ISE 314X
Computer Programing for Engineers

More on Lists, Tuples, Dictionaries, and Sets

Objectives

- To understand the concepts of lists, tuples, dictionary, and sets

Tuples

- Tuples are another sequence type like lists
- Lists are enclosed in square brackets "[ ]" and tuples in parentheses "( )"

```
>>> [10, 20, 30, 40, 50]
[10, 20, 30, 40, 50]
>>> ["spam", "bungee", "swallow"]
['spam', 'bungee', 'swallow']
>>> (2, 4, 6, 8)
(2, 4, 6, 8)
>>> ("two", "four", "six", "eight")
('two', 'four', 'six', 'eight')
```

Tuples

```
>>> ["cheese", ("red", "rojo"), [3, 5]]
['cheese', ('red', 'rojo'), [3, 5]]
>>> type(["cheese", ("red", "rojo"), [3, 5]])
<class 'list'>

>>> ("cheese", ("red", "rojo"), [3, 5])
('cheese', ('red', 'rojo'), [3, 5])
>>> type(("cheese", ("red", "rojo"), [3, 5]))
<class 'tuple'>
```

Tuples

- It is possible to drop the parentheses when specifying a tuple, and only use a sequence of values separated by commas:

```
>>> x = 2, 4, 6
>>> x
(2, 4, 6)
>>> type(x)
<class 'tuple'>
```

Tuples

- Also, it is required to include a comma when specifying a tuple with only one element:

```
>>> singleton = 2,
>>> singleton
(2,)
>>> singleton = (2,)
>>> singleton
(2,)
>>> type(singleton)
<class 'tuple'>
```

Tuples

```
>>> not_tuple = 2
>>> not_tuple
2
>>> not_tuple = (2)
>>> not_tuple
2
>>> type(not_tuple)
<class 'int'>
>>> empty_tuple = ()
>>> type(empty_tuple)
<class 'tuple'>
```

Tuples

- The indexing and slicing of tuples is similar to strings and lists

```
>>> prices = (3.99, 6.00, 10.00, 5.25)
>>> prices[3]
5.25
>>> prices[:2]
(3.99, 6.0)
>>> pairs = [('cheese', 'queso'),
             ('red', 'rojo'),
             ('school', 'escuela')]
>>> pairs[1:]
???
```

Tuples

- With lists and tuples, `len` returns the number of elements in the sequence:

```
>>> len(['a', 'b', 'c', 'd'])
4
>>> len((2, 4, 6, 8))
4
>>> pairs = [('cheese', 'queso'),
             ('red', 'rojo'),
             ('school', 'escuela')]
>>> len(pairs)
???
```

Tuples

- The `enumerate` function traverse a list or tuple
- It return both the index and the value of each element

Tuples

```
>>> fruits = ('apple', 'banana', 'blueberry')

>>> for frname in fruits:
...     print(frname)
apple
banana
blueberry

>>> for index, frname in enumerate(fruits):
...     print(frname,"is in position", index)
apple is in position 0
banana is in position 1
blueberry is in position 2
```

Tuples

- The `in` operator returns whether a given element is contained in a list or tuple:

```
>>> stuff = ['this', 'that', 'these', 'those']
>>> 'this' in stuff
True
```

- **Questions**

```
>>> 'everything' in stuff
>>> 5 in (2, 4, 6, 8)
>>> 'i' in 'apple'
>>> 'ap' in 'apple'
```

Mutability

- **Lists are mutable, but strings and tuples are not**:

```
>>> a = [2, 3]
>>> a[0] = 1
>>> a
[1, 3]

>>> b = (2, 3)
>>> b[0] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
```

Mutability

```
>>> myString = "Hello World"
>>> myString[2]
'l'
>>> myString[2] = "p"
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    myString[2] = "p"
TypeError: object doesn't support item
  assignment
```

Why Use Tuples?

- Python actually does quite a bunch of tuple stuff behind the scenes. For example, simultaneous assignment:

```
>>> x, y = 1, 2
```

Why Use Tuples?

- Python also uses tuples whenever we want to return multiple values from a function

```
# quadratic2.py
import math
def quadraticSolver(a, b, c):
    disc = math.sqrt(b**2 - 4 * a * c)
    root1 = (-b + disc)/(2*a)
    root2 = (-b - disc)/(2*a)
    return root1, root2

r1, r2 = quadraticSolver(1, 3, 1)
print(r1, r2)
```

Why Use Tuples?

- Run this program:

```
-0.3819660112501051 -2.618033988749895
```

List Deletion

- List deletion

```
>>> a = ['a', 'b', 'c', 'd']
>>> del a[1:3]
>>> a
['a', 'd']
```

List Methods

```
>>> mylist = []
>>> mylist
[]
>>> mylist.append('this')
>>> mylist
['this']
>>> mylist.append('that')
>>> mylist
['this', 'that']
>>> mylist.insert(1, 'thing')
>>> mylist
['this', 'thing', 'that']
```

List Methods

```
>>> mylist.sort()
>>> mylist
['that', 'thing', 'this']
>>> mylist.remove('thing')
>>> mylist
['that', 'this']
>>> mylist.reverse()
>>> mylist
['this', 'that']
```

Dictionaries

- A dictionary represents a list of key-value pairs contained in curly braces, {}

```
>>> inventory = {'apples': 430, 'bananas': 312,
...      'oranges': 525}
>>> inventory['bananas']
312
```

- Dictionaries are mutable

Dictionaries

```
>>> del inventory['bananas']
>>> print(inventory)
{'apples': 430, 'oranges': 525}
>>> inventory['oranges'] = 0
>>> print(inventory)
{'apples': 430, 'oranges': 0}
>>> len(inventory)
2
>>> 'apples' in inventory
True
>>> 'blueberries' in inventory
False
```

Dictionaries

```
>>> spanish = dict()
>>> spanish
{}
>>> spanish['hello'] = 'hola'
>>> spanish['yes'] = 'si'
>>> spanish
{'hello': 'hola', 'yes': 'si'}
>>> spanish['yes']
'si'
```

ISE 314X
Computer Programing for Engineers

More on Lists, Tuples, Dictionaries, and Sets (Part II)

Yong Wang
Assistant Professor
Systems Science & Industrial Engineering
Binghamton University

Sets

- A set is a data type that holds an unordered collection of unique elements
- Python uses curly braces to indicate a set, but with elements instead of key-value pairs:

```
>>> a = {'apples':32, 'bananas':47, 'pears':17}
>>> type(a)
<class 'dict'>
>>> b = {'apples', 'bananas', 3.14, -5}
>>> type(b)
<class 'set'>
```

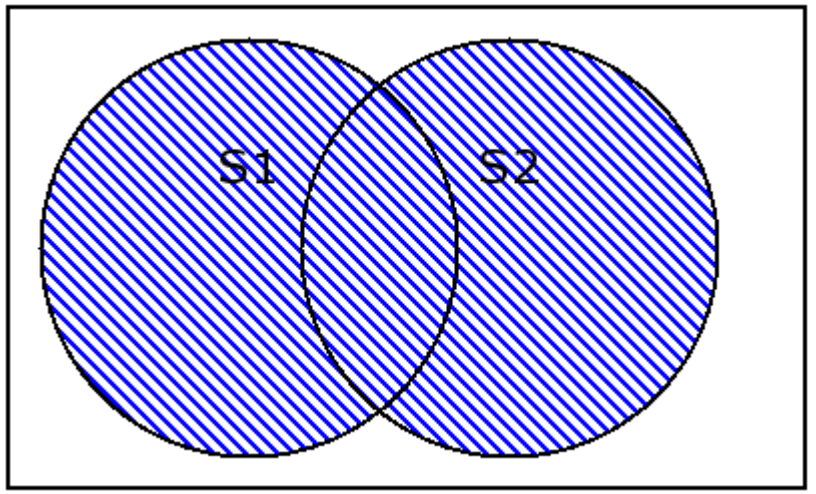
Sets

```
>>> set_of_numbers = {1, 2, 3, 4}
>>> set_of_numbers
{1, 2, 3, 4}
>>> set_of_numbers.add(5)
>>> set_of_numbers
{1, 2, 3, 4, 5}
>>> 3 in set_of_numbers
True
>>> 6 in set_of_numbers
False
>>> list_of_numbers = [1, 2, 1, 3, 4, 4]
>>> set(list_of_numbers)
{1, 2, 3, 4}
```

Sets

- Set Union

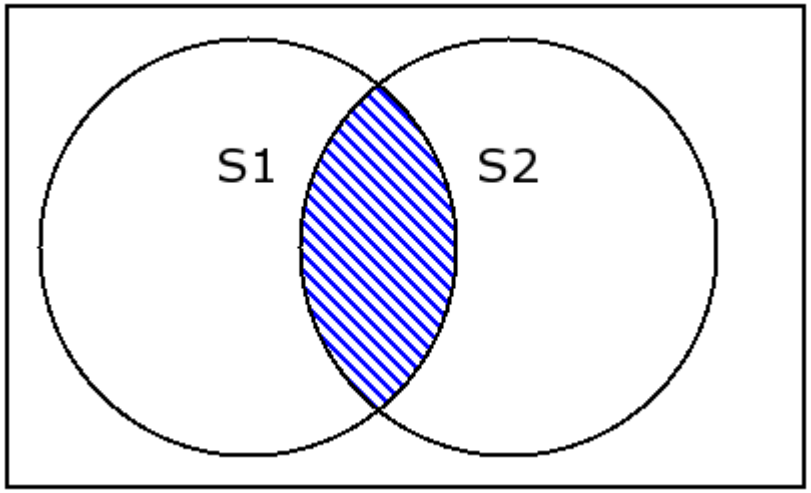
```
>>> s1 = {1, 2, 3, 4}
>>> s2 = {1, 2, 5, 6}
>>> s1 | s2
{1, 2, 3, 4, 5, 6}
```



Sets

- Set Intersection

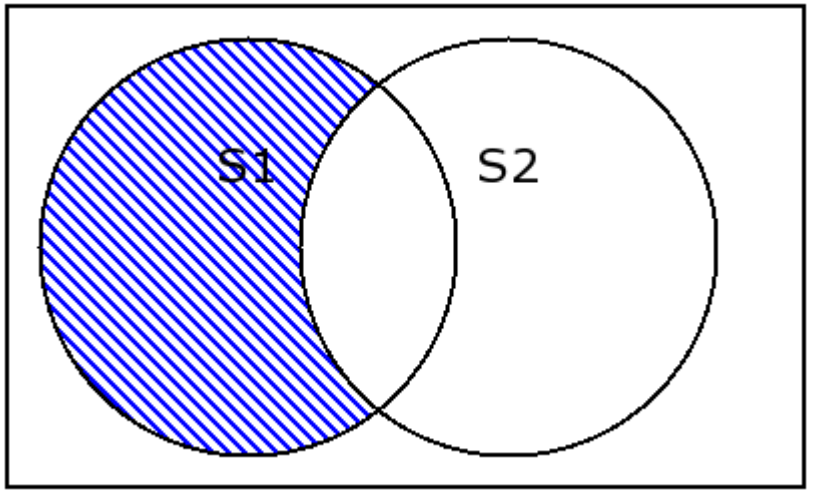
```
>>> s1 = {1, 2, 3, 4}
>>> s2 = {1, 2, 5, 6}
>>> s1 & s2
{1, 2}
```



Sets

- Set Difference

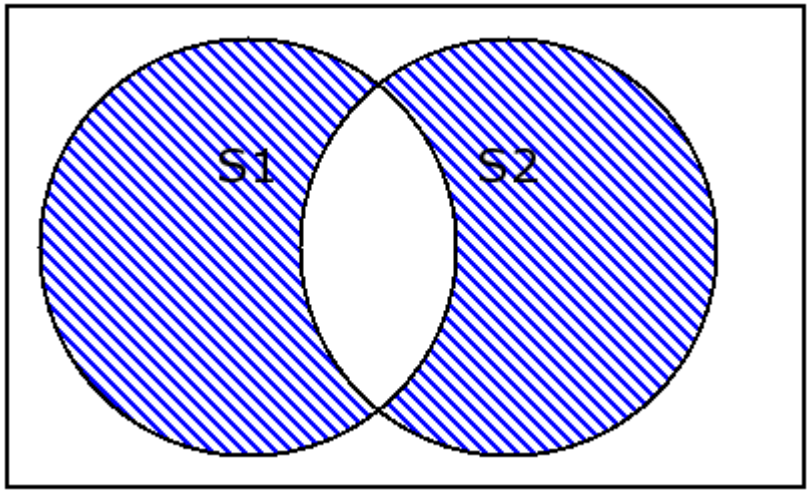
```
>>> s1 = {1, 2, 3, 4}
>>> s2 = {1, 2, 5, 6}
>>> s1 - s2
{3, 4}
```



Sets

- Symmetric difference

```
>>> s1 = {1, 2, 3, 4}
>>> s2 = {1, 2, 5, 6}
>>> s1 ^ s2
{3, 4, 5, 6}
```



Sets

- Subsets

```
>>> s1 = {1, 2, 3, 4}
>>> s2 = {1, 2, 5, 6}
>>> s3 = {1, 2}
>>> s1 <= s2  # s1 ⊆ s2?
False
>>> s1 > s3   # s1 ⊃ s3?
True
```

Sets

```
>>> engineers = {'Jane', 'Jack', 'Julie'}
>>> programmers = {'Jack', 'Sam', 'Susan'}
>>> managers = {'Jane', 'Susan', 'Zack'}

>>> allemployees = engineers | programmers | managers
>>> allemployees
{'Susan', 'Sam', 'Julie', 'Jack', 'Jane', 'Zack'}

>>> eng_management = engineers & managers
>>> eng_management
???

>>> fulltime_management = managers-engineers-programmers
>>> fulltime_management
???
```

Sets

```
>>> engineers.add('Mark')        #Mark joins
>>> allemployees.add('Mark')
>>> engineers
{'Mark', 'Jack', 'Julie', 'Jane'}
>>> allemployees
{'Susan', 'Sam', 'Julie', 'Jack', 'Jane', 'Zack', 'Mark'}

>>> for group in [engineers, programmers, managers,
...     allemployees]:
...     group.discard('Susan')   #Susan leaves
...     print(group)
{'Mark', 'Jack', 'Julie', 'Jane'}
{'Jack', 'Sam'}
{'Zack', 'Jane'}
{'Sam', 'Julie', 'Jack', 'Jane', 'Zack', 'Mark'}
```

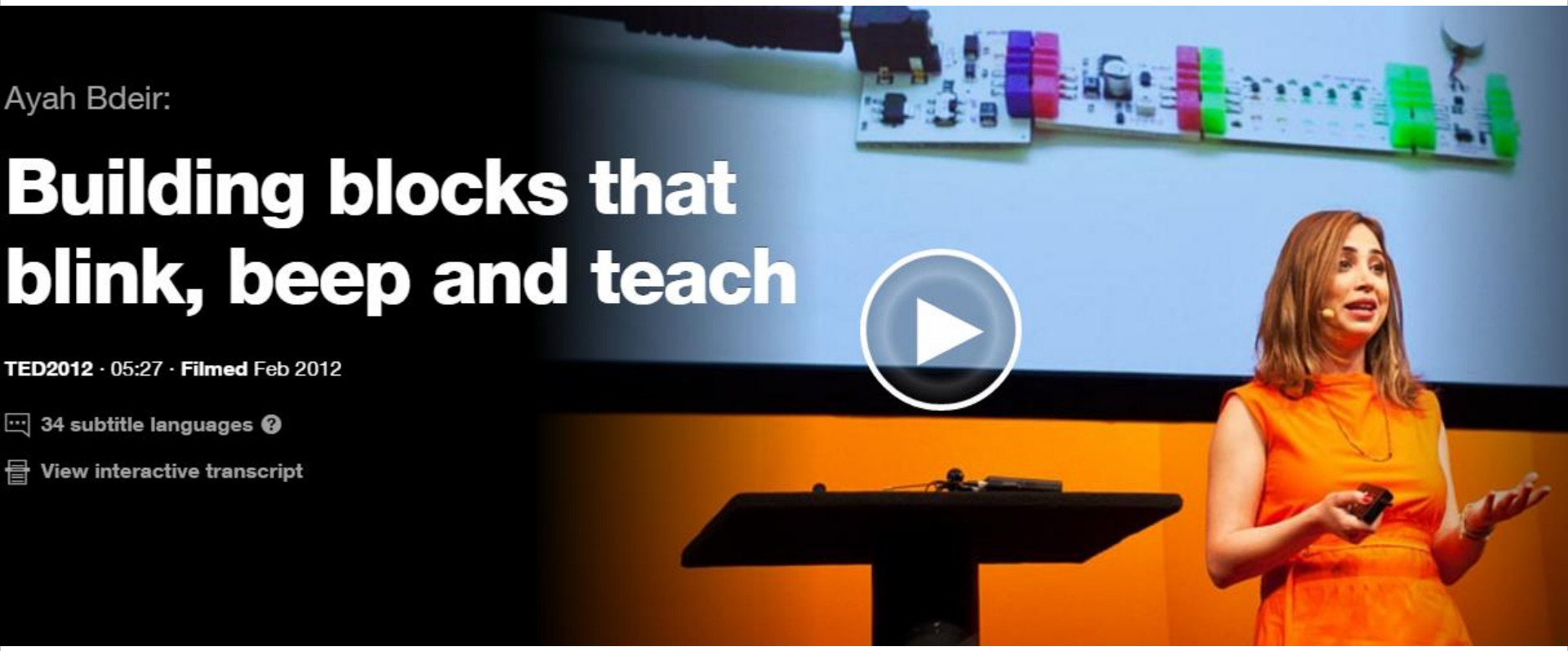
Sets

- Sets do not support indexing, slicing, or other sequence-like behavior

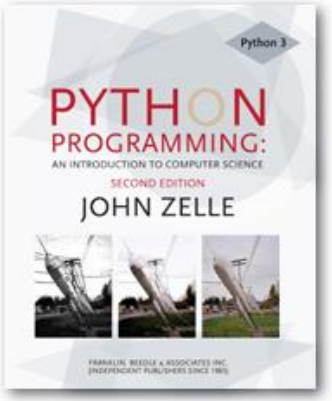
```
>>> s3 = {1, 2}
>>> s3[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support
indexing
```

- **Why? What about len?**

TED Talk:
A Different Kind of Programming



(5)



ISE 314X
Computer Programing for Engineers

Chapter 6
Defining Functions

Objectives

- To understand function calls and parameter passing
- To use functions to reduce code duplication and increase program modularity

Types of Functions

- Different types of functions:
  - Our programs comprise a single function called main
  - Built-in functions (print, range, eval, abs, etc.)
  - Functions from the standard libraries (math.sqrt)

Why Do We Need Functions?

- Having similar or identical code in more than one place has some drawbacks
  - Writing the same code twice or more
  - This same code must be maintained in two separate places

Functions, Informally

- A function is like a *subprogram*, a small program within a program
- Function definition: Write a sequence of statements and then give that sequence a name
- Calling the function: Execute this sequence by referring to the name

Functions, Informally

- Happy Birthday lyrics

```
>>> def main():
...     print("Happy birthday to you" )
...     print("Happy birthday to you" )
...     print("Happy birthday, dear Fred")
...     print("Happy birthday to you")
```

- Call the function

```
>>> main()
Happy birthday to you
Happy birthday to you
Happy birthday, dear Fred
Happy birthday to you
```

Functions, Informally

- Duplicated code:

```
print("Happy birthday to you")
```

- Define a function to print out this line

```
>>> def happy():
...     print("Happy birthday to you")
```

Functions, Informally

- Rewrite the program

```
>>> def singFred():
...     happy()
...     happy()
...     print("Happy birthday, dear Fred")
...     happy()
```

- Call the function

```
>>> singFred()
Happy birthday to you
Happy birthday to you
Happy birthday, dear Fred
Happy birthday to you
```

Functions, Informally

- What if it's Lucy's birthday?

```
>>> def singLucy():
...     happy()
...     happy()
...     print("Happy birthday, dear Lucy")
...     happy()
```

Functions, Informally

- Write a main program to sing to both

```
>>> def main():
...     singFred()
...     print()
...     singLucy()

>>> main()
Happy birthday to you
Happy birthday to you
Happy birthday, dear Fred
Happy birthday to you

Happy birthday to you
Happy birthday to you
Happy birthday, dear Lucy
Happy birthday to you
```

Functions, Informally

- There's still a lot of code duplication

```
>>> def singFred():
...     happy()
...     happy()
...     print("Happy birthday, dear Fred")
...     happy()
>>> def singLucy():
...     happy()
...     happy()
...     print("Happy birthday, dear Lucy")
...     happy()
```

Functions, Informally

- The generic function *sing*

```
>>> def sing(person):
...     happy()
...     happy()
...     print("Happy birthday, dear", person)
...     happy()
```

- This function uses a parameter named person

Functions, Informally

```
>>> sing("Fred")
Happy birthday to you
Happy birthday to you
Happy birthday, dear Fred
Happy birthday to you
```

Functions, Informally

```
# happy.py

def happy():
    print("Happy Birthday to you!")

def sing(person):
    happy()
    happy()
    print("Happy birthday, dear", person + ".")
    happy()

def main():
    sing("Fred")
    print()
    sing("Lucy")

main()
```

Functions, Informally

- Run the program:

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred.
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lucy.
Happy birthday to you!
```