class MyClass(object):
        x = RevealAccess(10, 'var "x"')
        y = 5

>>> m = MyClass()
>>> m.x
Retrieving var "x"
10
>>> m.x = 20
Updating var "x"
>>> m.x
Retrieving var "x"
20
>>> m.y
5
```

```
MyClass.x  # self=None, cls=MyClass
```

INSTANCIAS, __NEW__ Y __DELETE__

- Que se ejecuta cuando llamamos a una clase?
- __new__ es el real constructor de instancias
- Puede __new__ devolver algo diferente a una instancia de su clase?
- __new__ is for object creation - __init__ is for object initialization
- Por qué no se ejecuta __delete__ ?
- del es el opuesto a la asignación ('=') - desasocia un nombre de un valor - puede implicar que el garbage collector elimine el valor
- cuando del opera sobre un atributo, delega la operación al objeto que contiene el atributo instancia.attr

```
class Empty(object):
    pass

e1 = Empty()

#equivalente a
e2 = Empty.__new__(Empty)

if isinstance(e2,Empty):
    Empty.__init__(e2)
```

```
class MyClass(object):
    def __new__(cls, *args, **kwargs):
        return "string"


m = MyClass()
type(m)
<type 'str'>
```

TIPOS, CLASES, INSTANCEOF

- Todos los objetos tienen un tipo
- Para new style classes, el tipo de sus instancias es su clase
- Para classic classes (python2), el tipo es 'instance'
- type(object) devuelve el tipo del objeto
- isinstance(object, class) recorre la jerarquía de herencia y devuelve True si object tiene en sus bases a class
- alternativa: isinstance(object, (c1, c2, c3,...))
- issubclass chequea la jerarquía de herencia

```
>>> type(10)
<type 'int'>
>>> isinstance(10,(int, float))
True
>>> isinstance(10, type(20))
True
>>> issubclass(bool, int)
True
```

CONSTRUCCIÓN DE CLASES Y METACLASES

- Algunas cosas que pasan al construir una clase
- Funciones se convierten en métodos
- `__new__` se convierte en classmethod
- Las variables enlazadas en el cuerpo de una clase se convierten en atributos
- Suena a un algoritmo, podríamos implementar uno diferente?
- Metaclases son a las clases como las clases son a las instancias
- Las metaclases se implementan con clases (`__new__` e `__init__`). Generalmente heredan de type
- Se usa el atributo `__metaclass__` para especificar qué metaclase utilizar. (python2)
- Se usa class MiClase(base1, base2, metaclass=cls) en python3

```
    class CamelCase(type):
        @staticmethod
        def camel(word):
            return ''.join(s.capitalize() for s in word.split('_'))
        def __new__(meta, classname, bases, namespace):
            new_namespace = {}
            for k,v in namespace.items():
                if k.startswith('_'):
                    new_namespace[k] = v
                else:
                    new_namespace[meta.camel(k)] = v
            return type.__new__(meta, classname, bases, new_namespace)

    class Test1(object):
        __metaclass__ = CamelCase
        var1 = 10
        some_bool = False
        def my_method(self):
            pass
```

```
>>> hasattr(Test1, 'var1')
False
>>> hasattr(Test1, 'some_bool')
False
>>> hasattr(Test1, 'my_method')
False
>>> hasattr(Test1, 'Var1')
True
>>> hasattr(Test1, 'SomeBool')
True
>>> hasattr(Test1, 'MyMethod')
True
```

EXCEPTIONS

- Las excepciones en python son un mecanismo general para salir del flujo normal de ejecución
- No necesariamente indican un error (StopIteration en un iterador)
- Las excepciones se lanzan con la sentencia raise
- En las primeras versiones python, cualquier objeto se podía lanzar con raise
- Sólo se permiten Instancias descendientes de BaseException o bien old-style classes
- Si se lanza una clase en vez de una instancia, automáticamente se crea una instancia
- En ese caso, **init** debe tomar solo 1 argumento, self

RAISE - EJEMPLOS

```
>>> raise BaseException
Traceback (most recent call last):
File "<stdin>", line 1, in <module> BaseException

>>> raise 'hello'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: exceptions must be old-style classes or derived from BaseException, not str

>>> class old_style:
...    pass
...
>>> raise old_style
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
__main__.old_style: <__main__.old_style instance at 0xb765588c>
```

```
>>> class another(BaseException):
...     def __init__(self, some_argument):
...        pass
...
>>> raise another
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 2 arguments (1 given)
>>> raise another(1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module> __main__.another
```

CAPTURAR EXCEPCIONES

- Una excepción se puede capturar utilizando la construcción try ... except

```
>>> try:
 ...   1/0
 >>> except SyntaxError:
 ...   print('some syntax error')
 >>> except ZeroDivisionError:
       print('divison by 0')
```

- Except toma una tupla de clases y matchea una excepción via isisntance()
- Si la excepción no es capturada en el frame actual, se lanza en el frame superior
- Si no se captura la excepción en ningún frame, el intéprete termina

EXCEPT - SINTAXIS

- Se puede asociar el objeto excepción a un nombre en except, utilizando as name
- En versiones previas de python, la sintaxis era: except SyntaxError, e
- Puede haber un except sin clases, siempre al final, que captura todas las excepciones

```
>>> try:
...     raise OSError

>>> try:
...     raise OSError
>>> except SyntaxError:
...     pass
>>> except StandardError as e:
...     print(type(e))
...
<type 'exceptions.OSError'>
```

EXCEPTIONS - ELSE

- try except acepta un bloque else
- El bloque se ejecuta sólo si no ocurrió una excepción
- Qué diferencia hay con simplemente agregar el contenido de un bloque else a continuación del bloque try?
- Excepciones lanzadas en el bloque else no son capturadas por except

EXCEPTIONS - FINALLY

- se puede definir un bloque finally que se llama incondicionalmente al finalizar el bloque try (con o sin excepciones)
- En caso de haber una excepción, se llama luego del handler
- En caso de no excepción, se llama luego del else (si hay)
- Si el handler o el bloque else lanzan una excepción, esta se guarda temporalmente, se ejecuta el bloque finally y luego se relanza la excepción
- En caso de un break o continue, se ejecuta el bloque finally antes de resumir el bucle for/while

EXCEPTIONS - WITH

- La captura de excepciones puede hacer ilegible un programa
- Python permite construir objetos contexto que encapsulan un try except en un bloque

```
with File('test.txt') as f:
    do_something(f.read())
...
class File(object): # mock file implementation
    def __init__(self, path):
        #open the file
    def __enter__(self):
        return self
    def __exit__(self, type, value, trace):
        self.close()
        return True
```

- El statement with object as name tiene el siguiente protocolo:

```
1. se ejecuta el metodo object.__enter__
2. object.__enter__ devuelve un valor, as name
3. se ejecuta el bloque de código dentro del with
4. ante una excepción e, se llama a object.__exit__(type(e), e, sys.gettrace())
5. si finaliza el bloque with sin excep ejecuta object.__exit__(None, None, None)
```

MODULES AND PACKAGES

MODULES

- Un archivo .py que contiene código python se puede utilizar como un módulo
- Un módulo se utiliza mediante el keyword import

```
    #mymodule.py
    a = 10
    def foo():
        return False
    class Simple:
        pass
```

```
>>> import mymodule
>>> mymodule.a
    10
>>> mymodule.foo
<function foo at 0x8fb5f44>
```

- Import mymodule realiza básicamente 3 cosas:

```
- Crea un namespace nuevo, para ser utilizado por el archivo .py que se está
  importando
- Ejecuta el código del archivo.py, utilizando el nuevo namespace como scope global
- Se asocia el nombre mymodule con este namespace y sus atributos son accesibles
  desde la importación
```

SINTAXIS DE IMPORT

- Se pueden importar multiples modulos en un mismo import statement

```
import os                    # single import
import sys, subprocess, csv  # multiple imports
```

- Se puede especificar que nombre asociar al nuevo namespace, diferente al nombre del módulo

```
>>> import custom_socket as socket
>>> import sys as system, subprocess
>>> import sys
>>> sys is system
True
```

SINTAXIS DE IMPORT (2)

- Que diferencia hay entre correr e importar un archivo .py?
- Cuando se carga un módulo, la variable global `__name__` indica el nombre del módulo - Vale `'__main__'` cuando es ejecutado - Vale el nombre del módulo cuando es importado

```
#main.py
if __name__ == '__main__':
    print('ejecutando')
    call_my_init_function()
else:
    print('importando')
```

SINTAXIS DE IMPORT (3)

- Se puede importar un subconjunto específico de nombres definidos dentro de un módulo
- El símbolo * indica todos los nombres del módulo

```
#mymodule.py
a = 10
def foo():
    return False
class Simple:
    pass
```

```
>>> from mymodule import a, foo as some_function
>>> a
10
...
>>> some_function
<function foo at 0x8fb5f44>
...
>>> Simple
NameError: name 'Simple' is not defined
...
>>> from mymodule import *
>>> Simple
<class mymodule.Simple at 0xa27002c>
```

SEARCH PATH

- Para cargar un módulo, el intérprete debe encontrar el archivo a ejecutar
- Hay una lista de puntos de búsqueda en sys.path, que se examinan en orden para encontrar el archivo
- Generalmente el primer punto es un string vacío '', que indica el directorio actual
- Además de rutas absolutas y relativas a directorios, se pueden incluir archivos .zip y .egg
- Un archivo .zip se puede expresar como un directorio que contiene otros subdirectorios
- Un archivo .egg es basicamente un zip con metadata (version, dependencias, etc.)

TIPOS DE ARCHIVO A IMPORTAR

- import foo implica buscar en cada entrada de sys.path:
  1. Un directorio foo que denote un paquete
  2. Un archivo .pyd, .so o .dll (modulo de extensión compilado)
  3. Archivos ya traducidos a bytecode y optimizados (.pyo)
  4. Archivos ya traducidos a bytecode (.pyc)
  5. Archivos fuente .py (y en windows, también .pyw)

PACKAGES

- Los Paquetes permiten agrupar módulos en jerarquías de nombres
- Un package se define con un directorio que contiene un archivo llamado `__init__.py`
- Se pueden colocar módulos dentro de ese directorio, formarán parte del package

```
custom_math/
    __init__.py
    scalar.py
    vector/
        __init__.py
        matrix.py
```

```
>>> import custom_math
>>> custom_math.scalar
    AttributeError: 'module' object has no attribute 'scalar'
>>> import custom_math.scalar
    custom_math.scalar
    <module 'custom_math.scalar' from 'custom_math/scalar.py'>
```

`__INIT__.PY`

- Los packages son parte del protocolo de importación, pero se representan con módulos
- Que archivo fuente representa el paquete - modulo custom_path?
- Si es un módulo, debería referenciar un archivo custom_path.py
- Si es un package, `custom_path/__init__.py`
- `__init__.py` puede (y suele) estar vacío
- Los nombres definidos en `__init__.py` son inmediatamente accesibles como package.name

```
#custom_math/__init__.py
scalar = True
```

```
>>> import custom_math
>>> custom_math
<module 'custom_math' from 'custom_math/__init__.pyc'>
>>> custom_math.scalar
True
>>> import custom_math.scalar
>>> custom_math.scalar
<module 'custom_math.scalar' from 'custom_math/scalar.pyc'>
```

IMPORT * Y __ALL__

- Un módulo puede definir la variable `__all__` para controlar que atributos se exportan en from module import *
- En un paquete, `__all__` puede definirse en `__init__.py`
- `__init__.py` puede importar otros módulos

```
#modall.py
__all__ = ['a','b']
a=10
b=20
c=30
```

```
>>> from modall import *
>>> a
10
>>> b
20
>>> c
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

JERARQUÍAS DISTRIBUIDAS Y __PATH__

- Además de __name__, un package obtiene una variable __path__, similar a sys.path, pero exclusiva para el package
- __init__.py puede modificar __path__ para incluir otras ubicaciones de módulos para el mismo package

```
#custom_math/__init__.py
__path__.append('/some/other/path/another_math/')
```

```
#another_math/test.py
t=1
```

```
>>> import (custom_math.scalar, custom_math.test)
>>> custom_math.scalar
<module 'custom_math.scalar' from 'custom_math/scalar.pyc'>
>>> custom_math.test
<module 'custom_math.test' from '/some/other/path/another_math/test.pyc'>
```

IMPORTS ABSOLUTOS Y RELATIVOS

- Dentro de un package, como importar un submódulo?
- hasta python < 2.6, import some_mod dentro de un package busca primero módulos dentro del directorio del package
- desde python 2.6, los imports son absolutos por defecto, buscando siempre en sys.path
- un import relativo es explícito, utilizando la sintaxis from ... import ...
- Se pueden referenciar modulos dentro del paquete similar a un path relativo

```
#custom_math/__init__.py
import scalar #hasta version < 2.6.
#En 2.6+ puede colisionar con scalar global
from custom_math import scalar
#package path completo
from . import scalar
#relative import, python 2.6+
#en una estructura más compleja:
from ..math.custom import scalar_math as scalar
# .. referencia el paquete que contiene al actual
```

SYS.PATH, PYTHONPATH Y SITE MODULE

- Python busca módulos en la lista de fuentes sys.path
- La variable de entorno PYTHONPATH sobrecarga esta lista
- Al iniciar, el interprete importa un módulo llamado site, que generalmente aumenta sys.path para incluir modulos de terceros
- Se deshabilita con la opción -S del intérprete

DISTRIBUIR PROGRAMAS Y MÓDULOS PYTHON

- Python provee un módulo distutils para estandarizar la instalación de programas y paquetes
- Estructura instalable:

```
mymath/
    setup.py
    README.txt
    some_module.py
    some_script.py
    custom_math/
        __init__.py
        scalar.py
        vector/
            __init__.py
            matrix.py
```

```
#setup.py
from distutils.core import setup
setup(name='mymath', version='1.0', py_modules=['some_module'],
    packages=['custom_math'], scripts=['some_script.py'],
)
```

SETUP.PY

- Setup toma otros parametros opcionales, como 'author', 'author_email', 'url', 'description', etc
- setup.py acepta los siguientes subcomandos: *sdist: crea un zip o tar.gz en el subdirectorio sdist/ para distribuir (exluyendo archivos no referenciados del arbol original)* bdist: crea un archivo instalable en la plataforma elegida. soporta rpm, pkg, windows installer * python setup.py bdist --help-formats lista los formatos soportados
- hay otras implementaciones compatibles con distutils que soportan más opciones de instalación (development_mode,scripts automáticos desde módulos, etc)
- El más utilizado es setuptools o el sucesor distribute

INSTALACIÓN DE MÓDULOS DE TERCEROS

- La mayoría de las aplicaciones son instalables desde su fuente. python setup.py install
- Hay un repositorio no regulado en pypi.python.org (the Python Package Index)
- Utilidades como easy_install y pip instalan software y sus dependencias
- Utilizan pypi como fuente de paquetes por defecto

GRACIAS!!