Introduction to Python

Objectives of this Lesson

1. Data Types - Sets: sets
2. File I/O
3. Import
4. Built-In Libraries
5. Variable Scope
6. Object Oriented Programming
7. Programming with OOP

Data Types - Sets: sets

A ***set*** object is an **unordered** collection of distinct hashable objects.

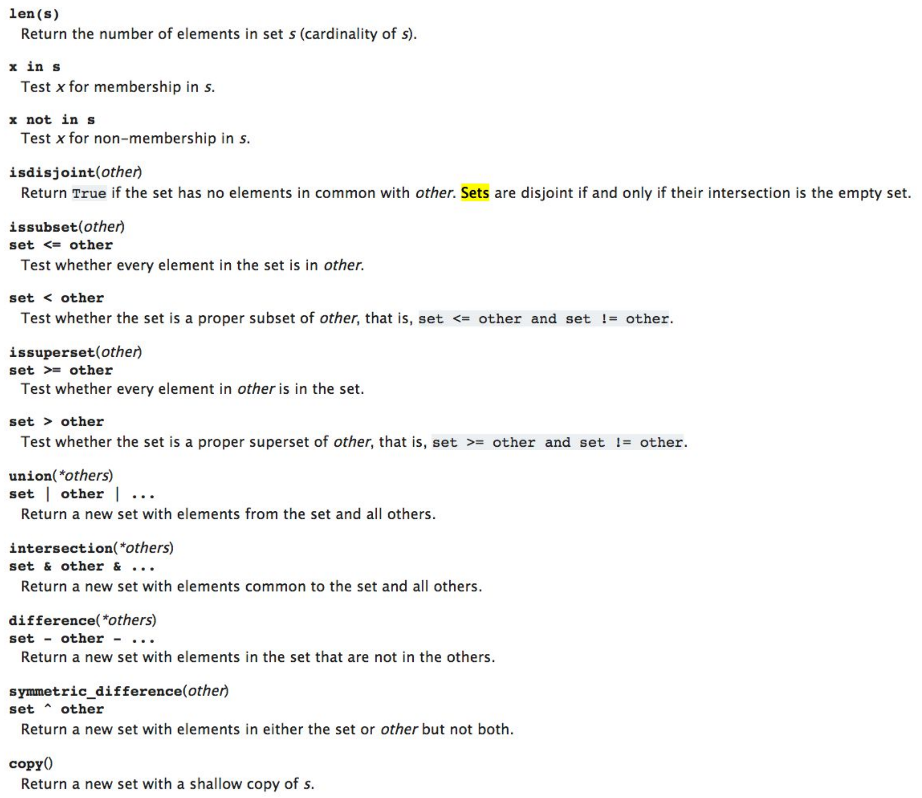
The set type is **mutable** — the contents can be changed using methods like add() and remove().

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

Common Uses: membership testing, removing duplicates from a sequence, computing mathematical operations such as intersection, union, difference, and symmetric difference.

Note: to create an empty set you have to use `set()`, not `{}`; To create a set of size one the value has to be followed by a comma: (a,).

Data Types - Sets: sets



`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`.

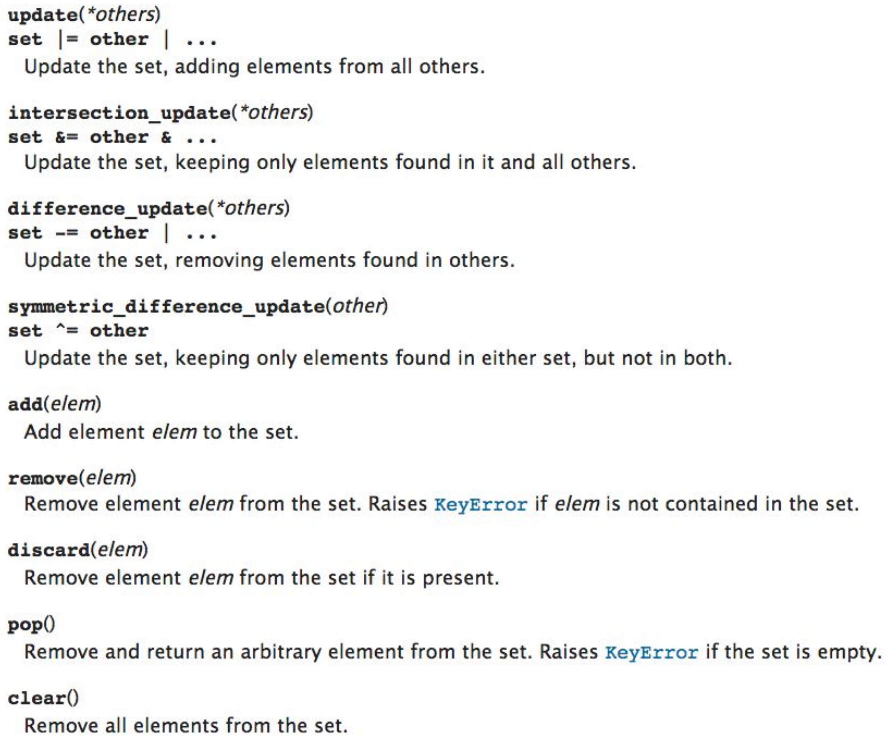
`union`(**others*)
`set | other | ...`
Return a new set with elements from the set and all others.

`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.

`copy`()
Return a new set with a shallow copy of *s*.



`update`(**others*)
`set |= other | ...`
Update the set, adding elements from all others.

`intersection_update`(**others*)
`set &= other & ...`
Update the set, keeping only elements found in it and all others.

`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.

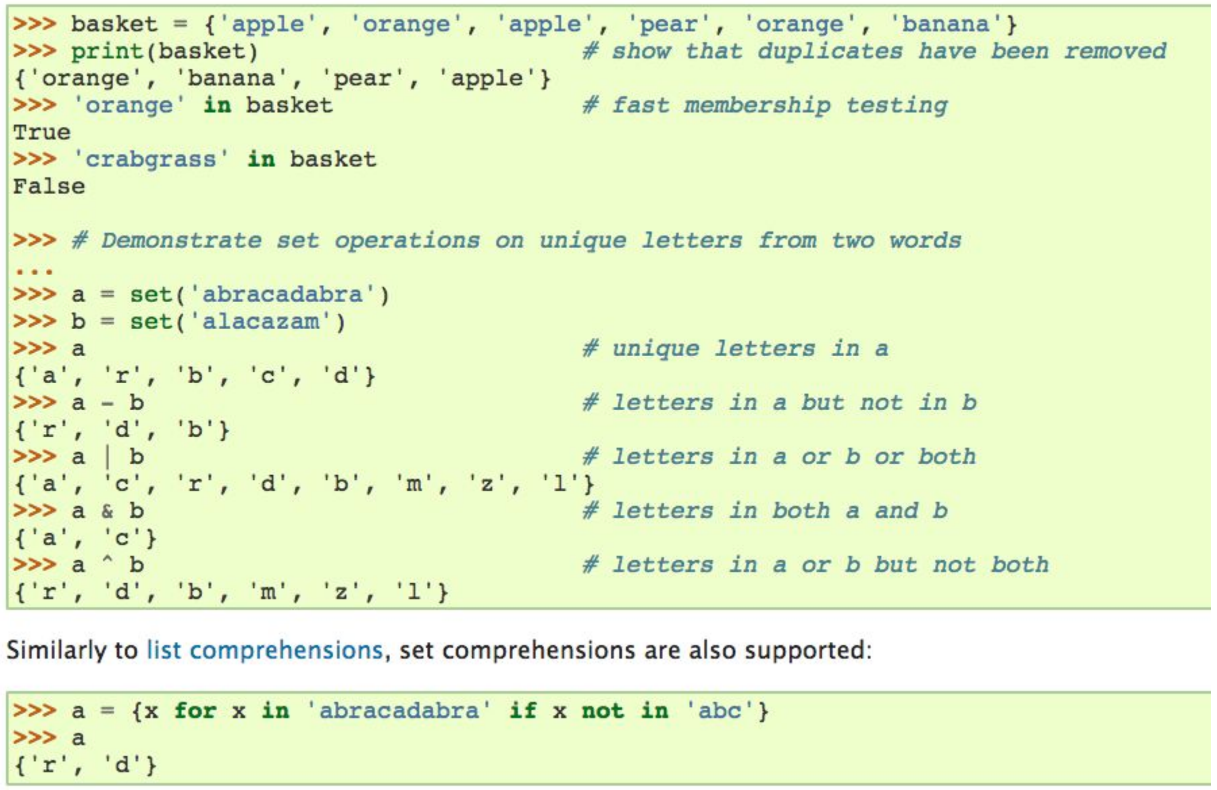
`remove`(*elem*)
Remove element *elem* from the set. Raises `KeyError` if *elem* is not contained in the set.

`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.

Data Types - Sets: sets



```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Similarly to list comprehensions, set comprehensions are also supported:

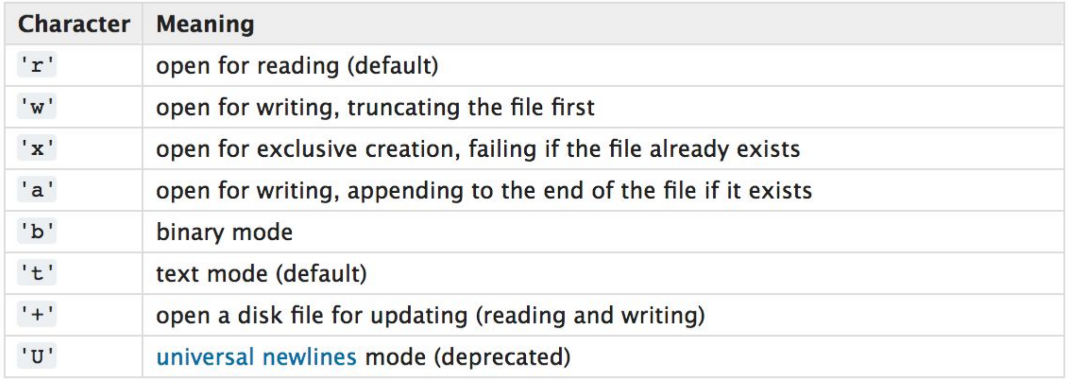
```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

File I/O

Before you can read or write a file, you have to open it using Python's built-in *open()* function.

This function creates a **file** object, which would be utilized to call other support methods associated with it.
It is most commonly used with two arguments: `open(filename, mode)`.

> *filename* is a path-like object giving the pathname (absolute or relative to the current working directory) of the file to be opened.
>
> *mode* is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode.



Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'x'`	open for exclusive creation, failing if the file already exists
`'a'`	open for writing, appending to the end of the file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open a disk file for updating (reading and writing)
`'U'`	universal newlines mode (deprecated)

File I/O

- To read a file’s contents, call *fileobject*.read(size), which reads some quantity of data and returns it as a string .
  - *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned.

- *fileobject*.readline() reads a single line from the file.

- For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in fileobject:
...         print(line)
...
This is the first line of the file.
Second line of the file
```

File I/O

- *fileobject*.seek(offset, reference point) moves to the position that is computed from adding *offset* to a *reference point*.
    - A *reference point* of 0 measures from the beginning of the file
    - 1 uses the current file position
    - 2 uses the end of the file as the reference point
    - r*eference point* can be omitted and defaults to 0.

- *fileobject*.write(**string**) writes the contents of ***string*** to the file, returning the number of characters written.

File I/O

- When you want to save more complex data types like nested lists and dictionaries Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation). The standard module called json can take Python data hierarchies, and convert them to string representations.
  - This process is called *serializing*.
  - Reconstructing the data from the string representation is called *deserializing*.

- If you have an object x, you can view its JSON string representation with a simple line of code:

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

- Another variant of the dumps() function, called dump(), simply serializes the object to a text file.
  So if f is a text file object opened for writing, we can do this:

```
json.dump(x, f)
```

- To decode the object again, if f is a text file object which has been opened for reading:

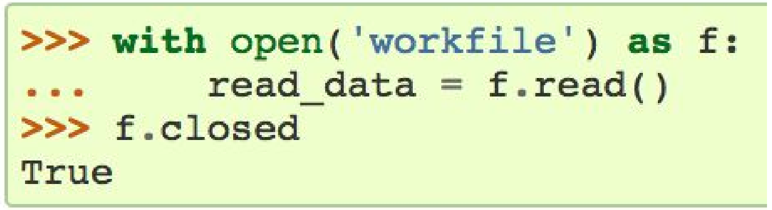
```
x = json.load(f)
```

- This simple serialization technique can handle lists and dictionaries.

File I/O

- Call f.close() to close the file and free up any system resources used by it.
  - If you don’t explicitly close a file, Python’s garbage collector will eventually destroy the object and close the open file for you, but the file may stay open for a while.

- It is good practice to use the with keyword when dealing with file objects.
  - The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point.



```
>>> with open('workfile') as f:
...     read_data = f.read()
>>> f.closed
True
```

Import Statement

- Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter.

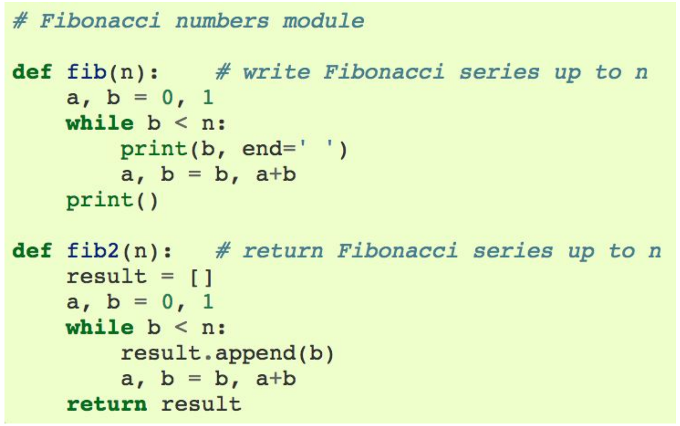
- Such a file is called a ***module***; definitions from a module can be ***imported*** into other modules.

- A module can contain executable statements as well as function definitions.
  - These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement.(They are also run if the file is executed as a script.)

- It is customary but not required to place all import statements at the beginning of a module (or script, for that matter).

Import Statement

“fibo.py”



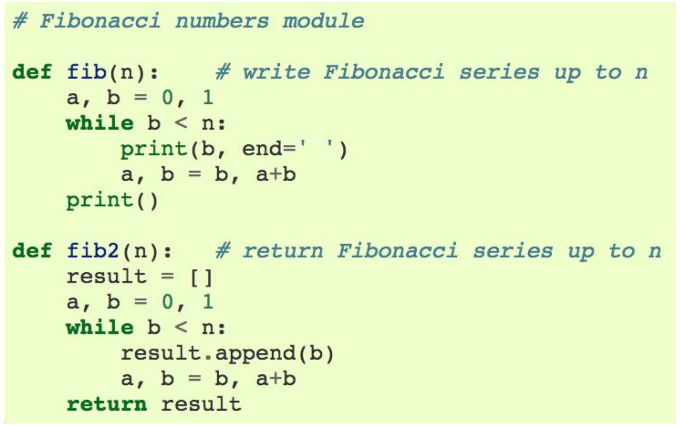
```python
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Using the module name you can now access the functions:

```
>>> import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Import Statement

“fibo.py”



```
# Fibonacci numbers module

def fib(n):     # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n):    # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Here is a variant of the import statement that imports names from a module directly. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken (so in the example, fibo is not defined).

Import Statement

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility.

Note that in general the practice of importing `*` from a module or package is frowned upon.

Import Statement

When you run a Python module with

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`.
That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

Import Statement

The built-in function dir() is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo
>>> dir(fibo)
['__name__', 'fib', 'fib2']
```

Without arguments, dir() lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Built-In Libraries - datetime

A datetime object is a single object containing all the information from a date object and a time object.

Constructor:

classmethod datetime.**datetime**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

The year, month and day arguments are required.

classmethod datetime.**now**(*tz=None*)

Return the current local date and time.

Instance methods:

datetime.**date**()

Return date object with same year, month and day.

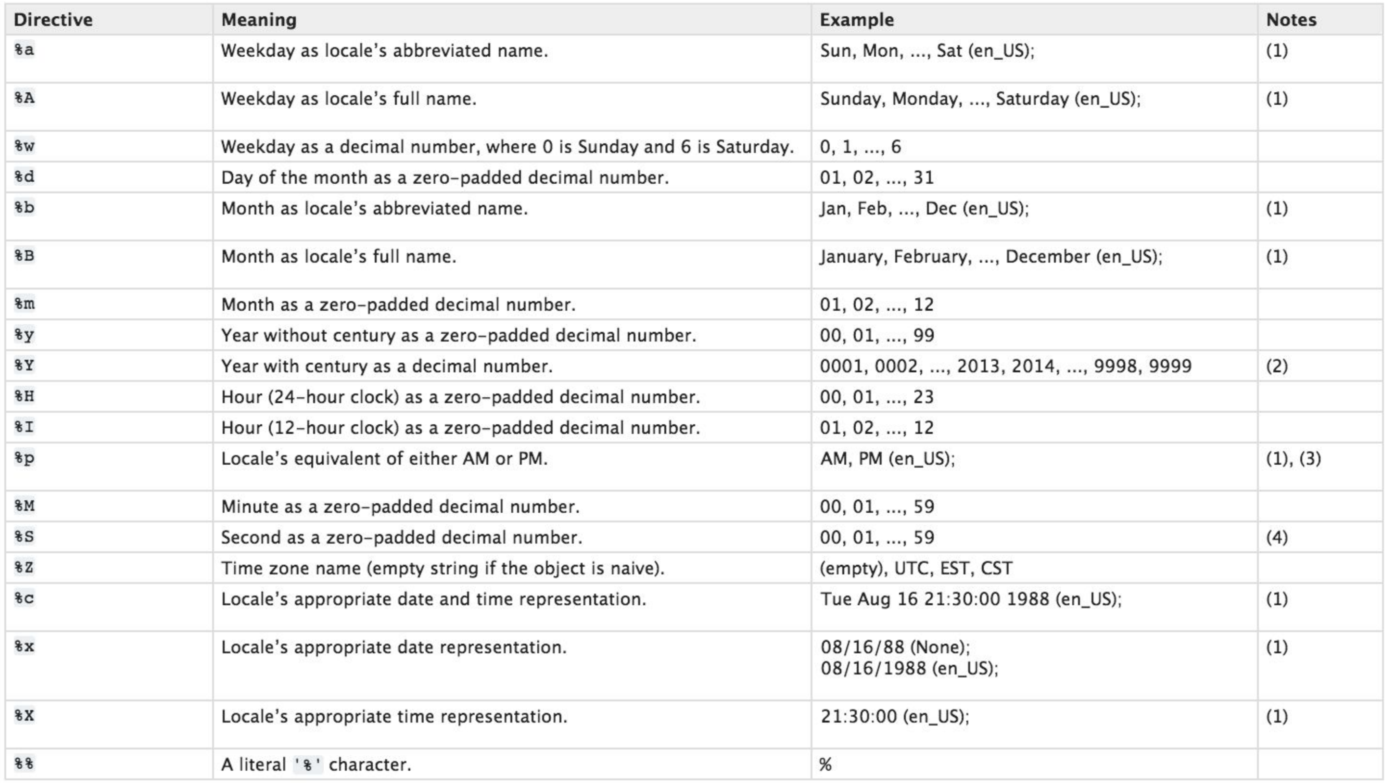
datetime.**time**()

Return time object with same hour, minute, second, microsecond and fold.

datetime.**strftime**(*format*)

Return a string representing the date and time

Built-In Libraries - datetime



Directive	Meaning	Example	Notes
`%a`	Weekday as locale's abbreviated name.	Sun, Mon, ..., Sat (en_US);	(1)
`%A`	Weekday as locale's full name.	Sunday, Monday, ..., Saturday (en_US);	(1)
`%w`	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, ..., 6	
`%d`	Day of the month as a zero-padded decimal number.	01, 02, ..., 31	
`%b`	Month as locale's abbreviated name.	Jan, Feb, ..., Dec (en_US);	(1)
`%B`	Month as locale's full name.	January, February, ..., December (en_US);	(1)
`%m`	Month as a zero-padded decimal number.	01, 02, ..., 12	
`%y`	Year without century as a zero-padded decimal number.	00, 01, ..., 99	
`%Y`	Year with century as a decimal number.	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(2)
`%H`	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23	
`%I`	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12	
`%p`	Locale's equivalent of either AM or PM.	AM, PM (en_US);	(1), (3)
`%M`	Minute as a zero-padded decimal number.	00, 01, ..., 59	
`%S`	Second as a zero-padded decimal number.	00, 01, ..., 59	(4)
`%Z`	Time zone name (empty string if the object is naive).	(empty), UTC, EST, CST	
`%c`	Locale's appropriate date and time representation.	Tue Aug 16 21:30:00 1988 (en_US);	(1)

`%x`	Locale's appropriate date representation.	08/16/88 (None);
		08/16/1988 (en_US);
			(1)

`%X`	Locale's appropriate time representation.	21:30:00 (en_US);	(1)
`%%`	A literal `'%'` character.	%	

Built-In Libraries - datetime

```
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)

>>> # Formatting datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
```

Built-In Libraries - random

random.**randrange**(*start*, *stop*[, *step*])

Return a randomly selected element from `range(start, stop, step)`.

random.**choice**(*seq*)

Return a random element from the non-empty sequence *seq*.

random.**choices**(*population*, *weights=None*, *k=1*)

Return a *k* sized list of elements chosen from the *population* with replacement.
If a *weights* sequence is specified, selections are made according to the relative weights.

random.**shuffle**(*x*)

Shuffle the sequence *x* in place.
To shuffle an immutable sequence and return a new shuffled list, use `sample(x, k=len(x))` instead.

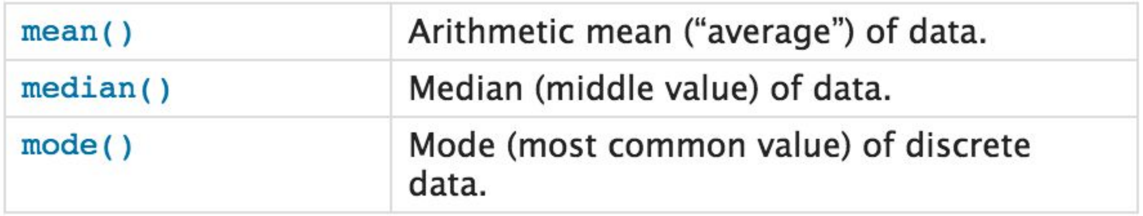
random.**sample**(*population*, *k*)

Return a *k* length list of unique elements chosen from the population sequence or set. Used for random sampling without replacement.

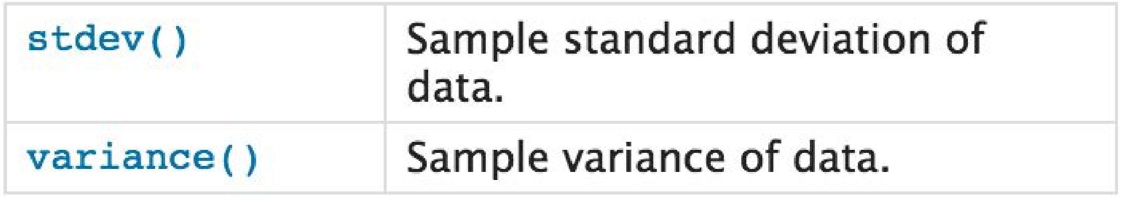
Built-In Libraries - random

```
>>> randrange(10)                              # Integer from 0 to 9 inclusive
7

>>> choice(['win', 'lose', 'draw'])            # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()        # Creates a list [‘ace’, ‘two’, ‘three’, ‘four’]
>>> shuffle(deck)                              # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4)          # Four samples without replacement
[40, 10, 50, 30]
```

Built-In Libraries - statistics



`mean()`	Arithmetic mean (“average”) of data.
`median()`	Median (middle value) of data.
`mode()`	Mode (most common value) of discrete data.



`stdev()`	Sample standard deviation of data.
`variance()`	Sample variance of data.

Built-In Libraries - csv

The CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases.

csv.**reader**(*csvfile*)

Return a reader object which will iterate over lines in the given *csvfile*.

csv.**writer**(*csvfile*)

Return a writer object responsible for converting the user's data into delimited strings on the given file-like object.

class csv.**DictReader**(*f, fieldnames*)

Create an object that operates like a regular reader but maps the information in each row to an OrderedDict whose keys are given by the optional *fieldnames* parameter.

class csv.**DictWriter**(*f, fieldnames*)

Create an object which operates like a regular writer but maps dictionaries onto output rows.The *fieldnames* parameter is a sequence of keys that identify the order in which values in the dictionary passed to the `writerow()` method are written to file *f*.

Built-In Libraries - csv

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

```
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
```

Output:

```
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

Built-In Libraries - csv

```python
import csv

with open('names.csv', 'w') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Eric', 'last_name': 'Idle'})
    writer.writerow({'first_name': 'John', 'last_name': 'Cleese'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

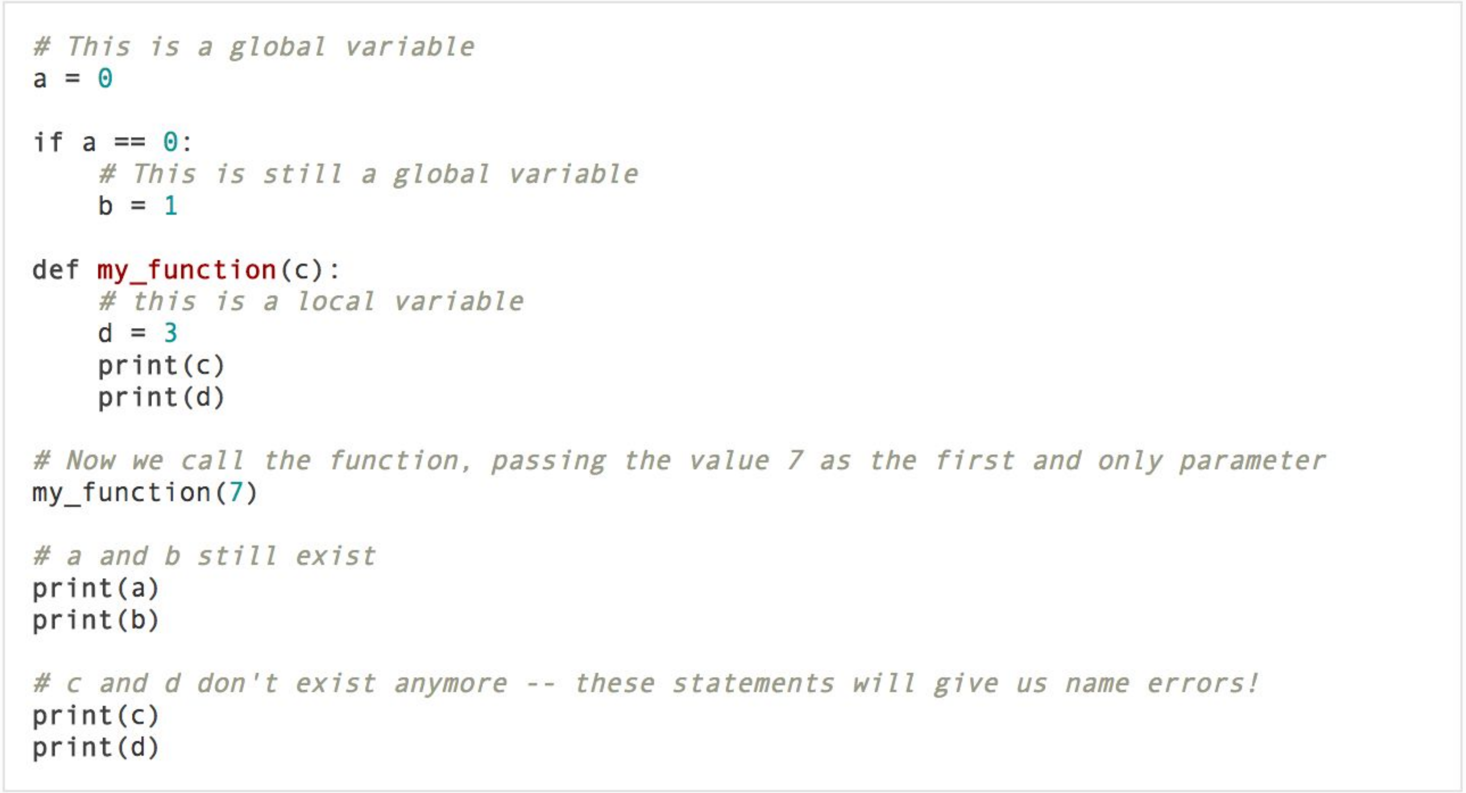
```
>>> with open('names.csv') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...

Eric Idle
John Cleese

>>> print(row)

OrderedDict([('first_name', 'John'), ('last_name', 'Cleese')])
```

Variable Scope

- A variable which is defined in the main body of a file is called a **global variable**.
    - It will be visible throughout the file, and also inside any file which imports that file.

- A variable which is defined inside a function is **local to that function**.
    - When we use the assignment operator (`=`) inside a function, it creates a new local variable.
    - It is accessible from the point at which it is defined until the end of the function.
    - The parameter names in the function definition behave like local variables.

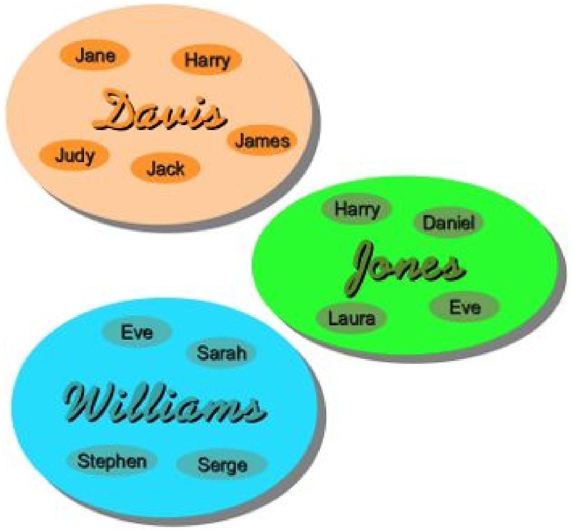
Variable Scope



```
# This is a global variable
a = 0

if a == 0:
    # This is still a global variable
    b = 1

def my_function(c):
    # this is a local variable
    d = 3
    print(c)
    print(d)

# Now we call the function, passing the value 7 as the first and only parameter
my_function(7)

# a and b still exist
print(a)
print(b)

# c and d don't exist anymore -- these statements will give us name errors!
print(c)
print(d)
```

Variable Scope

- *A* **namespace** is a mapping from names to objects.
- Generally speaking, a namespace is a naming system for making names unique to avoid ambiguity.
  - A namespacing system from daily life - the naming of people in firstname and surname.



Variable Scope

- Many programming languages use namespaces for **identifiers**.
    - An identifier defined in a namespace is associated with that namespace. This way, the same identifier can be independently defined in multiple namespaces.

- Examples of namespaces:
    - the set of built-in names
    - the global names in a module
    - the local names in a function invocation
    - In a sense the set of attributes of an object also form a namespace

- The important thing to know about namespaces is that there is **absolutely no relation between names in different namespaces**
    - for instance, two different modules may both define a function maximize without confusion — users modules must prefix it with the module name.

Variable Scope

Lifetime of a Namespace

- Not every namespace is accessible (or alive) at any moment during the execution of the script.
- There is one namespace which is present from beginning to end: The namespace containing the built-in names.
- The global namespace of a module is generated when the module is read in.
- Module namespaces normally last until the script ends.
- When a function is called, a local namespace is created for this function. This namespace is deleted either if the function ends, i.e. returns, or if the function raises an exception, which is not dealt with within the function.

Variable Scope

Scopes

The scope of a namespace is the area of a program where this name can be unambiguously used, for example inside of a function.

At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:

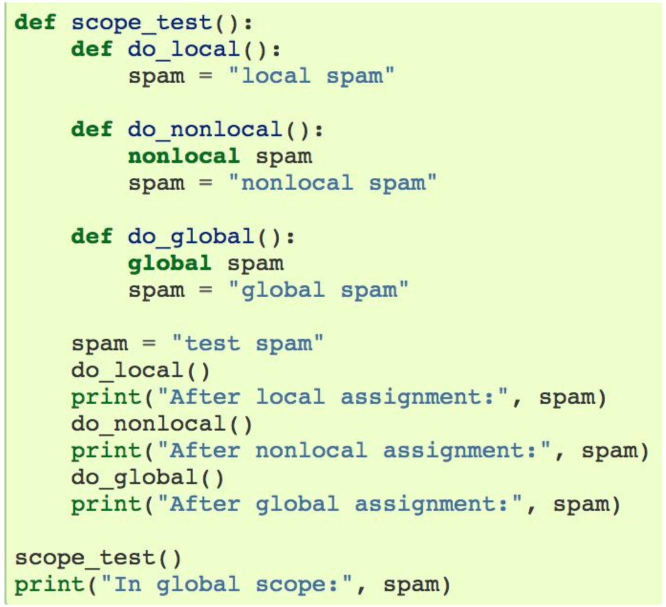
- the innermost scope, which is searched first, contains the **local names**
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains **non-local, but also non-global names**
- the next-to-last scope contains the current module’s **global names**
- the outermost scope (searched last) is the namespace containing **built-in names**

Variable Scope

- If a name is declared **global**, then all references and assignments go directly to the middle scope containing the module’s global names.
- To rebind variables found outside of the innermost scope, the **nonlocal** statement can be used
  - if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).
- Usually, the local scope references the local names of the current function.
  - Outside functions, the local scope references the same namespace as the global scope: the module’s namespace.
  - Class definitions place yet another namespace in the local scope.

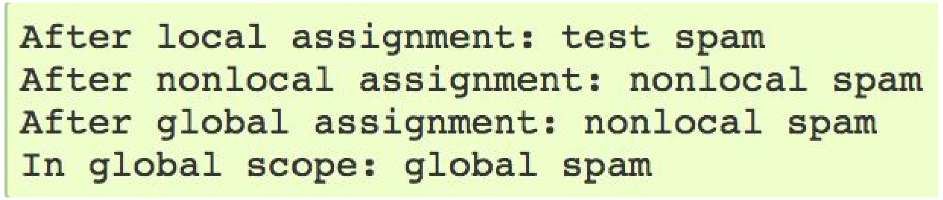
Variable Scope

This is an example demonstrating how to reference the different scopes and namespaces, and how global and nonlocal effect variable binding:



```python
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Variable Scope



```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Note how the *local* assignment (which is default) didn’t change *scope_test*‘s binding of *spam*.
The nonlocal assignment changed *scope_test*‘s binding of *spam*
The global assignment changed the module-level binding.
You can also see that there was no previous binding for *spam* before the global assignment.

Object Oriented Programming

Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data. The programming challenge was seen as how to write the logic, not how to define the data.

Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them.

Examples of objects range from human beings (described by name, address, and so forth) to buildings and floors (whose properties can be described and managed) down to the little widgets on a computer desktop (such as buttons and scrollbars).

OOP is a programming paradigm based on the concept of "**objects**", which may contain:

- data, often known as **attributes**
- and code, often known as **method**

Object Oriented Programming

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

- **Class variable:** A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.

- **Method :** A special kind of function that is defined in a class definition.

- **Instantiation:** The creation of an instance of a class.

- **Instance:** An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.

- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.

- **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it.

- **Object:** A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Object Oriented Programming - Class Definition

The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Class definitions, like function definitions must be executed before they have any effect.

In practice, the statements inside a class definition will usually be function definitions.

Object Oriented Programming - Attribute References

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`.

So, if the class definition looked like this:

```python
class MyClass:
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively.

Class attributes can also be assigned to, so you can change the value of `MyClass.i`

Object Oriented Programming - Class Instantiation

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

The instantiation operation ("calling" a class object) creates an empty object.

Object Oriented Programming - Class Instantiation

Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named __init__(), like this:

```
def __init__(self):
    self.data = []
```

When a class defines an __init__() method, class instantiation automatically invokes __init__() for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the __init__() method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to __init__(). For example,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Object Oriented Programming - Instance Objects

The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

data attributes need not be declared; like local variables, they spring into existence when they are first assigned to.

For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value `16`, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

Object Oriented Programming - Instance Objects

The other kind of instance attribute reference is a *method*.

A method is a function that “belongs to” an object.

Usually, a method is called right after it is bound:

```
x.f()
```

In the MyClass example, this will return the string `'hello world'`.

However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print(xf())
```

will continue to print `hello world` until the end of time.

Object Oriented Programming - Instance Objects

You may have noticed that `x.f()` was called without an argument above, even though the function definition for f() specified an argument. What happened to the argument? .

The special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`.

Object Oriented Programming

```
class Dog:

    kind = 'canine'                  # class variable shared by all instances

    def __init__(self, name):
        self.name = name             # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                           # shared by all dogs
'canine'
>>> e.kind                           # shared by all dogs
'canine'
>>> d.name                            # unique to d
'Fido'
>>> e.name                           # unique to e
```

Object Oriented Programming

```
class Dog:

    kind = 'canine'                     # class variable shared by all instances

    def __init__(self, name):
        self.name = name                # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                              # shared by all dogs
'canine'
>>> e.kind                              # shared by all dogs
'canine'
>>> d.name                               # unique to d
'Fido'
>>> e.name                              # unique to e
```

Object Oriented Programming

```
class Dog:

    tricks = []                                   # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                                      # unexpectedly shared by all dogs
['roll over', 'play dead']
```

The *tricks* list in the following code should not be used as a class variable because just a single list would be shared by all *Dog* instances.

Object Oriented Programming

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

Object Oriented Programming - Inheritance

The syntax for inheritance definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    <statement-N>
```

When the class object is constructed, the base class is remembered.
This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class.
There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class.

Derived classes may override methods of their base classes.
There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`.

Python has two built-in functions that work with inheritance:

- Use isinstance() to check an instance's type: `isinstance(obj, int)` will be `True` only if `obj.__class__` is int or some class derived from int.
- Use issubclass() to check class inheritance: `issubclass(bool, int)` is `True` since bool is a subclass of int. However, `issubclass(float,int)` is `False` since float is not a subclass of int.

Object Oriented Programming

```
class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr
```

```
class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

Object Oriented Programming

```
class Parent:        # define parent class
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent): # define child class
   def myMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```