COSC 2306
Data Programming

Python Basics

Lists and functions

```
def addItem(list, x):
      list.append(x)

mylist = [1,2,3]
addItem(mylist,4)
print(mylist)
```

>[1, 2, 3, 4]

```
def clearList(list):
      list = []

mylist = [1,2,3]
clearList(mylist)
print(mylist)
```

>[1, 2, 3]


```
mylist.clear()

def clearList(list):
    list = []
    return list

mylist = [1,2,3]
mylist = clearList(mylist)
print(mylist)
```

Lists and for loops

For loops offer a straightforward way to traverse a list

```
fruits = ["apple", "orange", "banana", "cherry"]

for afruit in fruits:      # by item
    print(afruit)
```

```
fruits = ["apple", "orange", "banana", "cherry"]

for position in range(len(fruits)):      # by index
    print(fruits[position])
```

Let's capitalize all the fruits in the list

```
fruits_upper = []
for item in fruits:
    fruits_upper.append(item.upper())

fruits_upper = [item.upper() for item in fruits]
```

List Comprehension

Syntax: newList = [expression(element) for element in oldList if condition]

Example: create a new list from an existing list with items contain “a”

```
# using for loop
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []
for x in fruits:
  if "a" in x:
    newlist.append(x)
print(newlist)
>> ['apple', 'banana', 'mango’]

# using list comprehension
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
>> ['apple', 'banana', 'mango’]
```

Nested List Comprehensions

```
matrix = [[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4],
[0, 1, 2, 3, 4]]


matrix = []
for i in range(5):
    matrix.append([]) # Append an empty sublist inside the list
    for j in range(5):
        matrix[i].append(j)
print(matrix)

matrix = [[j for j in range(5)] for i in range(5)]
print(matrix)
```

Nested List Comprehensions

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
duplicate = [1,2,3]
newlist = [x for x in fruits for y in duplicate if "a" in x]
print(newlist)
```

```
['apple', 'apple', 'apple', 'banana', 'banana', 'banana', 'mango', 'mango',
'mango']
```

```
newlist = []
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
duplicate = [1,2,3]
for x in fruits:
    for y in duplicate:
        if “a” in x:
            newlist.append(x)
print(newlist)
```

String

Name	Purpose
lower()	Returns a string in all lowercase
upper()	Returns a string in all uppercase
center(w)	Returns a string centered in a field of size *w*
find(item)	Returns the index of the first occurrence of item
split(s_char)	Splits a string into substrings at s_char
count(item)	Returns the number of occurrences of item in the string

```
my_name = 'David'
my_name.upper()  # 'DAVID'
my_name.center(10)  # '  David   '
my_name.find('v')   # 2
my_name.split('v') # ['Da', 'id']
```

Tuple

Tuple is very similar to list, but immutable (similar to string)

```
my_tuple = (2,True,4.96)

print(my_tuple)
>>(2, True, 4.96)

print(len(my_tuple))
>>3

print(my_tuple[0])
>>2

print(my_tuple * 3)
>>(2, True, 4.96, 2, True, 4.96, 2, True, 4.96)

print(my_tuple[0:2])
>>(2, True)
```

Set

Name	Purpose	Example
union	Returns a new set with all elements from both sets	set1.union(set2)
intersection	Returns a new set with only the elements common to both sets	set1.intersection(set2)
difference	Returns a new set with all items from first set not in second	set1.difference(set2)
issubset	Asks whether all elements of one set are in the other	set1.issubset(set2)
add	Adds item to the set	set.add(item)
remove	Removes item from the set	set.remove(item)
pop	Removes an arbitrary element from the set	set.pop()
clear	Removes all elements from the set	set.clear()

Dictionaries

- Mapping type → associative collection (key + value)

```
eng2sp = {}
eng2sp['one'] = 'uno'
eng2sp['two'] = 'dos'
eng2sp['three'] = 'tres'

#or
eng2sp = {'three': 'tres', 'one': 'uno', 'two': 'dos'}

print(eng2sp['two’])
>> dos
```

Dictionaries operations

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
```

Operation	Example
Delete (by key)	`del(inventory['pears'])`
Modify	`inventory['pears'] = 0` `inventory['bananas'] = inventory['bananas'] + 200`

Method	Parameters	Description
keys	none	Returns a view of the keys in the dictionary
values	none	Returns a view of the values in the dictionary
items	none	Returns a view of the key-value pairs in the dictionary
get	key	Returns the value associated with key; None otherwise
get	key,alt	Returns the value associated with key; alt otherwise

Dictionaries operations

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.get("price", 15000)

print(x)
```

```
15000
```

Dictionaries and aliasing

Dictionaries have the same aliasing problem as lists (mutable)

```
opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
alias = opposites

print(alias is opposites)

alias['right'] = 'left'
print(opposites['right'])
```

True
left

Use copy instead:
`acopy = opposites.copy()`

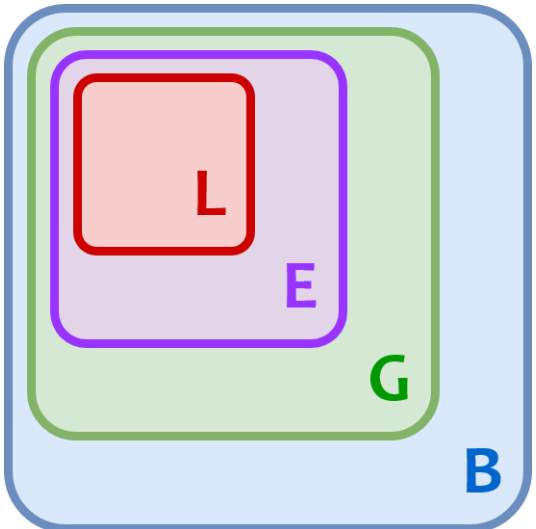
Namespace

1. **Why?**
   - **assignment statement** creates a **symbolic name** that you can use to reference an object
   - hundreds or thousands of such names
   - need a good way to track all these names to avoid interference
2. **How?** namespace - collection of currently defined symbolic names + object information that each name references
3. **What?** a namespace: a dictionary in which the keys are the object names and the values are the objects themselves
4. **Multiple namespaces and variable scope:** the interpreter determines the region where name has meaning (variable scope)
5. **LEGB rule** (Python literature): searches from inside out

Namespace

1. **Local**: refer to x inside a function, the interpreter first searches for the innermost scope (local to that function)
2. **Enclosing**: if x isn't in the local scope but in a function that resides inside another function, the interpreter searches in the enclosing function's scope
3. **Global**: if the above searches fail, the interpreter looks in the global scope
4. **Built-in**: if it can't find x anywhere else, the interpreter tries the built-in scope.

LEGB rule



Namespace

```
x = 'global'  #defines x in the global scope
def f():
        x = 'enclosing' #defines x in the enclosing scope
                def g():
                        x = 'local' #defines x in the scope local to g()
                        print(x)
                g()
f()

>> local
```

Namespace

Python namespace dictionaries

- globals() returns global namespace dictionary

```
x = 'foo'
print(globals())
x is globals()['x']
>>{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'x': 'foo'}
>>True
```

- local() returns local namespace dictionary

```
def f(x, y):
    s = 'foo'
    print(locals())
f(10, 0.5)
>>{'s': 'foo', 'y': 0.5, 'x': 10}
```

Namespace

The global declaration

```
x = 20
def f():
    global x
    x = 40
    print(x)
f()
print(x)
>>40
>>40
```

#x is the x in the global namespace

#if x is not defined (remove x=20 above),
#it will create x as a global variable

not a good practice

-- similar purpose can be achieved by function return value

Manipulating files in Python

- We must "open" a file with a name, which makes it a Python object with a "file handle" for manipulation
-
```
fileHandle = open('myfile.txt')
```

- Future reference to the file data is through fileHandle
- And close the file in the end

Opening Files in Python

- filehandle = open(filename,'r')
  #Open a file called filename and use it for reading

- filehandle = open(filename,'w')
  #Open a file called filename and use it for writing

- filehandle.close( )
  #File use is complete

- *variable filehandle refers to the file content*

Open file options

Mode	Description	Read?	Write?	Overwrite?	Create missing file?
'r'	Read from file	Yes	No	No	No
'w'	Write to file	No	Yes	Yes	Yes
'a'	Append content to file	No	Yes	No	Yes