

# Software Development with Scripting Languages: Python Crash Course

Onur Tolga Şehitoğlu

Computer Engineering, METU

24 February 2011

- 1 Python Syntax
- 2 Values and Types
- 3 Composite Types
- 4 Operators
- 5 Conditional
- 6 While loop
- 7 For loop
- 8 Escapes
- 9 Exceptions

- 10 Function definition
- 11 Class definition
- 12 Class Members
- 13 Operator Overloading
- 14 Variable Scope and Lifetime
- 15 Assignment Semantics
- 16 Iterators
- 17 Generators
- 18 String Processing
- 19 Decorators

# 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/parenthesis expressions explicitly introduce continuation

# Values and Types

- `type(..)` returns the type of any expression
- Types
  - Primitive: `int` `float` `complex` `bool` `str` `bytes`
  - Composite: `tuple`, `list`, `set`, `dict`
  - User defined and library classes
- Literals
  - `123` , `231.23e-12` , `2.3+3.534j` , `True` , `'hello'` , `"world"` , `b'ho\x05\x09fgeldin'`
  - `0o432` (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. Usually heterogeneous (each value may have a different data type)
- `help(classname)` is provided interactively
- **Tuples**: sequence of values separated by comma, enclosed in parenthesis. Immutable
- **Lists**: sequence of values, enclosed in brackets. Mutable
- **Sets**: sequence of values separated by comma, enclosed in curly braces.
- **Dicts**: key-value pairs. key can be any hashable value (primitive)

# Operators and delimiters

+	-	*	**	/	//	%
<<	>>	&		^	~	
<	>	<=	>=	==	!=	is
[]	.	@				
=	+=	-=	*=	/=	//=	%=
&=	=	^=	>>=	<<=	**=	
or		and	not	del	in	

- `is` operands use same exact memory area
- `/` floating point division (flooring for integers in Python 2)
- `//` flooring division
- `**` 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
- `else` part is optional
- `else` part is executed after loop terminates even if loop body is not executed at all.
- When loop terminates without a condition test, i.e. by a `break`, `else` part is not executed.



# 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 terminates
- `else` part is optional. Executed when loop terminates without `break` or exception.
- lists, tuples, strings, dictionaries,... are iterable 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
        classname.cx += 1      # class member update,
    def increment(self):
        self.x += 1
    def _notprivate(self):
        pass              # no private members but methods starting with
                          # _ 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

```

# Class Members

```
class T:
    x          # class member
    def __init__(self):
        self.v = x      #!!!! Invalid
        self.v = self.x  # Valid as r-value, readonly access
        self.x = self.x + 1 # LHS instance member, LHS instance
        T.x += 1          # correct usage

a=T()
print(a.x)      # r-value, OK
a.x = 10        # l-value, creates an instance member
T.x = 10        # class member
```

- Class members work in class scope, not in object.
- Instances can access them as r-value, not l-value.
- Scope should be given explicitly, otherwise considered local variable.
- `classname.membername` is the correct way for l-value access.

- No private/public/protected in Python. All values are accessible.
- Leading `_` avoids importing from a module as `from module import *`. Some frameworks libraries hide them with internal mechanisms.
- Leading `__` mangles name of method as `_ClassName__method` when called outside. Other methods can call it with original name (with leading underscores). It is also used to avoid overriding due to inheritance.
- `@classmethod` decorator can be used to create methods getting class methods getting class as the parameter
- `@staticmethod` decorator can be used to create methods in class scope without any object/class parameter constraints.

- `self` is always the first parameter of the class method, it is passed as the first parameter
- `__init__` is the constructor name
- `__str__` can be implemented to get 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(C1, C2)` is subclass check
- `super()` is the super class of the class
- `super().__init__()` calls the super class constructor. Not implicit.
- `superclassname.__init__(self, ...)` can be used as well.



# Operator Overloading

- Operator overloading achieved through special member functions:

`x * y`  $\rightarrow$  `x.__mult__(y)`

`x / y`  $\rightarrow$  `x.__div__(y)`

`x // y`  $\rightarrow$  `x.__floordiv__(y)`

`x > y`  $\rightarrow$  `x.__gt__(y)`

`x[y]`  $\rightarrow$  `x.__getitem__(y)`

`x[y]=z`  $\rightarrow$  `x.__setitem__(y,z)`

`del x[y]`  $\rightarrow$  `x.__delitem__(y)`

`x in y`  $\rightarrow$  `x.__contains__(y)`

`x += y`  $\rightarrow$  `x.__iadd__(y)`

`x.y`  $\rightarrow$  `x.__getattr__(y)`

`x.y = z`  $\rightarrow$  `x.__setattr__(y,z)`

`del x.y`  $\rightarrow$  `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 raised on `next` to end iteration

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

*is equivalent to:*

```
it = iter(a)  
try:  
    while True:  
        i = next(it)  
        # loop body  
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` corresponds to a `next()`
- Generator functions only `return` without parameter to mark end of computation

```
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 and return a [generator](#) object.



# A Tree Example

```

class BSTree:
    ''' A binary search tree example'''
    def __init__(self):
        self.node = None
    def __setitem__(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[key] = val # insert on left subtree
        elif key > self.node[0]:
            self.right[key] = val # insert on right subtree
        else: self.node = (key, val) # update
    def __getitem__(self, key):
        if self.node == None: # empty tree
            raise KeyError # list, tuple also raise this
        elif key < self.node[0]:
            return self.left[key]
        elif key > self.node[0]:
            return self.right[key]
        else: return self.node[1] # found, return the 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]:
    a[i] = i*i
print(a)

```

# 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 (delimiter)` is used to convert string to an array of strings delimited by delimiters.
- `string . join (array)` is used to convert a array of strings to a concatenated string, separated 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
- `+` concatenates 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.

# Decorators

- Python decorators are evaluated during function/class definition and maps function/class definition into a new one
- Decorators are functions returning a callable that replaces qactual function or class constructor.

```
def dec(callable):
    def f():
        ..... use callable here
    return f
```

```
# ---
```

```
@dec
```

```
class cls(): class defn here
```

```
# equivalent to
```

```
class cls(): class defn here
```

```
...
```

```
cls = dec( cls )
```

```
# ---
```

```
obj = cls()           # actually calls f() inside dec()
```

- Decorators are used to alter functionalities of classes and functions for routine tasks.
- A simple trace decorator:

```
def trace(f,*p, **kw):  
    def func(*p, **kw):  
        print('entered', p, kw)  
        r = f(*p, **kw)  
        print('exitted', p, kw, r)  
        return r  
    return func  
  
@trace  
def f(x):  
    return x+1  
  
@trace  
def g(a,b):  
    if a < b: (a,b) = (b,a)  
    return a if b == 0 else g(b, a % b)  
  
print(f(3))  
print(g(210,63))
```

■ Example: an instance counting decorator:

```
icount = {}  
def counted(cls,*p,**kw):  
    def func(*p, **kw):  
        icount[cls] = icount[cls]+1 if cls in icount else 1  
        return cls(*p, **kw)  
    return func  
  
@counted  
class A:  
    def __init__(self,a):  
        self.v = a  
  
a = A(3)
```

# Builtin Decorators

There are couple of useful decorators:

- **@classmethod** makes a method expect a class argument as first argument. Method works in class scope instead of instance scope
- **@staticmethod** makes a method independent from object. It works as a normal function without object argument. It becomes a function in the class scope. Useful for auxiliary functions inside class definition.

```
class R:
    sumx, sumy = 0, 0                # class members: R.sumx,...
    def __init__(self, x,y):
        t = self._f(x,y)            # calls the staticmethod
        self.a, self.b = x/t, y/t
        R.sumx, R.sumy = R.sumx+x, R.sumy+y
    @classmethod
    def sums(cls):                   # First argument is the class, not instance
        return cls.sumx, cls.sumy
    @staticmethod
    def _f(a,b):                     # No extra argument
        if a < b: a,b = b,a
        return a if b == 0 else R._f(b, a % b) # recursive call

a = R(4,6)
print(R.sums(), a.sums())
print(a._f(72,48), R._f(72,48))
```



- `@property` decorator. Defines a member value that is calculated with a function.

```
class P:
    def __init__(self, name, surname):
        self.name, self.sname = name, surname
    @property
    def fullname(self):
        return '␣'.join( (self.name , self.sname))

p = P('Bugs', 'Bunny')
print(p.fullname)
```

- Also setter and deleter functions can be defined for setting the member and deleting it:

```
#... continued ...
@fullname.setter
def fullname(self, value): # gets the RHS as argument
    self.name, self.sname = value.split('␣')
    @fullname.deleter
    def fullname(self): # removes the value
        self.name, self.sname = '', ''

p.fullname = 'Duffy␣Duck'
print(p.fullname)
del p.fullname
print(p.fullname)
\item Functional control over members can be implemented. (Invalid value assignment)
```