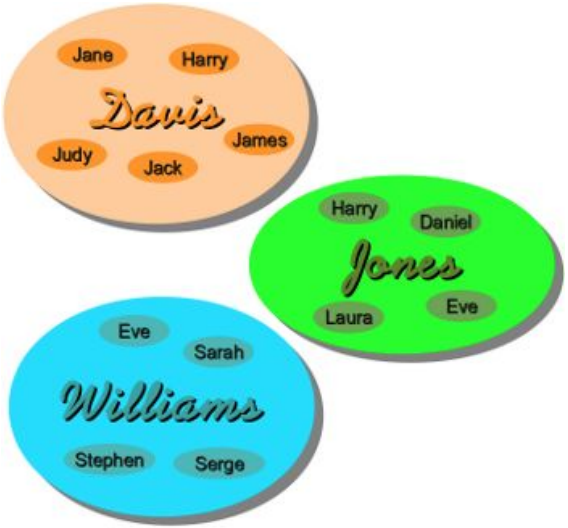
Introduction to Python

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**.
  - In a single Python program there are multiple namespaces.
  - An identifier defined in a particular namespace is **bound**/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

- **There is absolutely no relation between names in different namespaces**

Variable Scope

Lifetime of a Namespace

Not every namespace is accessible (or alive) at any moment during the execution of the script.

- The **built-in namespace** is present from beginning to end.
- The **global namespace** of a module is generated when the module is read in.
- The **module namespaces** normally last from when it is imported until the script ends.
- The **local namespace of a particular function** is created when the function is called. The namespace is destroyed either if the function ends, i.e. returns, or if the function raises an exception.

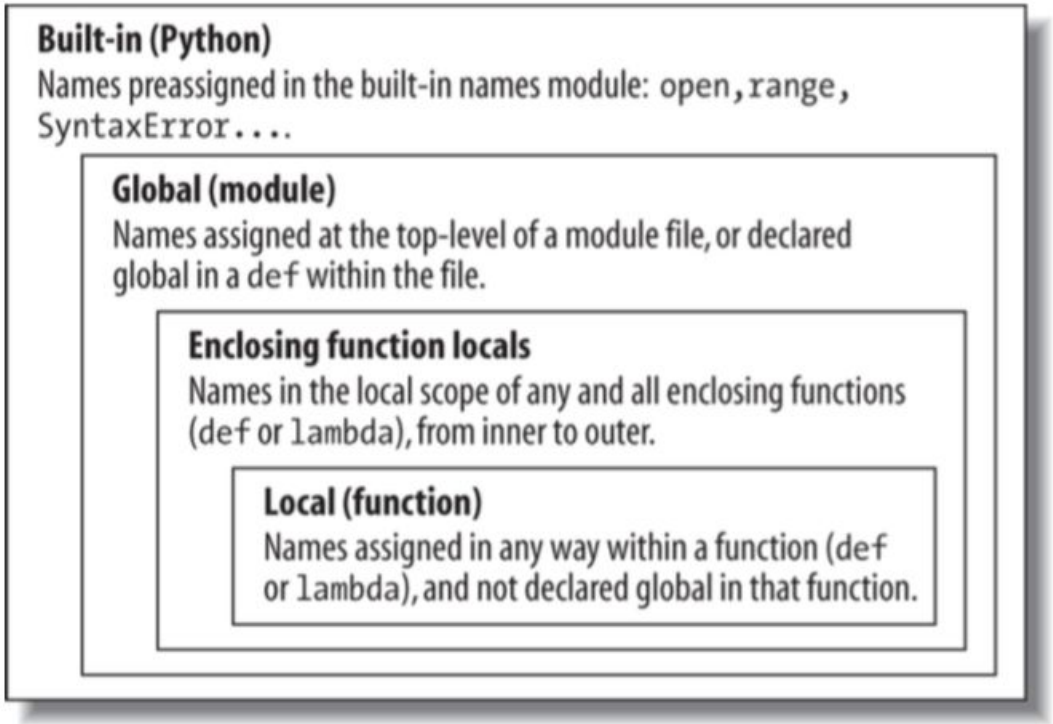
Variable Scope

Scopes
The scope of a namespace is the area of a program where this name can be unambiguously used.

At any time during execution, there are at least three nested scopes whose namespaces are directly accessible. Python works its way out to resolve which scope you are referring to.

- the innermost scope i.e. the current and alive function, 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 is searched next and it contains the current module's **global names**
- the outermost scope, which is searched last, is the namespace containing **built-in names.**

Variable Scope



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

- 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).

Variable Scope

```
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)
```

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.

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

Functions

Recursion

Sometimes it is difficult to define an object explicitly.

However,
It may be easy to define this object in terms of itself.

Functions

Recursion

Termination Condition:

A recursive function has to **terminate** to be correct and usable.
A recursive function terminates, if with every recursive call the solution of the problem is downsized (becomes easier) and moves towards a **base case**.

Base Case:

A **base case** is a case, where the problem can be solved without further recursion.
A recursion can lead to an infinite loop, if the base case is not met in the calls.

Python implements recursion by pushing information on a call stack at each recursive call, so it remembers where it must return and continue later.

Functions

Using Recursion To Calculate Factorials:

procedure *factorial*(n: nonnegative integer)
if $n = 0$ **then return** 1
else return $n \cdot factorial(n - 1)$
{output is $n!$}

4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1

Replacing the calculated values gives us the following expression
4! = 4 * 3 * 2 * 1

Functions

```
def factorial(n):

    print("factorial has been called with n = " + str(n))

    if n == 1:
        return 1

    else:
        res = n * factorial(n-1)

        print("intermediate result for ", n, " * factorial(" ,n-1, "): ",res)

        return res

print(factorial(5))
```

Output

```
factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1

intermediate result for  2  * factorial( 1 ):  2
intermediate result for  3  * factorial( 2 ):  6
intermediate result for  4  * factorial( 3 ):  24
intermediate result for  5  * factorial( 4 ):  120

120
```

Functions

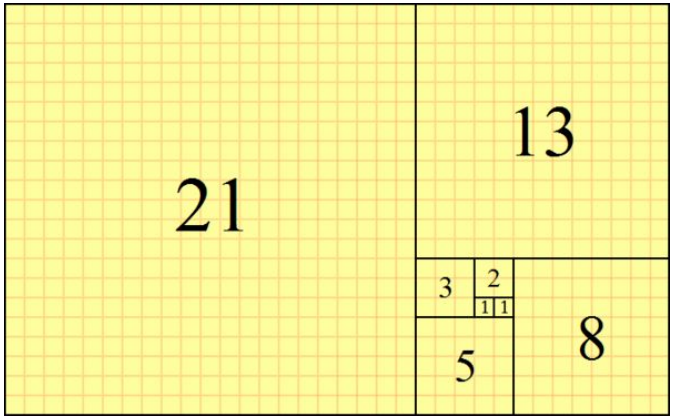
Recursion to Sum the Elements of a List:

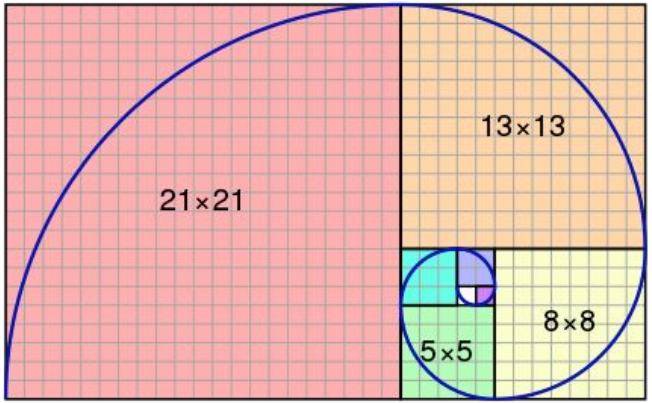
```
>>> def mysum(L):
        print(L)                        # Trace recursive levels
        if not L:                       # L shorter at each level
            return 0
        else:
            return L[0] + mysum(L[1:])

>>> mysum([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
[]
15
```

Fibonacci

The **Fibonacci numbers** are the numbers characterized by the fact that every number after the first two is the sum of the two preceding ones. The sequence of Fibonacci numbers appears like this:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144





Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

Task: Given an integer as input, determine the fibonacci sequence of that length. Use recursion.

i.e.	fib(1)	returns 0
	fib(5)	returns 0,1,1,2,3
	fib(13)	returns 0,1,1,2,3,5,8,13,34,55,89,144

Fibonacci

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def fib_seq(n):
    for i in range(n):
        if i == n-1:
            print(fib(i))
        else:
            print(fib(i), end =', ')

def main():
    n = int(input('Enter A Number: '))
    fib_seq(n)

main()
```

Functions

First Class Functions

Everything in Python is an object.
This includes functions.
This model is called the *first class object model*.

This means:

- We can assign a function to a variable.
- We can pass a function to another function as an argument.
- We can return a function from another function.
- Etc.

Functions

First Class Functions

```
>>> def echo(message):                  # Name echo assigned to function object
        print(message)

>>> echo('Direct call')                 # Call object through original name
Direct call

>>> x = echo                            # Now x references the function too
>>> x('Indirect call!')                 # Call object through name by adding ()
Indirect call!

>>> def indirect(func, arg):
        func(arg)                       # Call the passed-in object by adding ()

>>> indirect(echo, 'Argument call!')    # Pass the function to another function
Argument call!
```

Functions

First Class Functions

```
>>> def make(label):                    # Make a function but don't call it
        def echo(message):
            print(label + ':' + message)
        return echo

>>> F = make('Spam')                    # Label in enclosing scope is retained
>>> F('Ham!')                           # Call the function that make returned
Spam:Ham!
>>> F('Eggs!')
Spam:Eggs!
```

Functions

Lambda

Sometimes called **anonymous functions**.

Like def, this expression creates a function, but it returns the function instead of assigning it to a name.

The lambda's general form is the keyword lambda, followed by one or more arguments followed by an expression after a colon:

lambda *argument1, argument2,... argumentN* : *expression using arguments*

```
>>> def func(x, y, z): return x + y + z

>>> func(2, 3, 4)
9
```

```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)
9
```

Functions

Lambda

Lambda is an expression, not a statement

- This means lambda can appear in places a def statement cannot.
  - Inside a list literal, as a function argument,

- Lambda is meant for designing simple functions.
- Complex functions should be written with a def statement.

Functions

Lambda

Lambda can be used for these reasons but are entirely optional:

- Write a simple function on a single line
- Jump tables

```
L = [lambda x: x ** 2,          # Inline function definition
     lambda x: x ** 3,
     lambda x: x ** 4]          # A list of three callable functions

for f in L:
    print(f(2))                 # Prints 4, 8, 16

print(L[0](3))                  # Prints 9
```

```
def f1(x): return x ** 2
def f2(x): return x ** 3        # Define named functions
def f3(x): return x ** 4

L = [f1, f2, f3]                # Reference by name

for f in L:
    print(f(2))                 # Prints 4, 8, 16

print(L[0](3))                  # Prints 9
```

```
>>> key = 'got'
>>> {'already': (lambda: 2 + 2),
     'got':     (lambda: 2 * 4),
     'one':     (lambda: 2 ** 6)}[key]()
```

```
>>> def f1(): return 2 + 2

>>> def f2(): return 2 * 4

>>> def f3(): return 2 ** 6

>>> key = 'one'
>>> {'already': f1, 'got': f2, 'one': f3}[key]()
```

Functions

Generator Functions

Generators produce results only when needed.
This saves memory space and only computes when requested.

Coded like normal **def statements** but use the **yield statement** instead of **return statement**.
Generators implement the **iteration protocol**.

Functions

Generator Functions

Yield Statement

- Yield suspends and saves the state of a function so that the function can be resumed at a later time.
- It also sends one value back to the caller.
- The yield statement causes the function to adopt the iteration protocol.
  - So the next() method resumes the function.
  - And we can use a for loop to iterate over the generator.

Functions

Generator Functions

```
>>> def gensquares(N):
        for i in range(N):
            yield i ** 2

>>> x = gensquares(4)
```

```
>>> next(x)                    # Same as x.__next__() in 3.X
0
>>> next(x)                    # Use x.next() or next() in 2.X
1
>>> next(x)
4
>>> next(x)
9
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
>>> for i in gensquares(5):
        print(i, end=' : ')

0 : 1 : 4 : 9 : 16 :
>>>
```

Functions

Decorator Functions

A decorator augments a function's definition by wrapping it in an extra layer of logic.

Syntactically:

- A function decorator is a sort of runtime declaration about the function that follows.
- A function decorator is coded on a line by itself just before the def statement that defines a function or method.
- It consists of the @ symbol, followed by what we call a **metafunction** - a function that manages another function.

Functions

Decorator Functions

```
def my_decorator(some_function):

    def wrapper():

        print("Something is happening before some_function() is called.")

        some_function()

        print("Something is happening after some_function() is called.")

    return wrapper


def just_some_function():
    print("Wheee!")


just_some_function = my_decorator(just_some_function)

just_some_function()
```

```
Something is happening before some_function() is called.
Wheee!
Something is happening after some_function() is called.
```

Functions

Decorator Functions

```
@decorator              # Function decoration syntax
    def method():
```

Is equivalent to

```
def method()
method = decorator(method)
```

Functions

Decorator Functions

```
>>> def decorater(function):
...     def wrapper(string):
...             return '***{}***'.format(function(string))
...     return wrapper
>>> @decorater
... def fun(word):
...     return 'how now brown {}'.format(word)
...
>>> print(fun('donkey'))
>>> ***how now brown donkey***
```

Functions

Decorator Functions With Arguments

```
def tags(tag_name):
    def tags_decorator(func):
        def func_wrapper(name):
            return "<{0}>{1}</{0}>".format(tag_name, func(name))
        return func_wrapper
    return tags_decorator

@tags("p")
def get_text(name):
    return "Hello "+name

print get_text("John")

# Outputs <p>Hello John</p>
```

Python Conceptual Hierarchy

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. *Expressions create and process objects.*

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.
- 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”

```
# 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 underscore (_) facility.

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:

class 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'

 >>> dt.strftime("%x %X")

>>> datetime.datetime.now()
```

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()
>>> 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))
```

Built-In Libraries - csv

```
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'})
```

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

Object Oriented Programming

Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data (functional programming). 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:

```
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`.

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:

    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
```

Objectives of this Lesson

1. Installing Third Party Libraries - pip
2. NumPy
3. MatPlotLib
4. Pandas

pip[3]

pip is a package management system used to install and manage software packages written in Python. Many packages can be found in the Python Package Index (PyPI).

Installation

Do I need to install pip?

pip is already installed if you're using Python 2 >=2.7.9 or Python 3 >=3.4 binaries downloaded from python.org, but you'll need to upgrade pip.

Additionally, pip will already be installed if you're working in a Virtual Environment created by virtualenv or pyvenv.

Installing with get-pip.py

To install pip, securely download get-pip.py. [2]

Then run the following:

pip[3]

Upgrading pip

On Linux or macOS:

```
pip install -U pip
```

On Windows [5]:

```
python -m pip install -U pip
```

pip[3]

```
$ pip install SomePackage
  [...]
  Successfully installed SomePackage
```

If installation fails you may have to run pip as an administrator.

NumPy

NumPy is an acronym for "Numeric Python" or "Numerical Python".

It is an open source extension module for Python, which provides fast precompiled functions for mathematical and numerical routines.

Furthermore, NumPy enriches the programming language Python with powerful data structures for efficient computation of multi-dimensional arrays and matrices.

The most powerful feature of NumPy is n-dimensional array. This library also contains basic linear algebra functions, Fourier transforms, advanced random number capabilities and tools for integration with other low level languages like Fortran, C and C++.

NumPy

NumPy's main object is the **homogeneous multidimensional array**.

It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.

In NumPy dimensions are called *axes*. The number of axes is *rank*.

For example, the coordinates of a point in 3D space [1, 2, 1] is an array of rank 1, because it has one axis. That axis has a length of 3.

In the example below, the array has rank 2 (it is 2-dimensional). The first dimension (axis) has a length of 2, the second dimension has a length of 3.

```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

NumPy

NumPy's array class is called **ndarray**.

Note that **numpy.array** is not the same as the Standard Python Library class **array.array**, which only handles one-dimensional arrays and offers less functionality.

The more important attributes of an ndarray object are:

ndarray.ndim

the number of axes (dimensions) of the array. In the Python world, the number of dimensions is referred to as *rank*.

ndarray.shape

the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with *n* rows and *m* columns, `shape` will be `(n,m)`. The length of the `shape` tuple is therefore the rank, or number of dimensions, `ndim`.

ndarray.size

the total number of elements of the array. This is equal to the product of the elements of `shape`.

ndarray.dtype

an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

ndarray.itemsize

the size in bytes of each element of the array. For example, an array of elements of type `float64` has `itemsize` 8 (=64/8), while one of type `complex32` has `itemsize` 4 (=32/8). It is equivalent to `ndarray.dtype.itemsize`.

ndarray.data

the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

NumPy

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<type 'numpy.ndarray'>
```

NumPy

You can create an array from a regular Python list or tuple using the `array` function.

```
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')

>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

A frequent error consists in calling `array` with multiple numeric arguments, rather than providing a single list of numbers as an argument.

```
>>> a = np.array(1,2,3,4)    # WRONG
>>> a = np.array([1,2,3,4])  # RIGHT
```

NumPy

`array` transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

The type of the array can also be explicitly specified at creation time:

```
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

NumPy

Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

The function `zeros` creates an array full of zeros, the function `ones` creates an array full of onesBy default, the dtype of the created array is `float64`.

```
>>> np.zeros( (3,4) )
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

NumPy

To create sequences of numbers, NumPy provides a function analogous to range that returns arrays instead of lists.

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])

>>> np.arange( 0, 2, 0.3 )                 # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

When arange is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function linspace that receives as an argument the number of elements that we want, instead of the step:

```
>>> np.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
```

NumPy

Arithmetic operators on arrays apply *elementwise*. A new array is created and filled with the result.

```
>>> a = np.array( [20,30,40,50] )
>>> b = np.arange( 4 )
>>> b
array([0, 1, 2, 3])

>>> c = a-b
>>> c
array([20, 29, 38, 47])

>>> b**2
array([0, 1, 4, 9])

>>> 10*np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])

>>> a<35
array([ True, True, False, False], dtype=bool)
```

NumPy

Unlike in many matrix languages, the product operator * operates elementwise in NumPy arrays. The matrix product can be performed using the dot function or method.

```
>>> A = np.array( [[1,1], [0,1]] )
>>> B = np.array( [[2,0], [3,4]] )

>>> A*B                         # elementwise product
array([[2, 0],[0, 4]])

>>> A.dot(B)                    # matrix product
array([[5, 4], [3, 4]])

>>> np.dot(A, B)                # another matrix product
array([[5, 4],[3, 4]])
```

NumPy

Some operations, such as `+=` and `*=`, act in place to modify an existing array rather than create a new one.

```
>>> a = np.ones((2,3), dtype=int)
>>> b = np.random.random((2,3))
```

```
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
```

```
>>> b += a
>>> b
array([[ 3.417022  ,  3.72032449,  3.00011437],
       [ 3.30233257,  3.14675589,  3.09233859]])
```

NumPy

Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the ndarray class.

```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.18626021,  0.34556073,  0.39676747],
       [ 0.53881673,  0.41919451,  0.6852195 ]])


>>> a.sum()
2.5718191614547998


>>> a.min()
0.1862602113776709


>>> a.max()
0.6852195003967595
```

NumPy

By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the axis parameter you can apply an operation along the specified axis of an array:

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])


>>> b.sum(axis=0)                          # sum of each column
array([12, 15, 18, 21])


>>> b.min(axis=1)                          # min of each row
array([0, 4, 8])


>>> b.cumsum(axis=1)                       # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

NumPy

One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])

>>> a[2]
8

>>> a[2:5]
array([ 8, 27, 64])

>>> a[:6:2] = -1000    # equivalent to a[0:6:2] = -1000; from start to position 6, exclusive, set every 2nd element to -1000
>>> a
array([-1000,     1, -1000,    27, -1000,   125,   216,   343,   512,   729])

>>> a[ : :-1]                                 # reversed a
array([  729,   512,   343,   216,   125, -1000,    27, -1000,     1, -1000])
```

NumPy

Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas:

```
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])

>>> b[2,3]
23

>>> b[0:5, 1]                  # each row in the second column of b
array([ 1, 11, 21, 31, 41])

>>> b[ : ,1]                   # equivalent to the previous example
array([ 1, 11, 21, 31, 41])

>>> b[1:3, : ]                 # each column in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

NumPy

Iterating over multidimensional arrays is done with respect to the first axis:

```
>>> for row in b:
...     print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

However, if one wants to perform an operation on each element in the array, one can use the `flat` attribute which is an iterator over all the elements of the array:

```
>>> for element in b.flat:
...     print(element)
...
0
1
2
3
10
11
12
13
```

MatPlotLib

Matplotlib is a plotting library.

matplotlib.pyplot is a collection of command style functions that make matplotlib work like MATLAB.

Matplotlib is widely considered to be a perfect alternative to MATLAB, if it is used in combination with Numpy and Scipy. And whereas MATLAB is expensive and closed source, Matplotlib is free and open source code.

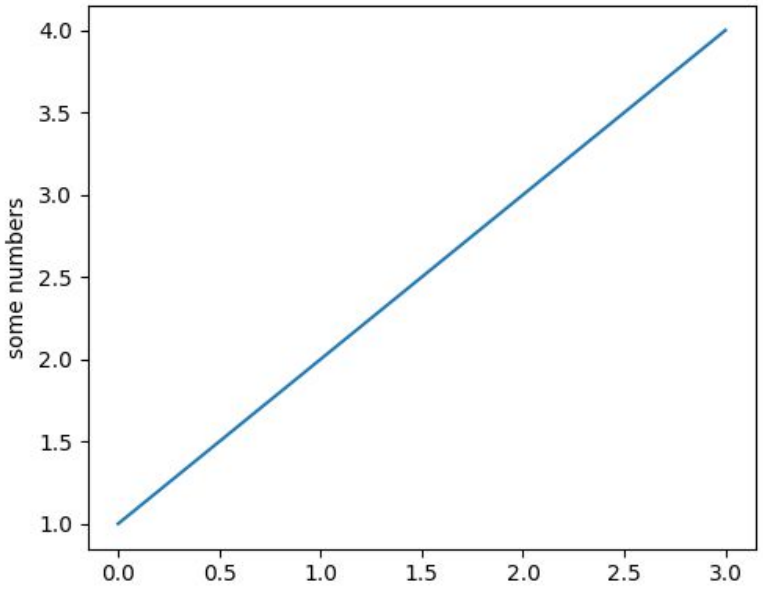
Matplotlib is object-oriented and can be used in an object oriented way.

Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

MatPlotLib

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```

If you provide a single list or array to the `plot()` command,
matplotlib assumes it is a sequence of y values,
and automatically generates the x values for you.
Since python ranges start with 0,
the default x vector has the same length as y but starts with 0.
Hence the x data are `[0,1,2,3]`.



MatPlotLib

`plot()` is a versatile command, and will take an arbitrary number of arguments.
For example, to plot x versus y, you can issue the command:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and linetype of the plot.
The default format string is ‘b-‘, which is a solid blue line.
For example, to plot the above with red circles, you would issue

```
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
```

The `axis()` command in the example above takes a list of `[xmin, xmax, ymin, ymax]` and specifies the viewport of the axes.

MatPlotLib

Generally, you will use numpy arrays. In fact, all sequences are converted to numpy arrays internally.
The example below illustrates a plotting several lines with different format styles in one command using arrays.

```
import numpy as np

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
```

MatPlotLib

Pyplot has the concept of the current figure and the current axes.
All plotting commands apply to the current axes.
The function gca() returns the current axes, and gcf() returns the current figure.

```
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
```

The subplot() command specifies `numrows, numcols, fignum` where `fignum` ranges from 1 to `numrows*numcols`. The commas in the `subplot` command are optional if `numrows*numcols<10`. So `subplot(211)` is identical to `subplot(2, 1, 1)`

MatPlotLib

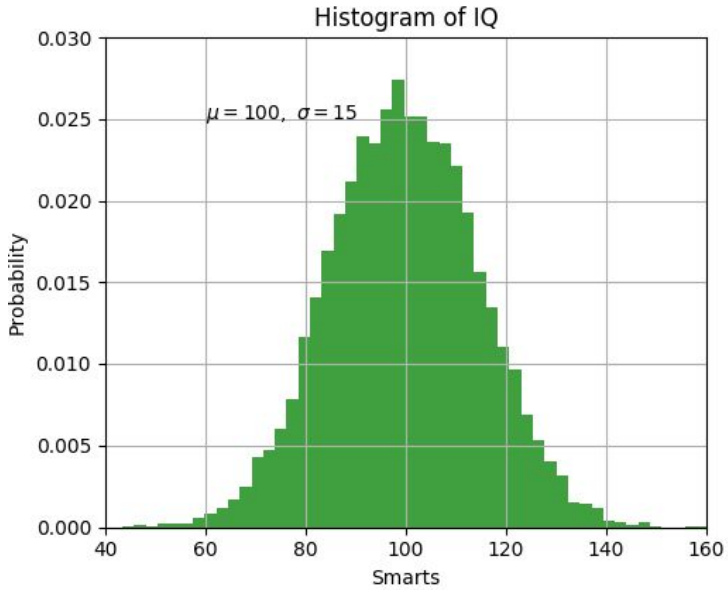
You can create multiple figures by using multiple `figure()` calls with an increasing figure number. Of course, each figure can contain as many axes and subplots as your heart desires:

```
plt.figure(1)                # the first figure

plt.subplot(211)             # the first subplot in the first figure
plt.plot([1, 2, 3])
plt.subplot(212)             # the second subplot in the first figure
plt.plot([4, 5, 6])



plt.figure(2)                # a second figure
plt.plot([4, 5, 6])          # creates a subplot(111) by default

plt.figure(1)                # figure 1 current; subplot(212) still current
plt.subplot(211)             # make subplot(211) in figure1 current
plt.title('Easy as 1, 2, 3') # subplot 211 title
```

MatPlotLib

The `text()` command can be used to add text in an arbitrary location, and the `xlabel()`, `ylabel()` and `title()` are used to add text in the indicated locations

```
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
```



MatPlotLib

`matplotlib.pyplot` supports not only linear axis scales, but also logarithmic and logit scales.
This is commonly used if data spans many orders of magnitude.
Changing the scale of an axis is easy:

plt.xscale('log')

Pandas

Pandas is a Python module, which is rounding up the capabilities of Numpy, Scipy and Matplotlab.

The word pandas is an acronym which is derived from "Python and data analysis" and "panel data".

There is often some confusion about whether Pandas is an alternative to Numpy, SciPy and Matplotlib.
The truth is that it is built on top of Numpy.
This means that Numpy is required by pandas.
Scipy and Matplotlib on the other hand are not required by pandas but they are extremely useful.
That's why the Pandas project lists them as "optional dependency".

There are two important data structures of Pandas:

- Series
- DataFrame

```
import pandas as pd
```

Pandas

Series

A Series is a one-dimensional labelled array-like object.
It is capable of holding any data type, e.g. integers, floats, strings, Python objects, and so on.
It can be seen as a data structure with two arrays: one functioning as the index, i.e. the labels, and the other one contains the actual data.

We define a simple Series object in the following example by instantiating a Pandas Series object with a list.

```
import pandas as pd
S = pd.Series([11, 28, 72, 3, 5, 8])
S
```

Pandas created a default index starting with 0 going to 5, which is the length of the data minus 1.

We can directly access the index and the values of our Series S:

```
print(S.index)
print(S.values)
```

Pandas

Series

We can start defining Series objects with individual indices:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
S
```

Pandas

Series

We can use arbitrary indices.

If we add two series with the same indices, we get a new series with the same index and the corresponding values will be added:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits)
print(S + S2)
print("sum of S: ", sum(S))
```

Pandas

Series

The indices do not have to be the same for the Series addition. The index will be the "union" of both indices. If an index doesn't occur in both Series, the value for this Series will be NaN:

```
fruits = ['peaches', 'oranges', 'cherries', 'pears']
fruits2 = ['raspberries', 'oranges', 'cherries', 'pears']
S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits2)
print(S + S2)
```

Pandas

Series

It's possible to access single values of a Series or more than one value by a list of indices:

```
print(S['apples'])

print(S[['apples', 'oranges', 'cherries']])
```

Pandas

Series

Similar to Numpy we can use scalar operations or mathematical functions on a series:

```
print((S + 3) * 4)
```

Pandas

pandas.Series.apply

Series.apply(func, convert_dtype=True, args=(), **kwds)

The function "func" will be applied to the Series and it returns either a Series or a DataFrame, depending on "func".

Parameter	Meaning
func	a function, which can be a NumPy function that will be applied to the entire Series or a Python function that will be applied to every single value of the series
convert_dtype	A boolean value. If it is set to True (default), apply will try to find better dtype for elementwise function results. If False, leave as dtype=object
args	Positional arguments which will be passed to the function "func" additionally to the values from the series.
**kwds	Additional keyword arguments will be passed as keywords to the function

Pandas

DataFrame

The underlying idea of a DataFrame is based on spreadsheets.
It contains an ordered collection of columns.
Each column consists of a unique data typye, but different columns can have different types, e.g. the first column may consist of integers, while the second one consists of boolean values and so on.
A DataFrame has a row and column index; it's like a dict of Series with a common index.

```
cities = {"name": ["London", "Berlin", "Madrid", "Rome",
                   "Paris", "Vienna", "Bucharest", "Hamburg",
                   "Budapest", "Warsaw", "Barcelona",
                   "Munich", "Milan"],
          "population": [8615246, 3562166, 3165235, 2874038,
                         2273305, 1805681, 1803425, 1760433,
                         1754000, 1740119, 1602386, 1493900,
                         1350680],
          "country": ["England", "Germany", "Spain", "Italy",
                      "France", "Austria", "Romania",
                      "Germany", "Hungary", "Poland", "Spain",
                      "Germany", "Italy"]}
city_frame = pd.DataFrame(cities)
city_frame
```

	country	name	population
0	England	London	8615246
1	Germany	Berlin	3562166
2	Spain	Madrid	3165235
3	Italy	Rome	2874038
4	France	Paris	2273305
5	Austria	Vienna	1805681
6	Romania	Bucharest	1803425
7	Germany	Hamburg	1760433
8	Hungary	Budapest	1754000
9	Poland	Warsaw	1740119
10	Spain	Barcelona	1602386
11	Germany	Munich	1493900
12	Italy	Milan	1350680

Pandas

DataFrame

Rearranging the Order of Columns

We can also define or rearrange the order of the columns.

```
city_frame = pd.DataFrame(cities, columns=["name", "country", "population"])
```

Pandas

DataFrame

We can calculate the sum of all the columns of a DataFrame or the sum of certain columns:

```
print(city_frame.sum())
```

The previous code returned the following output:

```
name          LondonBerlinMadridRomeParisViennaBucharestHamb...
population                                             33800614
dtype: object
```

```
city_frame["population"].sum()
```

The previous code returned the following output:

```
33800614
```

Pandas

DataFrame

We can use "cumsum" to calculate the cumulative sum:

```
x = city_frame["population"].cumsum()
print(x)
```

Pandas

Assigning New Values to Columns

x is a Pandas Series. We can reassign the previously calculated cumulative sums to the population column:

```
city_frame["population"] = x
```

Instead of replacing the values of the population column with the cumulative sum, we want to add the cumulative population sum as a new column with the name "cum_population".

```
city_frame = pd.DataFrame(cities, columns=["country", "population", "cum_population"])
```

The column "cum_population" is set to Nan, as we haven't provided any data for it.

We will assign now the cumulative sums to this column:

```
city_frame["cum_population"] = city_frame["population"].cumsum()
```

Pandas

Accessing the Columns of a DataFrame

There are two ways to access a column of a DataFrame. The result is in both cases a Series:

```
# in a dictionary-like way:
print(city_frame["population"])

# as an attribute
print(city_frame.population)
```

Accessing the Rows of a DataFrame

We can also access the rows directly. We access the info of the fourth city in the following way:

```
city_frame.ix["Hamburg"]
```

Pandas

Reading a csv File

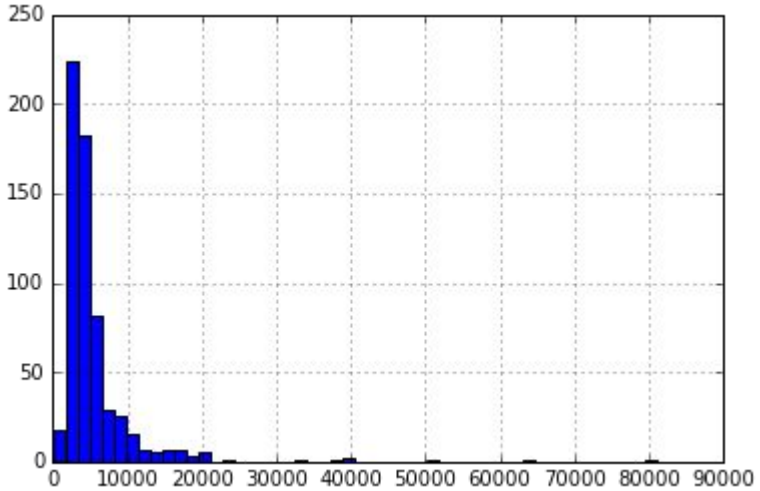
We want to read in a csv file with the population data of all countries (July 2014). The delimiter of the file is a space and commas are used to separate groups of thousands in the numbers:

```
pop = pd.read_csv("countries_population.csv",
                  header=None,
                  names=["Country", "Population"],
                  index_col=0,
                  quotechar="'",
                  sep=" ",
                  thousands=",")
```

Example

Lets start by plotting the histogram of ApplicantIncome

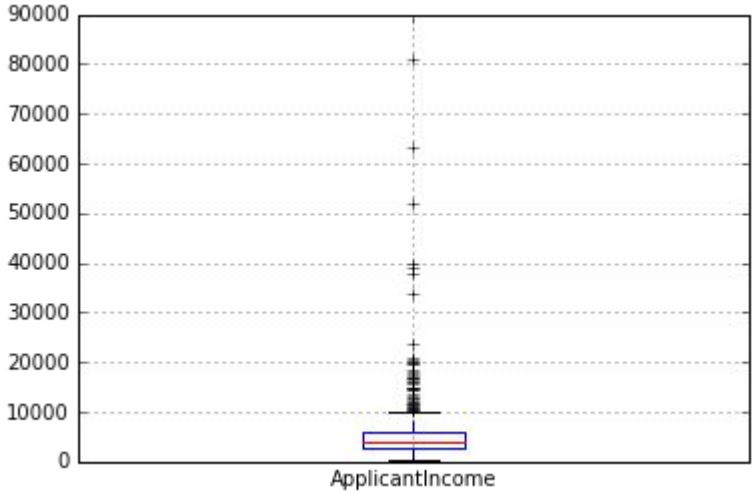
Here we observe that there are few extreme values. This is also the reason why 50 bins are required to depict the distribution clearly.



Example

Next, we look at box plots to understand the distributions.

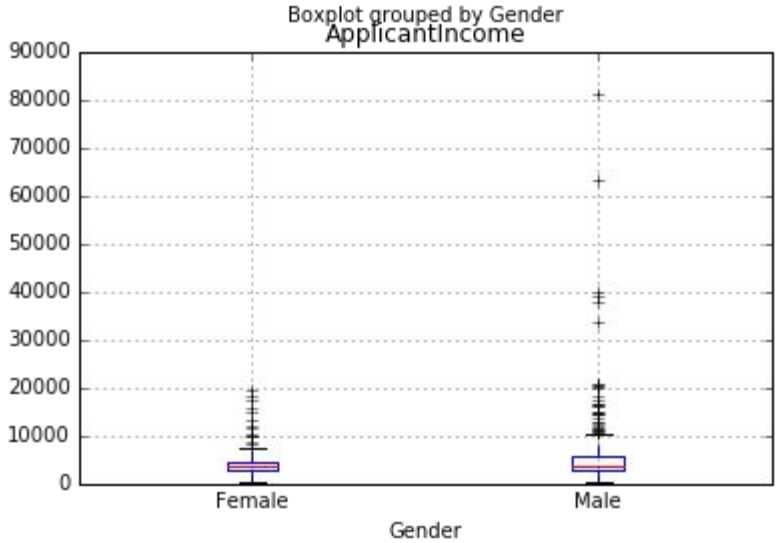
This confirms the presence of a lot of outliers/extreme values. This can be attributed to the income disparity in the society. Part of this can be driven by the fact that we are looking at people with different education levels.



Example

Let us segregate them by Education.

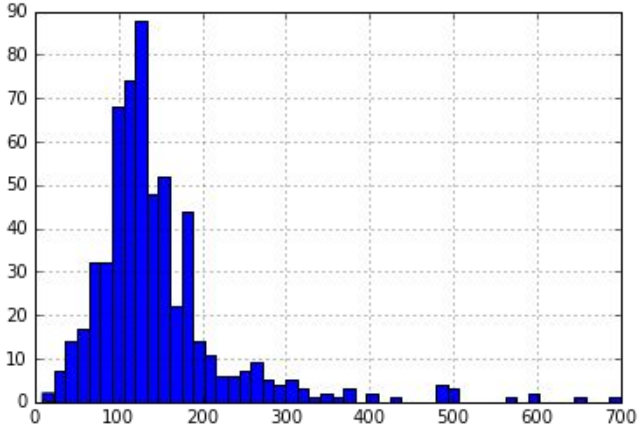
We can see that there is no substantial difference between the mean income of graduate and non-graduates. But there are a higher number of graduates with very high incomes, which are appearing to be the outliers.

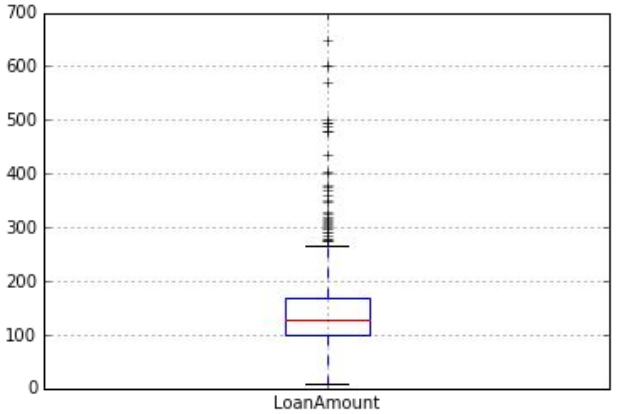


Example

Now, Let's look at the histogram and boxplot of LoanAmount

Again, there are some extreme values. Clearly, both ApplicantIncome and LoanAmount require some amount of data munging. LoanAmount has missing and well as extreme values values, while ApplicantIncome has a few extreme values, which demand deeper understanding.





Example

Now that we understand distributions for ApplicantIncome and LoanIncome, let us understand categorical variables in more details.

```
Frequency Table for Credit History:
0     89
1    475
Name: Credit_History, dtype: int64

Probility of getting loan for each Credit History class:
Credit_History
0    0.078652
1    0.795789
Name: Loan_Status, dtype: float64
```