CS 38003 PYTHON PROGRAMMING

Ruby Tahboub

SEQUENCE TYPES

SEQUENCE TYPES

- String: a sequence of characters enclosed within quotation marks (") or apostrophes(')
  - Strings are **immutable**.
- List: an ordered sequence of arbitrary values
  - Lists are **mutable**.
- Tuples: an ordered sequence of arbitrary values
  - Tuples are **immutable**.
- Set: a collection of unordered and unique values
  - Sets are **immutable**.
- Dictionary: a collection of unordered values accessed by key

STRINGS

```
myString = "Purdue-University"
```

- To select from a string: <string>[<expression>]
- The positions in a string are numbered from the left, starting with 0.

```
print(myString[0])
```

```
P
```

- Strings can be indexed using negative values (-1 index is the last character)

```
print(myString[-1])
```

```
y
```

STRING SLICING

- Slicing enables accessing a contiguous sequence of characters.
- <string>[<start>:<end>]
- start and end should be integers.
- The slice contains the substring at position start and runs up to but doesn't include end.
- If either expression is missing, then the start or the end of the string is used.

STRING SLICING

```
>>>myString = "Purdue-University"
>>>print(myString[0:6])
```
Purdue

```
>>>print(myString[:6])
```
Purdue

```
>>>print(myString[6:])
```
-University

```
>>>print(myString[:])
```
Purdue-University

```
>>>print(myString[7:10])
```
Uni

STRING OPERATIONS

- + concatenates two strings together.

```
>>>print("Good " + "Morning")
```
Good Morning

- * builds up a string by multiple concatenations of a string with itself.

```
>>>print("Good" * 3)
```
GoodGoodGood

- The function len returns the length of a string.

```
>>>print(len("Good"))
```
4

STRING OPERATIONS

- Try the following string operations

```
s = "Purdue University"

>>>print(s.capitalize())
>>>print(s.title())
>>>print(s.count('u'))
>>>print(s.lower())
>>>print(s.rfind('i'))
```

```
Purdue university
Purdue University
2
purdue university
14
```

MORE STRING OPERATIONS

- `ord` returns the ASCII representation of a character

```
>>> ord('A')
65
>>> ord('B')
66
>>> ord('Z')
90
```

- `chr` returns the character of an ASCII value

```
>>> chr(122)
'z'
>>> chr(65)
'A'
>>> chr(97)
'a'
```

- Explore additional operations on your own!

STRINGS ARE IMMUTABLE

- The content of immutable types cannot be changed after they are created.

```
s = "Purdue University"
s[0] = 'X'
```

```
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    s[0] = 'X'
TypeError: 'str' object does not support item assignment
```

LISTS

- Lists are ordered sequence of arbitrary values.

```
myList = ["X" , ["Y1", "Y2"], 3, True, 7.8]
```

- Python lists are stored in memory as an array of references.

LIST INDEXING

```
>>> X = [10,20,30,40,50,60,70,80,90,100]
>>> print (X[0])
10

>>> print (X[-1])
100

>>> print (X[0:5])
[10, 20, 30, 40, 50]

>>> print (X[:3])
[10, 20, 30]

>>> print (X[3:])
[40, 50, 60, 70, 80, 90, 100]

>>> print (X[:-1])
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

LISTS OPERATIONS

```
>>> x1 = ['a', [10,20] ,'Purdue']
>>> x2 = [25, 50]
```

- Concatenation

```
>>> x1 + x2
['a', [10, 20], 'Purdue', 25, 50]
```

- Repetition

```
>>> x2 * 3
[25, 50, 25, 50, 25, 50]
```

LISTS OPERATIONS

```
>>> x1 = ['a',[10,20],'Purdue']
>>> x2 = [25,50]
```

- Length

```
>>> len(x1)
3
```

- Membership

```
>>> 'Purdue' in x1
True
>>> 25 in x1
False
```

- Adding items to the end of the list

```
>>> x2.append('abc')
>>> x2
[25, 50, 'abc']
```

LISTS OPERATIONS

```
>>> x1 = ['a',[10,20],'Purdue']
>>> x2 = [25,50]
```

- Adding a list at the end

```
>>> x1.extend(x2)
>>> x1
['a', [10, 20], 'Purdue', 25, 50]
```

- Inserting item x at index i

```
>>> x2.insert(1,'alpha')
>>> x2
[25, 'alpha', 50, 'abc']
```

- Returning the index of an element

```
>>> x1.index('Purdue')
2
```

LISTS OPERATIONS

```
>>> y1= [10,20,10,40,50]
>>> x2 = [25, 'alpha', 50, 'abc']
```

- Counting the number of times an element appeared in the list

```
>>> y1.count(10)
2
```

- Removing the first occurrence of an item in the list

```
>>> y1.remove(10)
>>> y1
[20, 10, 40, 50]
```

- Removing and returning the last element in the list

```
>>> y1.pop()
50
>>> y1
[20, 10, 40]
```

LISTS OPERATIONS

```
>>> y1= [20,10,40]
```

- Removing and returning the element at index i

```
>>> y1.pop(1)
10
>>> y1
[20, 40]
```

- Deleting the element at index i

```
>>> del y1[1]
>>> y1
[20]
```

LISTS OPERATIONS

```
>>> y2 = [10,4,-1,2,4,7]
```

- Sorting the list

```
>>> y2.sort()
>>> y2
[-1, 2, 4, 4, 7, 10]
```

- Reversing the list

```
>>> y2.reverse()
>>> y2
[10, 7, 4, 4, 2, -1]
```

COPYING LISTS

`list2 = list1`

- results in a shallow copy (`list2` refers to `list1`)

`list2 = list(list1)`

- creates a new list, i.e., new memory location with values equal to `list1`

LIST COMPREHENSION

[<expression> for <variable> in <sequence> if <condition>]

- A list comprehension is a programming construct used for creating a list based on another sequence.
- A powerful and popular feature in Python.
  - Generates a new list by applying a function to every member of another sequence.
  - The if condition is optional.

LIST COMPREHENSION

[<expression> for <variable> in <sequence> if <condition>]

- The <expression> is a an operation that will be performed on each <variable> in the <sequence> if the is <condition> True.
  - For each sequence member that satisfies the condition:
    - Set variable equal to that member.
    - Calculate a new value using expression.
    - Add the new value to a List
  - Return the list.

```
>>> [ x * 2 for x in range(6,15,3)]
[12, 18, 24]
```

LIST COMPREHENSION

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

```
>>> [ x for x in range(20) if x%2 == 0]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

TUPLES

- Tuples are collection of ordered and immutable sequence of elements.
- Tuples are also a sequence like Strings and Lists, so indexing and slicing works with tuples as well.
- A Tuple is similar to a list, the difference being they are immutable.
- No Tuple comprehension, as tuples are immutable and looping and adding values is not allowed
- Tuples definition:

```
T3 = (23, 45, 67)
T3 = 1,5,9
T3 = tuple([2,3,4])
T0 = () # tuple of length 0
T1 = (23, ) #tuple of length 1 must be followed by a comma
```

TUPLES

- Tuples cannot be updated or deleted (immutable).
- New tuples can be created by taking elements from existing tuples:

```
>>> tup1 = (3,4)
>>> lst = list(tup1) # convert the tuple to a list
>>> lst[0] = 5 # modify
>>> tup1 = tuple(lst) # convert back to a tuple
>>> tup1
(5, 4)
```

- Tuples operations are similar to lists.
- No support for operations that modify the sequence (e.g., remove, sort, ect.)

SETS

- A collection of unordered and unique values.
- Follow the abstract mathematical definition of a set: a collection of unique values.
- Common use cases are membership testing, removing duplicates, set operations, etc.

SET OPERATIONS

```
# create an empty set and add elements
>>> s1 = set()
>>> s1.add('A')
>>> s1.add(5)
{5, 'A'}

#create a set using a list of values
>>> s2 = set([1,2,3])
>>> s2
{1, 2, 3}

# testing for membership
>>> 'A' in s1
True

# sets are not subscriptable
>>> s1[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

SET OPERATIONS

```
>>> s1 = {1,2,3,4} # another way to initialize a set.
>>> s2 = {3,4,5,6}
>>> s1 - s2 # set difference
{1, 2}

>>> s1 | s2 # set union
{1, 2, 3, 4, 5, 6}

>>> s1 & s2 # set intersection
{3, 4}

>>> s1 ^ s2 # set symmetric difference
{1, 2, 5, 6}
```

DICTIONARIES- A MAPPING TYPE

- Dictionaries store a mapping between a set of keys and a set of values.
  - Keys can be any immutable type.
  - Values can be any type.
  - Values and keys can be of different types in a single dictionary.
- You can define, modify, view, lookup, delete the key-value pairs in the dictionary.

DICTIONARIES- A MAPPING TYPE

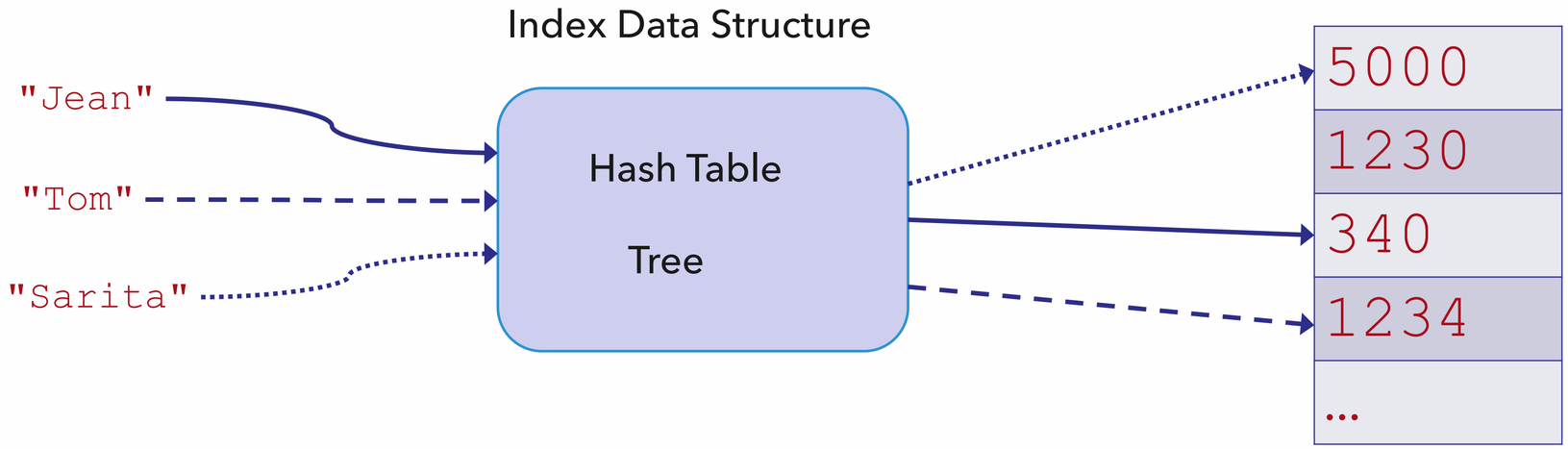
- Examples of Key → Value mappings
  - Name → Phone Number/ Address
  - Word → Frequency of the word in a text
  - Inventory Number → Current Inventory Count

DICTIONARIES- A MAPPING TYPE

- Keys must be unique.
- Assigning to an existing key replaces its value.
- Dictionaries are unordered.
- New entry might appear anywhere in the output.
- Internally, dictionaries work by hashing.

DICTIONARIES- NOT Simply Lookup Tables

```
Savings = {”Jean": 340, ”Tom": 1234, ”Sarita": 5000}
```



CREATING AND ACCESSING DICTIONARIES

- Create a dictionary for person (key) and address (value)

```
>>> address={"John":"Oakwood 345", "Peter":"Evergreen
546", "Mary": "Kingston 564"}
>>> print(address["Mary"])
Kingston 564


>>> print (address)
{'John': 'Oakwood 345', 'Peter': 'Evergreen 546', 'Mary':
'Kingston 564'}


# deleting an entry
>>> del address ["Peter"]
>>> print (address)
{'John': 'Oakwood 345', 'Mary': 'Kingston 564'}
```

DICTIONARY OPERATIONS

```
>>> address["Wayne"]="Young 678"
>>> print (address)
{'John': 'Oakwood 345', 'Mary': 'Kingston 564', 'Wayne': 'Young 678'}

>>> print (address.keys())
dict_keys(['John', 'Mary', 'Wayne'])

>>> print (address.values())
dict_values(['Oakwood 345', 'Kingston 564', 'Young 678'])

# Checking whether a key exists
>>> "John" in address
True
```

DICTIONARY OPERATIONS

```
address={"John":"Oakwood 345", "Peter":"Evergreen
546", "Mary": "Kingston 564"}

for k in address:
  print(k,":", address[k])
```

```
John : Oakwood 345
Peter : Evergreen 546
Mary : Kingston 564
```

```
for k in sorted(address.keys()):
  print(k,":", address[k])
```

```
John : Oakwood 345
Mary : Kingston 564
Peter : Evergreen 546
```

```
>>> phoneNumbers = {}

# creating new items
>>> phoneNumbers["Jean"] = "765 555-2552"
>>> phoneNumbers['Tom'] = "866 555-1234"
>>> phoneNumbers
{'Jean': '765 555-2552', 'Tom': '866 555-1234'}

>>> phoneNumbers['Sarita'] = ["866 555-3340", "765 555-4917"]
>>> phoneNumbers["Sarita"][0]
'866 555-3340'

>>> phoneNumbers["Bob"] = 7655558923

>>> list(phoneNumbers.items())
[('Jean', '765 555-2552'),
 ('Tom', '866 555-1234'),
 ('Sarita', ['866 555-3340', '765 555-4917']),
 ('Bob', 7655558923)]
```

```
# is Lawrence in the dictionary phoneNumbers?
# if not phoneNumbers["Lawrence"] produces an error
# if it's a list, writing a new number will overwrite it
>>> if "Lawrence" in phoneNumbers:
...   print ("already entered")
... else:
      phoneNumbers["Lawrence"] = 7655550198
>>> phoneNumbers
{'Jean': '765 555-2552',
 'Tom': '866 555-1234',
 'Sarita': ['866 555-3340', '765 555-4917'],
 'Lawrence': 7655550198}

# Dictionaries cannot be sorted.
# We can get the keys in sorted order (this does not change the dictionary)
>>> for f in sorted(list(phoneNumbers)):
...   print(f)

Jean
Lawrence
Sarita
Tom
```

COMPREHENSION

- Initialize the following dictionary using comprehension:

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

```
d = {x: x * x for x in range(5)}
```

LOOPS ON SEQUENCES

- The for loop works with sequences

```
friends = ["Alex", "John", "Hudson", "Susan", "Yamini"]
```

```
# approach 1: for loop with in
for f in friends:
    print(f)
```

```
# approach 2: for loop with range
for f in range(len(friends)):
   print(friends [f])
```

```
Alex

John

Hudson

Susan

Yamini
```

LOOPS ON NESTED LISTS

- Nested lists are list of lists
- Nested loops are used to traverse nested list.

```
myList = [['A', 'B'], ['X', 'Y', 'Z'],[1,2,3]]
```

```
# approach 1: for loop with in
for row in myList:
    for element in row:
        print (element, end = " ")
    print()
```

```
# approach 2: for loop with range
for i in range(len(myList)):
    for j in range(len(myList[i])):
        print(myList[i][j], end=" ")
    print()
```

```
A B
X Y Z
1 2 3
```

THANK YOU!