Programming Lab

Autumn Semester

Course code: PC503

I can toast!
Let me rephrase that
I can do ANYTHING you tell me to...
... provided you speak my language

Dr. Rahul Mishra
Assistant Professor
DA-IICT, Gandhinagar

Lecture 11

Different Data Types

Tuples and Sequences

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton = 'hello'
>>> len(singleton)
5
>>> singleton
'hello'
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(singleton)
1
>>> singleton
('hello',)
>>>
```

Tuples and Sequences

The statement t = 12345, 54321, 'hello!' is an example of tuple packing: the values 12345, 54321 and 'hello!' are packed together in a tuple. The reverse operation is also possible:

```
>>> t = 12345, 54321, 'hello!'
>>> t
(12345, 54321, 'hello!')
>>>
>>> x, y, z = t
>>> t
(12345, 54321, 'hello!')
>>> x
12345
>>> y
54321
>>> z
'hello!'
>>>
```

Sets

1. A set is an unordered collection with no duplicate elements.

2. Basic uses include membership testing and eliminating duplicate entries.

3. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the set() function can be used to create sets.

Note: to create an empty set you have to use set(), not {};

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)  #Shows the duplicate elements are removed
{'banana', 'apple', 'orange', 'pear'}
>>>
>>> 'orange' in basket              # fast membership testing
True
>>> 'crabgrass' in basket
False
>>>
```

Sets

# Demonstrate set operations on unique letters from two words

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'d', 'b', 'r', 'a', 'c'}
>>> a - b
{'d', 'b', 'r'}
>>> a | b # letters in a or b or both
{'m', 'd', 'b', 'r', 'z', 'a', 'c', 'l'}
>>> a & b
{'a', 'c'}
>>> a ^ b
{'m', 'r', 'd', 'b', 'l', 'z'}
>>>
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'d', 'r'}
>>>
```

Sets

# Demonstrate set operations on unique letters from two words

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'d', 'b', 'r', 'a', 'c'}
>>> a - b
{'d', 'b', 'r'}
>>> a | b # letters in a or b or both
{'m', 'd', 'b', 'r', 'z', 'a', 'c', 'l'}
>>> a & b
{'a', 'c'}
>>> a ^ b
{'m', 'r', 'd', 'b', 'l', 'z'}
>>>
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'d', 'r'}
>>>
```

Dictionaries

- Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”.
- Unlike sequences, which are indexed by a range of numbers, ***dictionaries are indexed by keys***, which can be any immutable type; **strings and numbers can always be keys**.
- Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object ***either directly or indirectly, it cannot be used as a key.***
- You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().
- It is best to think of a dictionary as a **set of key: value pairs**, with the requirement that the keys are unique (within one dictionary).
- A pair of braces creates an empty dictionary: {}.
- Placing a comma-separated list of **key: value pairs within the braces** adds initial key: value pairs to the dictionary; this is also the way dictionaries are written on output.

Dictionaries

- The main operations on a dictionary are ***storing a value with some key and extracting the value given*** the key.
- It is also possible to delete **a key: value pair with del**.
- If you store using a key that is already in use, the old value associated with that key is forgotten.
- It is an error to extract a value using a non-existent key.
- Performing list(d) on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use sorted(d) instead).

Dictionaries

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
>>>
```

Dictionaries

The **dict()** constructor builds dictionaries directly from sequences of key-value pairs:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

In addition, **dict** comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
>>>
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Dictionaries

Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
>>>
```

Dictionaries
Looping Techniques

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the zip() function

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed() function.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

To loop over a sequence in sorted order, use the sorted() function which returns a new sorted list while leaving the source unaltered.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange',
'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
>>>
```

Using set() on a sequence eliminates duplicate elements.

The use of sorted() in combination with set() over a sequence is an idiomatic way to loop over unique elements of the sequence in sorted order.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange',
'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
>>>
```

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5,
float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
>>>
```

More on Conditions

- The conditions used in **while and if** statements can contain any operators, not just comparisons.
- The comparison **operators in and not** in are membership tests that determine whether **a value is in (or not in) a container.**
- The **operators is and is not compare whether two objects are really the same object**.
- All comparison operators have the same priority, which is lower than that of all numerical operators.
- Comparisons can be chained. **For example, a < b == c tests** whether a is less than b, and moreover b equals c.
- Comparisons may be combined using the Boolean operators **and and or, and** the outcome of a comparison (or of any other Boolean expression) may be negated with not.

More on Conditions

It is possible to assign the result of a comparison or other Boolean expression to a variable.

For example,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer
Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Comparing Sequences and Other Types

```
>>> (1, 2, 3)          < (1, 2, 4)
True
>>> (1, 2, 3)          < (1, 2, 4)
True
>>> [1, 2, 3]          < [1, 2, 4]
True
>>> 'ABC' < 'C' < 'Pascal' < 'Python'
True
>>> (1, 2, 3, 4)        < (1, 2, 4)
True
>>> (1, 2)             < (1, 2, -1)
True
>>> (1, 2, 3)          == (1.0, 2.0, 3.0)
True
>>> (1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```