Strings, Lists, Functions, Classes

HANDS-ON INTRODUCTION TO PYTHON

Lab

- Commencing this week
- Time: T3-4
- Venue: SHB123

- Read (best try the examples in notes) before attending the lab
- Submit your programs in an archive with your student ID to the Blackboard account (at corresponding lab slot).

Strings

- Enclosed either by single quote ‘ or double quote “

```
>>> 'doesn\'t‘
 "doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," \nhe said.'
"Yes," \nhe said.‘
>>> print '"Yes," \nhe said.’
"Yes,"
he said.
>>> "\"Yes,\" he said.“
 '"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Strings

- Concatenated or repeated with + and * respectively

```
>>> word = '1040' + 'super‘
>>> word
'1040super‘
>>> '['+word*3 + ']‘
'[1040super1040super1040super]'
```

Strings

- Access can be in subscripted notation as in C

```
>>> word[2]
'4`
```

- substrings can be specified with the *slice notation*

```
>>> word[0:2]
'10`
>>> word[:2]
'10`
>>> word[1:]
'040super`
```

- For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds

Strings

- Immutable

```
>>> word[0] = 2
Traceback (most recent call last):
File "<interactive input>", line 1, in
  <module>
TypeError: 'str' object does not support
  item assignment
```

- But can create a new string

```
>>> 'csci'+word[0:4]
'csci1040'
```

String functions

Operation	Description
s.capitalize()	capitalize the first character of s
s.capwords()	capitalize the first letter of each word in s
s.count(sub)	count number of occurrence of sub in s
s.find(sub)	find first index of sub in s, or -1 if not found
s.index(sub)	find first index of sub in s, or raise a ValueError if not found
s.rfind(sub)	find last index of sub in s, or -1 if not found
s.rindex(sub)	find last index of sub in s, or raise a ValueError if not found
s.lower()	convert s to lowercase
s.split()	return a list of words in s
s.join(lst)	join a list of words into a single string with s as separator
s.strip()	strips leading/trailing white space from s
s.upper()	convert s to upper case
s.replace(old,new)	replace all instances of old with new in string

Functions

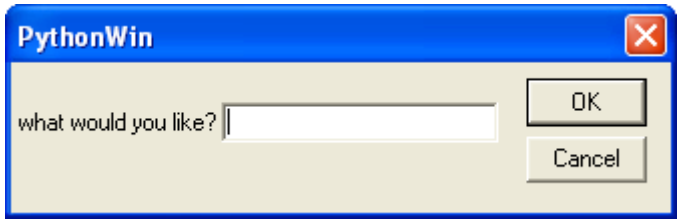
- Define a block of statements with a name, e.g.

```
>>>def areaOfRect (w, h):
      return w*h

>>> print 'area of rect 2 by 3.2 is' , areaOfRect(2, 3.2)
area of rectangle 2 by 3.2 is 6.4

>>> print 'silly in silly out ' , areaOfRect('abc', 3.2)
silly in silly outTraceback (most recent call last):
File "<interactive input>", line 1, in <module>
File "<interactive input>", line 2, in areaOfRectTypeError:
   can't multiply sequence by non-int of type 'float'
```

Default argument values

- specify a default value for one or more arguments
- function that can be called with fewer arguments than it is defined to allow

```
>>> def ask_lunch(prompt, set = 'A'):
...     choice = raw_input(prompt)
...     if choice == '':
...             return set
...     else:
...             return choice
...
>>> ask_lunch("what would you like?")
'A'
```



Name Scope

- Names defined outside functions have global scope
- Any local names will shadow the global (same name)
- All values & names destroyed after return

```
>>> x=4
>>> def scopetest(a):
...     return x + a
...
>>> print scopetest(3)
 7
>>>
```

```
x=4
>>> def scopeTest(a):
...     x=7
...     return x + a
...
>>> print scopeTest(3)
10
```

Using globals

- To have assignment access on global variables, use global statement

```
>>> def scopetest (a):
...     global b
...     b = 4
...     print 'inside func, b is ', b
...
>>> a = 1
>>> b = 2
>>> scopetest(a)
inside func, b is  4
>>> print 'after func, b is ', b
after func, b is  4
```

Raise exception

- Halt execution when some values are not valid

```
>>> def compoundYear( balance, rate, numYears):
...     if rate < 0:
...             raise RuntimeError('-ve interest rate')
...     if numYears < 0:
...             raise RuntimeError('-ve number of years')
...     for year in range(9, numYears):
...             balance = compound(balance, rate)
...     return balance
... >>> print 'after 10 yrs,', compoundYear(1000, -5, 3)
after 10 yrs,Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "<interactive input>", line 3, in compoundYear
RuntimeError: -ve interest rate
```

Functions ⇔ values

- Function can be assigned to variable
- That variable can be used as function again

```
>>> def fun(a):
...   print 'argument is ', a
...
>>> print fun
  <function fun at 0x0125AA70>
>>> x = fun
>>> x('hi')
argument is  hi
>>>
```

None

- Name ‘None’ is used to designate nothing (NULL in C)

```
>>> def Nothing():
...   print "sorry nothing here"
...
>>> a = Nothing()
sorry nothing here
>>> print a
None
```

Tuples

- Similar to list, but form using parenthesis
- Immutable likes strings
- Non-mutable operations in list can apply to tuple

```
>>> tup = (1,7,3,1,7,3)
>>> 3 in tup
True
>>> list(tup)
[1, 7, 3, 1, 7, 3]
>>> tuple(['a', 'b', 'c'])
('a', 'b', 'c')
```

Tuples

- Comma operator implicitly creates a tuple

```
>>> 'a','b','c`
('a', 'b', 'c')
```

- Application in function returning more than 1 result

```
def minAndMax( info ):
    return (min(info), max(info))
>>> x, y = minAndMax( 'abcd')
>>> x
'a`
>>> y
'd'
```

String Formatting

- String formatting operator %
- Can do formatting function similar to that in C/C++

```
>>> 'int %d float %g and string %s' % (17, 3.1416, 'abc')
'int 17 float 3.1416 and string abc'
```

Lists

- Elements in a list can be changed

```
>>> lst = [1, 3, 5, 7, 9]
>>> lst[1] = 4
>>> lst
[1, 4, 5, 7, 9]
```

- Slice (part of list) can be extracted and be target of assignment

```
>>> lst[1:3]
[4, 5]
>>> lst[1:3] = [9, 8, 7, 6]
>>> lst
[1, 9, 8, 7, 6, 7, 9]
```

List Operations

Operation	Description
s.append(x)	appends element x to s
s.extend(ls)	appends list s with ls
s.count(x)	count number of occurrence of x in s
s.index(x)	return index of first occurrence of x
s.pop()	return and remove last element from s
s.pop(i)	return and remove element i from s
s.remove(x)	search for x and remove it from s
s.reverse()	reverse element of s in place
s.sort()	sort elements of s into ascending order
s.insert(i, x)	inserts x at location i

Classes

- encapsulate several related functions/data into a single unit
- Functions are thus called methods

```
class BankAccount(object):
    def __init__(self):
        self.balance = 0
    def deposit (self, amount):
        self.balance = self.balance + amount
    def withdraw (self, amount):
        self.balance = self.balance - amount
    def getBalance(self):
        return self.balance
```

Classes

- Can have more than one argument for constructor

```
class BankAccount(object):
    def __init__(self, initBalance):
        self.balance = initBalance
    def deposit (self, amount):
        self.balance = self.balance + amount
    def withdraw (self, amount):
        self.balance = self.balance - amount
    def getBalance(self):
        return self.balance

myAccount = BankAccount ( 200)
```

Classes

- receiver of the method take first argument implicitly
- Allow one to distinguish between different instance of same class

```
class BankAccount(object):
    def __init__(self):
        self.balance = 0
    def deposit (self, amount):
        self.balance = self.balance + amount
    def withdraw (self, amount):
        self.balance = self.balance - amount
    def getBalance(self):
        return self.balance
myAccount = BankAccount()
print myAccount.getBalance()
>>>0
myAccount.deposit(100)
print myAccount.getBalance()
>>>100
```

Field of class, qualified by class name

Classes

- Internal values of a class is accessible through '__'

```
>>> print BankAccount.__name__
BankAccount
>>> print myAccount.__class__
<class '__main__.BankAccount'>
```

- **Instance dictionary**

```
>>> print myAccount.__dict__
{'balance': 300}
```

Class Variables

- Possible to include assignment statement in class definition
- These variables are shared among all instance of class
- only one copy

```
class BankAccount(object):
    accountType = 'bank account'
    def __init__(self, initBalance):
:
>>> print BankAccount.accountType
bank account
```

Class boundaries

- Fields can be accessed directly

```
>>> print myAccount.balance
300
```

- Can be altered from outside object

```
>>> myAccount.balance = 500
>>> print myAccount.balance
500
```

- Everything in Python is **public**!
- Should respect the object boundaires i.e. not to exploit the fact at the expense of code complexity

Calling methods

- To invoke an object method, you must name explicitly the object it refers

```
def transfer(self, amount, toAccount):
        self.withdraw(amount)
        toAccount.deposit(amount)
```

- In Java/C, self is not needed

Exceptions & try

- To impose further protection on object

```
def withdraw (self, amount):
        if self.balance < amount:
            raise ValueError, 'insufficient funds'
        self.balance = self.balance - amount
```

- Program terminate when insufficient fund in account
- Exceptions can also be caught

```
try:
    newAccount.withdraw(1000)
    print 'wow, free money'
except ValueError, e:
    print 'no such luck'
    print 'error message is ', e
no such luck
error message is  insufficient funds
```

Objects are references

- Objects are internally stored as references
- Thus assigning an object only means its reference being copied

```
husbandAccount = BankAccount( 500 )
wifeAccount = husbandAccount
wifeAccount.withdraw( 500 )
print husbandAccount.balance
>>>0
```

Printing of Object

- __str__ method can be used to return content when print object command is invoked

```
def __str__(self):
  return 'Account Object, Balance = %f ' %(self.balance)
```

```
>>>print myAccount

Account Object, Balance = 200.000000
```