Python – a modern scripting PL

- Basic statements
- Basic data types & their operations
  - Strings, lists, tuples, dictionaries, files
- Functions
- Strings
- Dictionaries

Many examples originally from O' Reilly "Learning Python", 2nd Ed, 2003 or online Python tutorial at python.org

Python

- Used for flexibility in prototyping
  - Type by use (dynamic typing)
  - Uses a kind of static scoping
  - Succinct syntax: indentation shows extent of syntax constructs
  - Supports both object-oriented and functional paradigms
  - Class-based inheritance (can be multiple)
  - Framework libraries provide functionality for many domains
  - Useful built-in data structures with shared pre-defined functions
  - Reference semantics with mutable and immutable objects
  - Iterators and generators on collections

Basics -1

- **Assignments**

  `X='spam'  x,y='one','two'  [a,b]=[1,2]`

- **Variables**

  `_x2, a2,` case-sensitive, can't clash with reserved words (e.g., lambda, for, not, or, if)

- **If statements**

```
x="Barbara"
if x=='Barbara':
   print ('found Barbara')
elif x=='Mary':
   print ('found Mary')
else: print ('missing key')
```

Prints: found Barbara

```
if x==1:
   a=2
   if y==2:
     print ("y is 2")
   print ("x is 1")
print ("done")
```

y is 2
x is 1
done

Basics - 2

- **Boolean expressions evaluated by short-circuit**

  `or, and, not, true, false, ==, !=`

- **Looping constructs: while**

```
y=25
x = y//2
while x>1: #matches else below;shown by indenting
  if y%x==0:
    print (y, 'has factor', x)
    break  #skips the else
  x = x-1
else:    #is executed even if loop body isn't
  print (y, 'is prime')
```

Prints: 25 has factor 5

Basics -3

- Iterator for lists & strings

```
for x in ["spam", "eggs", "ham"]:
   print (x) #iterates over list elements
s = "you"
for y in s: print (y) #iterates over chars
```

- Comments delimited by # anywhere on a line
- Prints:

```
spam
eggs
ham
y
o
u
```


Basics - 4

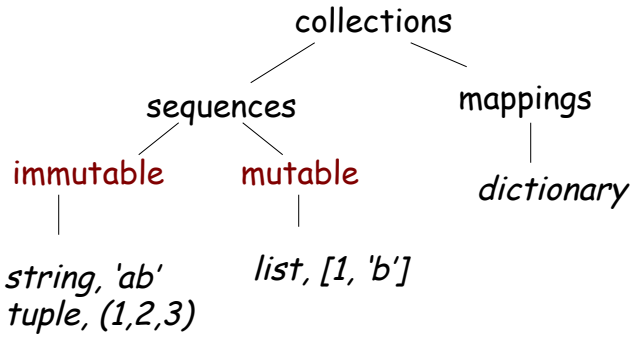
- Defining functions with `def` or `lambda`
  - *Python style: code a function to an object interface*
    - Idea is that a function should work on any datatype that supports the operations it needs
      - E.g., can use 'in' for any datatype admitting sequencing including lists, dictionaries, tuples, strings
      - Real polymorphism
- Two forms of function definition w def,lambda

```
def incr(x): return x+1 #function incr
#list of 2 functions
incrs = [lambda x: x+1, lambda x: x+2]
print (incrs[0])
print (incrs[0] (2))
Print (incrs[1] (6))
```

Prints:

```
<function <lambda> at 0x11a7dfe18>
3
8
```

Built-in Data Structures Type Hierarchy



Built-in data types

- **Lists – [1,2,3] - mutable**

```
print ([1,2.5,'a'])              # [1, 2.5, 'a']
print (len([1,[2,3],[4,5]]))     #3
xx = [1,2,3.4]
print (xx, xx[1:3])              #[1, 2, 3.4] [2, 3.4]
yy=['b','a']
print (xx+yy)                    #[1,2,3.4, 'b', 'a']
print (xx * 3)   #[1, 2 ,3.4, 1, 2, 3.4, 1, 2, 3.4]
```

- **Tuples – (1,2,3) - immutable**

```
print ((1,2.5,'b'))                  #(1, 2.4, 'b')
print (len( (1,2.5,'b') ))           #3
print ((1,2) * 2)                    #(1, 2, 1, 2)
print ((1,2,3,4,5) [1:3]) #slice     #(2, 3)
print((1,2,3) + (4,5,6)) #concat     #(1,2,3,4,5,6)
```

Strings

- Immutable sequences
- Literals – 'a' "bcd" "isn't" have built-in operations *repeat, concat, length, membership*

```
s1 = 'super'
s2 = "spam"
s3 = s1 + s2
s4 = s2 * 2
print (s2)              #spam
print (s3)              #superspam
print (s4)              #spamspam
print (s4[2:4])         #am
print (len(s4))         #8
print (s4.find('m'))    #3
print ('a' in s4)       #True
```


Strings-2

- String parsing operators : slice, index

```
s="spam"
print(s)        #spam
print(s[0])     #s
print( s[-1])   #m
print(s[1:3])   #pa
print(s[1:])    #pam
print(s[:-1])   #spa
```

  - First character at offset 0;
  - Negative offsets count from end of string
  - : with out a L or R value, goes to the end of the string in that direction

Strings -3

- Functions on string class *str*

```
s="Spam Is Bad"
print(s.upper())                #SPAM IS BAD
print(s.lower())                #spam is bad
print(s.replace('a','z'))       #Spzm Is Bzd
print(s.split())                #['Spam', 'Is', 'Bad'] – best way
print(s.split("I"))             #['Spam ', 's Bad']
print(s.split("a",2))           #['Sp', 'm Is B', 'd']
print(s.startswith("B"))        #False
```


Dictionaries

- Unordered heterogeneous collections of key,value pairs
  - Associative array (or map)
  - Objects in dictionary are unordered, iterators provided to allow access to all objects

```
dd={"Barbara": "professor", "Jon" : "systems engineer", "Andrew" : "developer"}
print (dd)
#{'Barbara': 'professor', 'Andrew': 'developer', 'Jon': 'systems engineer'}
print("Barbara: " + dd["Barbara"] +"\nJon: " + dd["Jon"])
#Barbara: professor
#Jon: systems engineer
for aa,bb in dd.items():        #Barbara professor
    print (aa,bb)               #Andrew developer
                                #Jon systems engineer
```

Dictionaries - 2

```
print (dd.keys())
 #dict_keys(['Barbara', 'Andrew', 'Jon'])
print ("karl" in dd.keys())
 #False
print("Barbara" in dd.keys())
 #True
print (dd.values())
 #dict_values(['professor', 'developer', 'systems engineer'])
print (dd.items())
 #dict_items([('Barbara', 'professor'), ('Andrew', 'developer'),
('Jon', 'systems engineer')])
ee={"Barbara":"professor", 1:[1]}
print (ee)   #{'Barbara': 'professor', 1: [1]}
print (ee.get("Barbara")) #professor
```


Dictionaries -3

```
ee["Beth"] ="surgeon"  #add a new element
print(ee)
  #{'Barbara': 'professor', 'Beth': 'surgeon', 1: [1]}
ee["Barbara"] = "former dept head "
print(ee)
   #{'Barbara': 'former dept head', 'Beth': 'surgeon', 1: [1]}
ee['barbara'] = 'avid reader'
print(ee)
  #{'Barbara': 'former dept head', 'Beth': 'surgeon', 'barbara':
'avid reader', 1: [1]}
```

Functions

- **Power of polymorphism**

```
#what datatypes can this function be applied to?
def intersect(seq1, seq2):
  res = [ ]
  for x in seq1:   #iterates over elements in seq1
    if x in seq2:  #checks if element is in seq2
      res.append(x) #if so, adds element to list
  return res

print (intersect([1,2,3][2,4,6]))       #[2]
print(intersect( (1,2,3,4), (4,5,6)))   #[4]
print(intersect ( (1,2,3), [1,2,3] ))   #[1,2,3]
print (intersect( {1:'a', 2:'b',3:'c'}, {1:'a',4:'d'} )) #[1]
print(intersect( {1:'a',2:'b'},{1:'c',2:'d'} )) #[1,2]
#clearly the intersection is on the keys of the dictionary,
   not the values!
```


List Comprehensions

- **Collect the results of applying an arbitrary expression to a sequence of values and returns them in a list**

```
print (range (10))
   range(0, 10) #object can generate numbers from 0 to 10 inclusive
print( ([x**2 for x in range(10)]))
   [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print([x**2 for x in range(10) if x%2==0])
   [0, 4, 16, 36, 64] #filters out odd numbers
print([x+y for x in [0,1,2] for y in [100,200,300]])
   [100, 200, 300, 101, 201, 301, 102, 202, 302]
```

Parameter Passing

- Pass reference to an object
  - If pass a reference to a mutable object, then callee may change value seen in caller on return
  - Immutable objects used as arguments cannot be changed; a new object is created

```
def change(x, y):
    x = 2 #local change
    y[0] = 'spam' #shared object change
z = (1, 2)        #immutable tuple
w = [1,2,3]       #mutable list
print ("before call", z,w) #before call (1,2) [1,2,3]
change(z,w)
print ("after call", z,w) #after call (1,2) ["spam",2,3]
```


Avoid Common Beginner Errors

- Don't forget colons (:)
- Start in column 1 and indent consistently
- Use simple for loops instead of while
- Don't expect return values from functions that mutate objects; they return *None*
- Always use parentheses to call a fcn, especially print() (diff than earlier Python version)