Sofware Development with Scripting Languages: Python Crash Course

Onur Tolga Şehitoğlu

Computer Engineering,METU

24 February 2011

Python Syntax

- Indentation sensitive!!
- Code blocks are marked by indentation, not by explicit block markers (like { } in C)
- Each physical line indented with respect to previous one is assumed to be a new block
- Blocks are only valid with block definitions (functions, loops, conditionals)
- Backslash is used to join next line to current line continuation
- Bracket/paranthesis expressions explicitly introduce continuation

Values and Types

- `type(..)` returns the type of any expression
- Types
  - Primitive: `int float complex bool str unicode`
  - Composite: `tuple`, `list`, `dict`
  - User defined and library classes
- Literals
  - `123` , `231.23e-12` , `2.3+3.534j` , `True` , `'hello'` , `"world"` , `u"hosgeldiniz"`
  - `1231412412312L` (long), `0432` (octal), `0x43fe` (hex), `0b0100110101` (binary)
  - Composite: `('ali',123) ['134',2,5] {'ali':4, 'veli':5}` `lambda x:x*x`

Composite Types

- All types have their classes and class interface
- `help(classname)` is provided interactively
- Tuples: sequence of values separated by comma, enclosed in parenthesis. Immutable
- Lists: sequence of values (non-homogeneous), enclosed in brackets. Mutable
- Sets: no specific syntax `set()` constructor is used.
- Dicts: key-value pairs. key can be any hashable value (primitive)

Operators and delimiters

```
+       -       *       **      /       //      %
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=      <>      is
(       )       [       ]       {       }      @
=       +=      -=      *=      /=      //=     %=
&=      |=      ^=      >>=     <<=     **=
or              and     not     del
.       ;       :
```

- `is` operands use same exact memory area
- `//` flooring division (floating point)
- `**` power operator, right associative high precedence
- `del` delete an item from a data structure

Conditionals

```
if condition :
    statements
elif condition :
    statements
else:
    statements
```

- Indentation is required for blocks.
- `elif` and `else` parts are optional
- Conditional expression:

  exp1 `if` condition `else` exp2

While loop

```
while condition :
    loop body statements
else:
    termination statements
```

- Loop body executed as long as condition is true
- If condition is false `else` part is executed and loop terminated
- `else` part is optional

For loop

```
for var in iterable expression :
    loop body statements
else:
    termination statements
```

- Definite iteration over a data structure
- Iterable expression evaluated once. Then for each `next()` value body of the loop is executed. The variable is assigned to value from next.
- When all elements iterated, `else` part is executed and loop terminated
- `else` part is optional
- lists, tuples, strings, dictionaries,... are iteratable objects

Escapes

- `break` terminates the last enclosing loop without executing `else:` part if defined
- `continue` jumps to the beginning of next iteration (skips remaining part of the loop body)
- `try` statement is used to handle exceptions

Exceptions

```
try:
    statements
except exceptionvalue :
    handler statements
except excetionvalue2 as var:
    handler can refer to var for exception arguments
except :
    any exception handling
```

- exception values belong to `exception` class
- `raise` statement can be used to raise an exception
- If not handled exceptions stop execution
- `exception` class can be extended to define user-defined exceptions

Function definition

```
def functionname(parameterlist) :
    """ function documentation here
        continues....
    """
    statements, function body
    return function return value
```

- Parameters can have default values as `def f(x=0,y=0): ...`
- When calling parameters can be explicitly chosen as `f(y=2, x=4)`
- Parameter passing is pass by reference. The mutability of values are significant.
- Assignment semantics is followed for parameter passing

Class definition

```
class classname(optionalbaseclass) :
    """ class documentation here
        continues....
    """
    cx = 0          # class member
    def __init__(self):
        """ this is constructor """
        self.x = 0    # how to create/access member variables
        self.y = 0
        self.cx += 1      # class member, classname.cx += 1 is simil
    def increment(self):
        self.x += 1
    def _notprivate(self):
        pass             # no private members but methods starting wit
                         # _ are private by convention


x = classname()       # how to create an instance
x.increment()         # call member
classname.increment(x) # other way of calling it
print(classname.cx)     # class members can be accessed as well
```

- self is always the first parameter of the class method, it is passed as the firs parameter
- __init__ is the constructor name
- __str__ can be implemented to get string representing the object
- __unicode__ can be implemented to get unicode string representing the object
- __repr__ can be implemented to change how interpreter displays the object. `str()` calls `repr()` when not implemented
- __new__ is the class constructor (called before init)
- `isinstance(x, Myclass)` is instance check
- `issubclass(Cl1, Cl2)` is subclass check
- `super()` is the super class of the class

Operator Overloading

- Operator overloading achieved through special member functions:

```
x * y → x.__mult__(y)
x / y → x. __div__ (y)
x // y → x. __floordiv (y)
x > y → x. __gt__ (y)
x[y] → x. __getitem__ (y)
x[y]=z → x. __setitem__ (y,z)
del x[y] → x. __delitem__ (y)
x in y → x. __contains__ (y)
x += y → x. __iadd__ (y)
x.y → x. __getattr__ (y)
x.y = z → x. __setattr__ (y,z)
del x.y → x. __delattr__ (y)
```

Variable Scope and Lifetime

- Variables are local to enclosing block
- Global variables have read only access unless they are used as l-value in the block
- If variable used as an l-value a local variable is created and all read-only accesses preceding it gives an error
- `global` keyword is used to make a global variable available in a local block (read and update)

Assignment Semantics

- Share semantics
- Assignment copies reference, not data
- Object assignment creates two variables denoting same object
- Primitive values copied, objects shared (like Java)
- Parameters pass by value for primitives, reference for objects
- Constructors needed for copying `list([1,2,3])`

Iterators

- Iterators are used to iterate on data structures or create sequences
- `__iter__` method returns the iterator object
- `next` method gives the next object for the iterator
- `StopIteration` exception is rased on `next` to end iteration

```
for i in a:
    # loop body
```

is equivalent to:

```
it=iter(a)
try:
    i=it.next()
    while True:
        # loop body
        i=it.next()
except StopIteration:
    pass
```

```
class Fibonacci:
   def __init__(self,n):
        self.a = 0
        self.b = 1
        self.count = 0
        self.n = n
   def __iter__(self):
        return self
   def next(self):
        self.count += 1
        if self.count >= self.n:
                raise StopIteration
        self.a, self.b = self.b , self.a + self.b
        return self.a
```

- In iterators the state has to be kept in iterator object
- Consider a single instance of Fibonacci iterated on a nested loop!
- A correct implementation has to create a new instance for each `iter()` call
- Hard to write iterators on objects with `next()` value is not trivial
- Generators automatically keep state of computation and continue where it left.
- Use of `yield` keyword is sufficient to write a generator
- Each `yield` corressponds to a `next()`
- Generator functions only `return` without parameter to mark end of computation

```python
def fibonacci(n):
        a = 0
        b = 1
        count = 0
        while n>count:
                yield  b
                a,b = b,a+b
                count += 1
```

Python creates all required intermediate objects and methods.

A Tree Example

```
class BSTree:
        ''' A binary search tree example'''
        def __init__(self):
                self.node = None
        def set(self, key, val):
                if self.node == None:   # empty tree
                        self.node = (key, val)  # node content is a tuple
                        self.left, self.right = BSTree(), BSTree()
                elif key < self.node[0]:        # not empty test key
                        self.left.set(key, val)         # insert on left subtree
                elif key > self.node[0]:
                        self.right.set(key, val)        # insert on right subtree
                else:
                        self.node = (key, val)          # update
        def get(self, key):
                if self.node == None:   # empty tree
                        raise KeyError  # list, tuple also raise this
                elif key < self.node[0]:
                        self.left.get(key)
                elif key < self.node[0]:
                        self.right.get(key)
                else:
                        return self.node[1]     # found, return value
        def __str__(self):
                if self.node == None:   return "*"
                else:   return "[" + str(self.left) + str(self.node) +\
                                str(self.right) + "]"

a = BSTree()
for i in [6, 2, 8, 3, 9, 1]:
```

An iterator on Tree Example

```
# ... added to BST
        def _nextof(self, key):          # return min value >key
                if self.node == None:    return None
                elif key == None or key < self.node[0]:
                        v = self.left._nextof(key)
                        return self.node if v == None else v
                else:
                        return self.right._nextof(key)

        def __iter__(self):
                return BSTree.BSTreeIter(self)  # new instance of nested class

        class BSTreeIter:
                def __init__(self, tree):
                        self.tree = tree
                        self.state = None
                def next(self):
                        nextnode = self.tree._nextof(self.state)
                        if nextnode == None:
                                raise StopIteration
                        else:
                                self.state = nextnode[0]
                                return nextnode
# main
a = BSTree()
#... insert values etc.
for (k,v) in a:
        print(k,v)
```

A generator on Tree Example

```
# ... added to BSTtree
        def traverse(self):
                if self.node == None:
                        raise StopIteration
                else:
                        for (k,v) in self.left.traverse():
                                yield (k,v)

                        yield self.node

                        for (k,v) in self.right.traverse():
                                yield (k,v)

# main
a = BSTree()
#... insert values etc.
for (k,v) in a.traverse():
        print(k,v)
```

String Processing

- `str()` class implements methods to process strings
- `string.split(delimeter)` is used to convert string to an array of strings delimeted by delimeters.
- `string.join(array)` is used to convert a array of strings to a concatanated string, seperated by the string object
- `map(function, sequence)` is used to apply function to all members of the sequence and return a list of return values
- `string.join(map(str, array))` will join string representation of all array types
- + concatanates to strings. All lexicographic comparisons are implemented as usual operators.
- `string.index(substr)` searches and returns position of substring in the string. `find()`: same but returns -1 instead of exception.