Introduction to Computing

MCS1101B

Lecture 2: Python

Sets : A Collection type

- Unordered list of unique immutable elements
- An empty set can be created using a call to `set()`
- A new set can ne created from any sequence types by the function `set()`

```
>>> a = set() --> set() /null set
>>> b = set([10,10,20, 40, 60]) --> {10,20, 40,60}
>>> c = set(('1','2','a')) --> {'1', '2', 'a'}
>>> d = set('abracadabra')--> ???
```

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
```

note that duplicates have been removed

```
>>> print(basket)
{'orange', 'banana', 'pear', 'apple'}
```

fast membership testing

```
>>> 'orange' in basket
True

>>> 'crabgrass' in basket
False
```

Sets (contd.)

```
>>> a = set('abracadabra')
>>> b = set('alacazam')

>>> a  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> b  # unique letters in b
{'a', 'l', 'c', 'z', 'm'}
```

```
>>> a - b # in a but not in b
{'r', 'd', 'b'}

>>> a | b  # in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}

>>> a & b   # in both a and b
{'a', 'c'}

>>> a ^ b #in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Elements are added as:

```
>>> a.add('z')
>>> a
{'a', 'r', 'b', 'c', 'd', 'z'}
```

Elements are removed as:

```
>>> a.remove('r')
>>> a
{'a', 'b', 'c', 'd', 'z'}
```

Dictionaries: A Mapping type

- Dictionaries store a mapping between a set of keys and a set of values
  - Keys can be any immutable type *(why only immutable?)*
  - Values can be any type
  - A single dictionary can store values of different types
- You can *define*, *modify*, *view*, *lookup* or *delete* the *key-value pairs* in the dictionary

Python's dictionaries are also known as hash tables and associative arrays

Creating & Accessing Dictionaries

Create an empty dictionary

```
>>> d1 = {}
```

Create a dictionary with initial values

```
>>>d={'user':'bozo', 'pswd':1234}
```

```
>>> d['user']
'bozo'
>>> d['pswd']
123
>>> d['bozo']
Traceback (innermost last):
  File '<interactive
   input>' line 1, in ?
KeyError: bozo
```

Updating & Removing in Dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}
>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}
```

Keys must be unique

Assigning to an existing key replaces its value

```
>>> d['id'] = 45
>>> d
{'user':'clown','id':45, 'pswd':1234}
```

Dictionaries are unordered

New entries can appear anywhere in output

Dictionaries work by hashing

Remove the entry for 'user'

```
>>> del d['user']
>>> d
{'p':1234, 'i':34}
```

Remove all entries in the dictionary

```
>>> d.clear()
```

Side note: del works on lists, too

```
>>> a=[1,2,3]
>>> del a[1]
>>> a
[1,3]
```

Useful Accessor Methods

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> d.keys()   # List of keys, VERY useful
['user', 'p', 'i']

>>> d.values() # List of values
['bozo', 1234, 34]

>>> d.items()  # List of item tuples
[('user','bozo'), ('p',1234), ('i',34)]
```

Defining and Calling Functions in Python

The syntax for a function definition is:

```
>>> def myfun(x, y):
        return x *
y  >>> def myfun2():
        print("hello")
```

The syntax for the function (defined above) call is:

```
>>> myfun(3, 4)
   12
>>> myfun2()
  'hello'
```

- Functions in python are defined using the keyword `def`
- You can give any name to the function and it must always be unique within your program
- The parameters are optional (or as per your requirements)
- Parameter types are automatically assigned based on the values passed during function calls
- Returns from a fucntion is optional too

Functions without Returns

All functions in Python have a return value,
Even if no return line inside the code

Functions without a return line, returns the *special value* None

- None is a special constant in the language
- None is used like NULL or void in C language
- None is also *logically equivalent* to False

The interpreter doesn't print None

Default Values for Arguments in Functions

You can provide default values for a function's arguments

These arguments are optional when the function is called

```
>>> def myfun(b, c=3, d="hello"):
        return b + c
>>> myfun(5,3,"hello")  # returns 8
>>> myfun(5,3)          # returns 8
>>> myfun(5)            # returns 8
>>> myfun(5,4)          # returns 9
>>> myfun(5,"Hi")       # returns error
>>> myfun(5,d="Hi")       # returns 8
```

Keyword Arguments in Functions

You can call a function with some or all of its arguments out of order as long as you specify their names

You can also just use keywords for a final subset of the arguments.

```
>>> def myfun(a, b, c):
        return a-b
>>> myfun(2, 1, 43)
 1
>>> myfun(c=43, b=1, a=2)
 1
>>> myfun(2, c=43, b=1)
 1
>>> myfun(c=43, 2, b=1)
 ???
```

Functions are first-class objects

Functions can be used as any other datatype, e.g.,

- Arguments to function
- Return values of functions
- Assigned to variables
- Parts of tuples, lists, etc

```
>>> def square(x):
        return x*x

>>> def applier(q, x):
        return q(x)

>>> applier(square, 7)
49
```

Lambda Notation

Python uses a `lambda` notation to create anonymous functions

Python supports functional programming idioms, including closures and continuation

```
>>> def applier(q, x):
        return q(x)

>>> applier(lambda z: z * 4, 7)
28
```

Example: Lambda Notation

```
>>> f = lambda x,y : 2 * x + y
>>> f
<function <lambda> at 0x87d30>
```

```
>>> f(3, 4)
10
>>> f(7, 1)
???
```

```
>>> v = lambda x: x*x(100)
>>> v
<function <lambda> at 0x87df0>
>>> vx = (lambda x: x*x)(100)
```

```
>>> v(10)
???
>>> vx
10000
```

Example: composition

```
>>> def square(x):
        return x*x
>>> def twice(f):
        return lambda x: f(f(x))
>>> twice
<function twice at 0x87db0>
>>> quad = twice(square)
>>> quad
<function <lambda> at 0x87d30>
```

```
>>> quad(5)
625
```

Explanation:

```
square(square(5)= ???
```

Example: closure

```
>>> def counter(start=0, step=1):
        x = [start]
        def _inc():
            x[0] += step
            return x[0]
        return _inc
```

```
>>> c1 = counter()
>>> c2 = counter(100, -10)
>>> c1()
1
>>> c1()
2
>>> c1()
Guess ???
>>> c2()
90
>>> c2()
Guess ???
```

Logical Expressions

- *True* and *False* are constants in Python.
- Other values equivalent to *True* and *False*:
  - *False*: zero, *None*, empty container or object
  - *True*: non-zero numbers, non-empty objects
- Comparison operators: ==, !=, <, <=, etc.
  - X and Y have same value: **X == Y**
  - Compare with **X is Y** :
    - X and Y are two variables that refer to the *identical same object.*

- You can also combine Boolean expressions.
  - *True* if a is True and b is True: **a and b**
  - *True* if a is True or b is True: **a or b**
  - *True* if a is False: **not a**

Use parentheses as needed to disambiguate complex Boolean expressions.

Special Properties of and & or

- Actually *and* and *or don't* return *True* or *False* but value of one of their sub-expressions, which may be a non-Boolean value
    - `X and Y and Z`
    - If all are true, returns value of Z
- Otherwise, returns value of first false sub-expression
    - `X or Y or Z`
    - If all are false, returns value of Z
- Otherwise, returns value of first true sub-expression
- *and* & *or* use *lazy evaluation*, so no further expressions are evaluated

Conditional Expressions (kind of)

```
x = true_value if condition else false_value
```

Uses lazy evaluation:

- First, `condition` is evaluated
- If *True*, `true_value` is evaluated and returned
- If *False*, `false_value` is evaluated and returned

Standard use:

```
x = (true_value if condition else false_value)
```

Control of Flow: if Statements

```
if x == 3:
  print ("X equals 3.")
elif x == 2:
    print ("X equals 2.")
else:
  print ("X equals something else.")
print ("This is outside the 'if'.")
```

- Any number of elif keyword can be used, same as else if in C language
- Make sure the indentation for if-elif-else remains the same – *see the example on the left*

Control of Flow: while Loops

```
x = 3
while x < 5:
  print (x, "still in the loop")
  x = x + 1
```

Outputs:
3 still in the loop
4 still in the loop

```
x = 6
while x < 5:
  print (x, "still in the loop")
```

Outputs:

- You can use the keyword *break* inside a loop to leave the *while* loop entirely.
- You can use the keyword *continue* inside a loop to stop processing the current iteration of the loop and to immediately go on to the next one.

Works the same as C Language

Python's higher-order functions

```
>>> def square(x):
        return x*x
>>> def even(x):
        return 0 == x % 2
>>> map(square, range(10,20))
[100, 121, 144, 169, 196, 225,
256, 289, 324, 361]
>>> filter(even, range(10,20))
[10, 12, 14, 16, 18]
>>> map(square, filter(even,
range(10,20)))
[100, 144, 196, 256, 324]
```

- Python supports higher-order functions that operate on lists similar to Scheme's
- But many Python programmers prefer to use list comprehensions, instead

List Comprehensions

```
>>>vals = [10,15,20,25]
>>>[x-10 for x in vals]
[0,5,10,15]
```

Another variation with condition

```
>>>[x for x in vals if x%2==0]
[10,20]
```

You can easily complicate your life as:

```
>>>x for x in [y-10 for y in vals] if
x%2==0]
[0,10]
```

- Why "comprehension"? The term is borrowed from math's set comprehension notation for defining sets in terms of other sets
- A powerful and popular feature in Python
- Generate a new list by applying a function to every member of an original list
- Python's notation:

  [expression for name in list]

List Comprehensions (contd.)

```
>>>vals = [10,15,20,25]
>>>[x-10 for x in vals]
[0,5,10,15]
```

Another variation with condition

```
>>>[x for x in vals if x%2==0]
[10,20]
```

You can easily complicate your life as:

```
>>>x for x in [y-10 for y in vals] if
x%2==0]
[0,10]
```

- Why "comprehension"? The term is borrowed from math's set comprehension notation for defining sets in terms of other sets
- A powerful and popular feature in Python
- Generate a new list by applying a function to every member of an original list
- Python's notation:
  [expression for name in list]

List Comprehensions (contd.)

- If list contains elements of different types, then expression must operate correctly on the types of all of list members.
- If the elements of list are other containers, then the name can consist of a container of names that match the type and "shape" of the list members.

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [ n * 3 for (x, n) in li]
[3, 6, 21]
```

expression can also contain user-defined functions.

```
>>> def subtract(a, b):
      return a – b

>>> oplist = [(6, 3), (1, 7), (5, 5)]
>>> [subtract(y, x) for (x, y) in
oplist]
[-3, 6, 0]
```

List Comprehensions (contd.)

List comprehensions can be viewed as syntactic sugar for a typical higher-order functions

```
[ expression for name in list ]
map( lambda name: expression, list )

[ 2*x+1 for x in [10, 20, 30] ]
map( lambda x: 2*x+1, [10, 20, 30] )
```

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem*2 for elem in li if elem > 4]
[12, 14, 18]
```

Only 6, 7, and 9 satisfy the filter condition
So, only 12, 14, and 18 are produce.

- Filter determines whether expression is performed on each member of the list.
- For each element of list, checks if it satisfies the filter condition.
- If the filter condition returns *False*, that element is omitted from the list before the list comprehension is evaluated.

List Comprehensions (contd.)

Including an if clause begins to show the benefits of the sweetened form

[expression for name in list if filt]
map(lambda name . expression, filter(filt, list))

```
[ 2*x+1 for x in [10, 20, 30] if x > 0 ]
```
map(lambda x: 2*x+1, filter(lambda x: x > 0 , [10, 20, 30])

[e1 for n1 in [e1 for n1 list]]
map(lambda n1: e1, map(lambda n2: e2, list))

[2*x+1 for x in [y*y for y in [10, 20, 30]]]
map(lambda x: 2*x+1, map(lambda y: y*y, [10, 20, 30]))

Since list comprehensions take a list as input and produce a list as output, they are easily nested

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
        [item+1 for item in li]
]
[8, 6, 10, 4]
```

The inner comprehension produces: [4, 3, 5, 2]
So, the outer one produces: [8, 6, 10, 4]

For Loops / List Comprehensions

- Python's list comprehensions provide a natural idiom that usually requires a for-loop in other programming languages.
    - As a result, Python code uses many fewer for-loops
    - Nevertheless, it's important to learn about for-loops
- A for-loop steps through each of the items in a collection type, or any other type of object which is "iterable"

```
for <item> in <collection>:
    <statements>
```

Example:

```
for ch in "Hello World":
    print (ch)
```

- If <collection> is a list or a tuple, then the loop steps through each element of the sequence
- If <collection> is a string, then the loop steps through each character of the string
- <item> can be more than a single variable name, e.g.,

```
for (x,y) in [(a,1),(b,2),(c,3)]:
    print (x)
```

For loops & the range() function

- Since a variable often ranges over some sequence of numbers, the *range()* function returns a list of numbers from 0 up to but not including the number we pass to it.
- range(5) returns [0,1,2,3,4]
- So we could say:

```
for x in range(5):
    print (x)
```

(There are more complex forms of range() that provide richer functionality…)

```
>>> ages = { "Sam" : 4, "Mary" : 3, "Bill" : 2 }
>>> ages
{'Bill': 2, 'Mary': 3, 'Sam': 4}

>>> for name in ages:
        print name, ages[name]
Bill 2
Mary 3
Sam 4
```

A Note on Multiple Assignments

We've seen multiple assignment before:

```
>>> x, y = 2, 3
```

- You can also do it with sequences.
- The type and "shape" just has to match.

```
>>> (x, y, (w, z)) = (2, 3, (4, 5))
>>> [x, y] = [4, 5]
```

String Operations and Formatting

- A number of methods for the string class perform useful formatting operations:

```
>>> "hello".upper()
'HELLO'
```

- Check the Python documentation for many other handy string operations.
- The builtin str() function can convert an instance of any data type into a string.

```
>>> “Hello ” + str(2)
“Hello 2”
```

Formatting a string in python. If can be done in two ways.

```
>>> x = "abc"
>>> y = 34
```

The first way is the default python printing style.

```
>>> f"{x} xyz {y}" % (x, y)
'abc xyz 34'
```

The second way works like c. You can use this method for forcing the outputs to be as desired.

```
>>> "%s xyz %d" % (x, y)
'abc xyz 34'
>>> "%s xyz %f" % (x, y)
'abc xyz 34.000000'
```

String operations: Join and Split

Join turns a list of strings into one string

<separator_string>.join(<some_list>)

```
>>> ";".join( ["abc", "def", "ghi"] )
"abc;def;ghi"
```

Split turns one string into a list of strings

<some_string>.split(<separator_string>)

```
>>> "abc;def;ghi".split( ";" )
["abc", "def", "ghi"]
```

Split and join can be used in a list comprehension in the following Python idiom:

```
>>> " ".join( [s.capitalize() for s
in "this is a test ".split( )] )
'This Is A Test`
```

For clarification:

```
>>> "this is a test" .split( )
['this', 'is', 'a', 'test']
>>> [s.capitalize() for s in "this is
a test" .split()]
['This', 'Is', 'A', 'Test’]
```

Next.

- Importing and using existing libraries