DSCI 510
PRINCIPLES OF PROGRAMMING FOR DATA SCIENCE

Itay Hen

Information Sciences Institute

USC Viterbi
School of Engineering

Ordered, Immutable

TUPLES

Tuples Are Like Lists

Tuples are another kind of sequence that functions much like a list
– they have elements which are indexed starting at 0
– you can iterate over the items in a tuple
– items in tuples are ordered

```
>>> t = ('Glenn', 'Sally', 'Joseph')
>>> print(t[2])
Joseph

>>> print(type(t))
<class 'tuple'>
```

```
>>> y = (1, 9, 2)
>>> print(y)
(1, 9, 2)

>>> print(max(y))
9
```

```
>>> for item in y:
...     print(item)
...
1
9
2
```

but… Tuples are immutable

Unlike a list, once you create a **tuple**, it's contents cannot be altered - similar to int, str,…

```
>>> x = [9, 8, 7]
>>> x[2] = 6

>>> print(x)
[9, 8, 6]
```

```
>>> y = 'ABC'
>>> y[2] = 'D'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not
support item assignment
```

```
>>> z =(3, 6, 9)
>>> z[2] = 8

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

Tuples are immutable

```
>>> x = (1, 2, 4)
>>> x
(1, 2, 4)
>>> id(x)
4316925056
>>> x.add(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'add'
>>> x.append(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> x[2] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> x += (3,)
>>> x
(1, 2, 4, 3)
>>> id(x)
4316792448
>>>
```

create a new tuple

print the ID

tuple does not support add(), append() or item assignment

The assignment operator += works, you can add another tuple. Notice the , after 3 for single valued tuples

But the ID has changed, so += created a new tuple instead of changing the previous tuple

Tuples as Immutable Lists

Often Tuples are introduced as “immutable lists”. Two key benefits ,

Clarity

When you see a tuple on code, you know its length will never change

Performance

A tuple uses less memory than a list of the same length, and it allows Python to do some optimizations.

In our code, when we have to make “temporary variables” we prefer tuples over lists.

Things **not** to do with Tuples

```
>>> t = (3, 2, 1)
>>> t.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'sort'
>>> t.append(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> t.reverse()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'reverse'
>>>
```

Tuples and Assignments

- We can also put a tuple on the left-hand side of an assignment statement

```
>>> x, y = 'dsci', 510
>>> print(y)
510
```

- Tuples as a data structure are often used to return multiple values from functions.

- Or as the (key, value) iteration variable when iterating over dictionaries.

Tuples are Comparable

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> ('Jones', 'Sally') < ('Jones', 'Sam')
True
>>> ('Jones', 'Sally') > ('Adams', 'Sam')
True
>>>
```

- The comparison operators work with tuples and other sequences.
- If the first item is equal, Python goes on to the next item, and so on, until it finds items that differ

Tuples as Records

- Tuples can hold records, each item in the tuple holds the data for one field, and the position of item gives it meaning.
- When using a tuple as a record, the number of items is usually fixed, and their order is always important.
- Sorting a tuple used as a record would destroy the information because meaning of each field is given by its position in the tuple
- Let's see some examples …

Tuples as Records: Examples

```
lax_coordinates = (33.9425, -118.408056)
```

Latitude and Longitude of LAX

```
city, year, pop, chg, area = ('Tokyo', 2003, 32_450, 0.66, 8014)
```

Data about Tokyo: (name, year, population, population change, area in square kms)

```
traveler_ids = [('USA', '31195855'), ('BRA', 'CE342567'),
                ('ESP', 'XDA205856')]
```

A list of tuples of form (country, passport_number)

```
for passport in traveler_ids:
    print('%s/%s' % passport)
```

```
USA/31195855
BRA/CE342567
ESP/XDA205856
```

The % formatting operator understands tuples and treats each item as separate field.

```
for country, _ in traveler_ids:
    print(country)
```

```
USA
BRA
ESP
```

An example of “unpacking”. We assign the second items to _ as we are not interested in it.

Unordered collection of immutable elements without duplicates

SETS

Sets

- A set is a collection of unique objects.
- A set is unordered.
- Set elements must be hashable (i.e., have a hash value; a numeric fingerprint used for fast lookups).
- Membership testing is very efficient, driven by hash tables.
- Typical set operations: union, intersection, difference, symmetric difference.

Creating Sets

```
s = {'jack', 'sjoerd'}
print(s)
```

1. using a comma-separated list of elements within curly braces

```
{'sjoerd', 'jack'}
```

```
s2 = set('foobar')
print(s2)
```

2. using the set constructor set() from a string

```
{'b', 'o', 'r', 'f', 'a'}
```

```
s3 = set(['a', 'b', 'foo'])
print(s3)
```

3. using the set constructor set() from a list

```
{'b', 'a', 'foo'}
```

```
s4 = {}
print(type(s4))
```

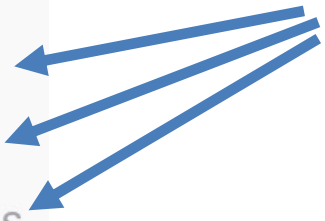
4. oops, {} is a dict, use set() to create an empty set

```
<class 'dict'>
```

Sets are mutable

- We can **add, remove, or modify** its elements in place using methods like:

```
s = {1, 2, 3}
s.add(4)          # adds an element
s.remove(2)       # removes an element
s.update({5,6}) # adds multiple elements
```



This is an example for the use of methods (functions tied to an object)

- Their id() will remain the same.
- The elements inside a set must be immutable (hashable), e.g., numbers, strings, tuples—but not lists or other sets.

Set Operation: union

```
odds = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
evens = {2, 4, 6, 8, 10, 12, 14, 16, 18}
primes = {2, 3, 5, 7, 11, 13, 17, 19}
```

Set Literals

```
print(odds.union(evens))
```

Set Union

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19}

```
print(odds | evens)
```

Set Union using the | operator

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19}

```
print(odds)
```

{1, 3, 5, 7, 9, 11, 13, 15, 17, 19}

The sets odds and evens are unchanged

```
print(evens)
```

{2, 4, 6, 8, 10, 12, 14, 16, 18}

Set Operation: intersection

```
odds = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
evens = {2, 4, 6, 8, 10, 12, 14, 16, 18}
primes = {2, 3, 5, 7, 11, 13, 17, 19}
```

Set Literals

```
print(odds.intersection(evens))
```

Set Intersection

```
set()
```

Why set() and not {}?

```
print(odds.intersection(primes))
```

Set Intersection

```
{3, 5, 7, 11, 13, 17, 19}
```

```
print(odds & primes)
```

Set Intersection using the & operator

```
{3, 5, 7, 11, 13, 17, 19}
```

Set Operation: difference

```
odds = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
evens = {2, 4, 6, 8, 10, 12, 14, 16, 18}
primes = {2, 3, 5, 7, 11, 13, 17, 19}
```

Set Literals

```
print(odds.difference(primes))
```

Set difference

```
{1, 9, 15}
```

```
print(primes - odds)
```

Set difference using the - operator

```
{2}
```

Set Operation: Symmetric Difference

Symmetric Difference (^) between two sets A and B is defined as:

$$A \wedge B = (A \mid B) - (A \,\&\, B)$$

```
odds = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
primes = {2, 3, 5, 7, 11, 13, 17, 19}
```

Set Literals

```
print((odds | primes) - (odds & primes))
```

Symmetric Difference using the formula

{1, 2, 9, 15}

```
print(odds.symmetric_difference(primes))
```

Symmetric Difference using the built-in method

{1, 2, 9, 15}

```
print(odds ^ primes)
```

Symmetric Difference using the ^ operator

{1, 2, 9, 15}

```
print(odds ^ primes == primes ^ odds)
```

Symmetric Difference is commutative

True

Set Inclusion: Subset

```
s1 = {1, 2, 3}
s2 = {1, 2, 3}
s3 = {1, 2, 3, 4}
```

issubset(other) set <= other **Test whether every element in the set is in** other.

```
print(s1 <= s2)
```

Check if s1 is a subset of s2 using the <= operator

```
True
```

```
print(s1.issubset(s2))
```

Check if s1 is a subset of s2 using the issubset() method

```
True
```

```
print(s1.issubset(s3))
```

Check if s1 is a subset of s3 using the issubset() method

```
True
```

Set Inclusion: Superset

```
s1 = {1, 2, 3}
s2 = {1, 2, 3}
s3 = {1, 2, 3, 4}
```

issuperset(other) set >= other Test whether every element in other is in the set.

```
print(s1 >= s2)
```

Check if s1 is a superset of s2 using the <= operator

True

```
print(s1.issuperset(s2))
```

Check if s1 is a superset of s2 using the issuperset() method

True

```
print(s3.issuperset(s2))
```

Check if s3 is a superset of s2 using the issuperset() method

True

Set Inclusion: Proper Subset

```
s1 = {1, 2, 3}
s2 = {1, 2, 3}
s3 = {1, 2, 3, 4}
```

set < other Test whether the set **is a proper subset of** other**, that is,** set <= other **and** set != other.

```
print(s1 < s1)
```

Check if s1 is a proper subset of itself

False

```
print(s1 < s2)
```

Check if s1 is a proper subset of s2

False

```
print(s1 < s3)
```

Check if s1 is a proper subset of s3

True

Set Inclusion: Proper Superset

```
s1 = {1, 2, 3}
s2 = {1, 2, 3}
s3 = {1, 2, 3, 4}
```

set > other Test whether the set **is a proper superset of** other**, that is,** set >= other **and** set != other.

```
print(s1 > s2)
```

Check if s1 is a proper superset of s2

False

```
print(s3 > s1)
```

Check if s3 is a proper superset of s1

True

Set Methods

len(s) Return the number of elements in set s (cardinality of s).

x in s Test x for membership in s.

x not in s Test x for non-membership in s.

isdisjoint(other) Return True if the set has no elements in common with other. Sets are disjoint if and only if their intersection is the empty set.

issubset(other) set <= other Test whether every element in the set is in other.

set < other Test whether the set is a proper subset of other, that is, set <= other and set != other.

issuperset(other) set >= other Test whether every element in other is in the set.

set > other Test whether the set is a proper superset of other, that is, set >= other and set != other.

Set Methods

union(*others) set | other | ... Return a new set with elements from the set and all others.

intersection(*others) set & other & ... Return a new set with elements common to the set and all others.

difference(*others) set - other - ... Return a new set with elements in the set that are not in the others.

symmetric_difference(other) set ^ other Return a new set with elements in either the set or other but not both.

```
# Define several sets
a = {1, 2, 3}
b = {3, 4, 5}
c = {5, 6, 7}
```

```
# Use union with multiple arguments
result = a.union(b, c)

print(result)
```

Output:

```
{1, 2, 3, 4, 5, 6, 7}
```

Set Methods

difference_update(others*)** set -= other | ... Update the set, removing elements found in others.

symmetric_difference_update(*other*) set ^= other Update the set, keeping only elements found in either set, but not in both.

add(*elem*) Add element elem to the set.

remove(*elem*) Remove element elem from the set. Raises KeyError if elem is not contained in the set.

discard(*elem*) Remove element elem from the set if it is present.

pop() Remove and return an arbitrary element from the set. Raises KeyError if the set is empty.

clear() Remove all elements from the set.

Testing List Overlap

```
needles = {'i', 'l', '|', '1', '!'}
haystack = set(string.ascii_lowercase) | \
           set(string.punctuation) |\
           set(string.digits)

found = len(needles & haystack)

print(found)
```

Without sets

```
found = 0
for n in needles:
    if n in haystack:
        found += 1

print(found)
```

found = 5

Sets

```
s = {1, 2, 1, 4}
print(s)
```

{1, 2, 4}

```
s -= {2, 4, 6}
print(s)
```

{1}

```
y = s.pop()
print(f"y: {y}, s: {s}")
```

y: 1, s: set()

```
y = s <{0}
print(f"y: {y}, s: {s}")
```

y: True, s: set()

List to Set Conversion

Question: Can all lists be converted to a set?

Answer: No.

Can be converted to a set:
[1,2,'hello',True,(3,4,5),6.7]
because all elements are immutable.

Can **not** be converted to a set:
[1,2,'hello',True,[3,4,5],6.7]
because an element is mutable (nested list).

OOP

OBJECT ORIENTED PROGRAMMING

Object Oriented Programming

- OOP is a programming paradigm that structures code around objects, bundling data (attributes) and behavior (methods) into units called classes.
- **Key Concepts**
  - **Classes**: Blueprints that define attributes and behaviors; declarations of a new data type.
  - **Objects**: Instances of classes, representing real-world entities; instances of the class.
  - **Encapsulation**, **Inheritance**, **Polymorphism** and **Abstraction**
- **Benefits**
  - **Reusability**: Encourage reuse through classes and inheritance.
  - **Modularity**: Promotes organized and manageable code through class-based components.
  - **Extensibility**: Allow easy addition of new features through class extensions.
  - **Maintainability**: Simplifies maintenances and updates by structuring code.

Four Pillars of OOP

- **Encapsulation**: Encapsulation involves bundling data (attributes) and related methods within a class to control access and modification.
- **Inheritance**: Inheritance is the mechanism in which a class inherits attributes and methods from another class, fostering code reusability and creating a class hierarchy.
- **Polymorphism**: Polymorphism refers to the ability of objects to take on multiple forms, depending on the context. It allows methods with the same name to be implemented differently in different classes, promoting flexibility and dynamic behavior.
- **Abstraction**: Abstraction is the process of hiding the complex internal implementation of an object and exposing only the necessary features to the outside world. It focuses on what an object does without revealing how it achieves its functionality.

Terminology: Class

- Definition: "A blueprint for creating objects that defines their structure, behavior, and initial state”; the declaration of a new data type.
- Components:
  - **Attributes**: Data properties representing object characteristics.
  - **Methods**: Functions defining object behaviors and actions.

For example, the class Dog would consist of traits shared by all dogs, such as breed and fur color (characteristics), and the ability to bark and sit (behaviors).

Terminology: Instance

- **Definition**: "An object created from a class, representing a specific occurrence of that class."
- **Key Points:**
  - **Created from a Class:** An instance is an individual realization of a class, with specific attribute values and behaviors defined by the class.
  - **Instance State:** Each instance can have unique attribute values while adhering to the structure defined by the class.
  - **Multiple Instances:** Multiple instances of a class can be created, each independent of the others.

Object and Instance are often used interchangeably

Terminology: Method

- **Definition: "**Functions defined within a class, representing behaviors or actions that objects of that class can perform."
- **Characteristics:**
  - **Belong to a Class:** Methods are defined within a class and operate on the attributes of instances of that class.
  - **Access to Object State:** Methods can access and manipulate the object's state (attributes).
  - **Behavior Definition:** Methods *encapsulate* specific behavior associated with the class.

Some Python Classes

```
>>> s = 'abc'
>>> type(s)
<class 'str'>
>>> i = 5
>>> type(i)
<class 'int'>
>>> x = 2.5
>>> type(x)
<class 'float'>
>>> l = list()
>>> type(l)
<class 'list'>
>>> d = dict()
>>> type(d)
<class 'dict'>
```

CLASSES

METHODS

>>> dir(s)

['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

>>> dir(i)

['as_integer_ratio', 'bit_count', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']

>>> dir(l)

['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

>>> dir(d)

['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']

Some Python Classes

- We have actually encountered classes: All data types in Python are classes.
- All values (including functions, modules, and even classes themselves) are instances of those classes.
- That's why Python can treat everything uniformly and dynamically.
- We talked a little bit about "methods"-- functions "attached" to variables:
- We haven't created classes of our own yet though.
- This is what's coming next.

```
s = "hello world"
print(s.upper())      # 'HELLO WORLD'
print(s.lower())      # 'hello world'
print(s.capitalize()) # 'Hello world'
print(s.title())      # 'Hello World'
print(s.swapcase())   # 'HELLO WORLD' → 'hello world'
```

A Dog class

class **is a reserved keyword**

```
class Dog:
    def __init__(self,
                 name: str,
                 breed: str):
        self.name = name
        self.breed = breed

    def bark(self):
        return f"{self.name} says Woof!"
```

constructor

instance attributes

The bark() **is a method representing behavior**

```
# Creating an instance of the Dog class
my_dog = Dog("Tiger", "Labrador")

# Accessing attributes and methods of the instance
print(my_dog.name)  # Output: Tiger
print(my_dog.bark())  # Output: Tiger says Woof!
```

A sample Class from the book

```
class PartyAnimal:
    x = 0
    def party(self):
        self.x += 1
        print(f'So far: {self.x}')
```

class attribute

access the class attribute using self

"self" is a formal argument that refers to the object itself.

"self" is global within this object, meaning it is accessible by all the methods in the class.

```
pa = PartyAnimal()
pa.party()
pa.party()
pa.party()
```

Create an instance of the class PartyAnimal
call the method party()
three times.

So far: 1
So far: 2
So far: 3

Class Attributes vs Instance Attributes

```python
class MyClass:
    class_attr = 0    class attribute

    def __init__(self, instance_attr):
        self.instance_attr = instance_attr
                             instance attribute

print(f"MyClass.class_attr: {MyClass.class_attr}")

obj1 = MyClass(10)
obj2 = MyClass(20)

obj1.class_attr = 5

print(f"obj1.class_attr: {obj1.class_attr}")
print(f"obj2.class_attr: {obj2.class_attr}")

MyClass.class_attr = 50

print(f"MyClass.class_attr: {MyClass.class_attr}")
print(f"obj1.class_attr: {obj1.class_attr}")
print(f"obj2.class_attr: {obj2.class_attr}")
```

A **class attribute** is a variable that belongs to the class rather than instances of the class.
Accessed using the class name or instances of the class.
Modified by using the class name, affecting all instances and future instances.

An **instance attribute** is a variable specific to each instance of the class.
Accessed using the instance name.
Modified using the instance name, affecting only that particular instance.

https://pythontutor.com

Instance Attributes

```
class Dog:
    def __init__(self, name):
        self.name = name    # defined inside __init__

fido = Dog("Fido")

# Add a new instance attribute after creation
fido.age = 5
fido.breed = "Labrador"
```

name, age, and breed are all instance attributes.
Can be created and assigned via the constructor, via methods, or dynamically (for user-defined instances).

dir() and type()

```
>>> l = list()
>>> type(l)
<class 'list'>
>>> dir(l)
['__add__', '__class__', '__class_getitem__',
'__contains__', '__delattr__', '__delitem__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

- The dir() command lists capabilities.
- The methods starting with '_' are conventionally known as magic/dunder methods. You can implement magic methods for your classes to make them behave as Python objects.
- In contrast, methods that do not start with '_' are considered “regular” or “user defined” methods.
- The type() method prints the type of the object.

dir() **and** type()

```
class PartyAnimal:
    x = 0

    def party(self):
        self.x += 1
        print(f'So far: {self.x}')
```

- You can use dir() to find "capabilities" of our newly created class.
- You can implement the magic methods to make your class behave like Python objects.

```
pa = PartyAnimal()

print(type(pa))
<class '__main__.PartyAnimal'>
print(dir(pa))
```

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', **'party'**, **'x'**]

__dict__

- __dict__ is a **built-in attribute** that stores an object's **namespace** — i.e., all of its writable attributes and their current values.
- It's a dictionary that maps attribute names (as strings) to their values for that object.

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

fido = Dog("Fido", 5)

print(fido.__dict__)
```

Output:

```python
{'name': 'Fido', 'age': 5}
```

Object Lifecycle

- Objects are created, used, and discarded
- We have special blocks of code (methods) that get called
  - At the moment of creation (constructor)
  - At the moment of destruction (destructor)
- Constructors are used a lot
- Destructors are seldom used

Constructor

- In object-oriented programming, a *constructor* in a class is a special block of statements called when an object is created

- The primary purpose of the *constructor* is to set up some instance variables to have the proper initial values when the object is created

Many Instances

- We can create lots of *objects* - the class is the template for the object
- We can store each distinct *object* in its own variable
- We call this having multiple *instances* of the same class
- Each *instance* has its own copy of the **instance attributes**.
- Each *instance* shares the **class attributes**.