We have used many Python functions

- **Mathematical functions:**
  ```python
  from math import *
  y = sin(x)*log(x)
  ```

- **Other functions:**
  ```python
  n = len(somelist)
  ints = range(5, n, 2)
  ```

- **Functions used with the dot syntax (called *methods*):**
  ```python
  C = [5, 10, 40, 45]
  i = C.index(10)  # result: i=1
  C.append(50)
  C.insert(2, 20)
  ```

- **What is a function?** So far we have seen that we put some objects in and sometimes get an object (result) out

- **Next topic:** learn to write your own functions
Function = a collection of statements we can execute wherever and whenever we want

Function can take input objects and produce output objects

Functions help to organize programs, make them more understandable, shorter, and easier to extend

Simple example: a mathematical function $F(C) = \frac{9}{5}C + 32$

```python
def F(C):
    return (9.0/5)*C + 32
```

Functions start with `def`, then the name of the function, then a list of arguments (here `C`) – the `function header`

Inside the function: statements – the `function body`

Wherever we want, inside the function, we can ”stop the function” and return as many values/variables we want
A function does not do anything before it is called

Examples on calling the $F(C)$ function:

```python
a = 10
F1 = F(a)
temp = F(15.5)
print F(a+1)
sum_temp = F(10) + F(20)
Fdegrees = [F(C) for C in Cdegrees]
```

Since $F(C)$ produces (returns) a `float` object, we can call $F(C)$ everywhere a `float` can be used
Example: sum the integers from start to stop

```python
def sumint(start, stop):
    s = 0  # variable for accumulating the sum
    i = start  # counter
    while i <= stop:
        s += i
        i += 1
    return s

print(sumint(0, 10))
sum_10_100 = sumint(10, 100)
```

- `i` and `s` are local variables in `sumint` — these are destroyed at the end (return) of the function and never visible outside the function (in the calling program); in fact, `start` and `stop` are also local variables.

- In the program above, there is one global variable, `sum_10_100`, and two local variables, `s` and `i` (in the `sumint` function).

- Read Chapter 2.2.2 in the book about local and global variables!!
Recall the formula \( y(t) = v_0 t - \frac{1}{2} gt^2 \):

We can make Python function for \( y(t) \):

```python
def yfunc(t, v0):
g = 9.81
return v0*t - 0.5*g*t**2
```

# sample calls:
y = yfunc(0.1, 6)
y = yfunc(0.1, v0=6)
y = yfunc(t=0.1, v0=6)
y = yfunc(v0=6, t=0.1)

Functions can have as many arguments as you like

When we make a call `yfunc(0.1, 6)`, all these statements are in fact executed:

```python
t = 0.1  # arguments get values as in standard assignments
v0 = 6

g = 9.81
return v0*t - 0.5*g*t**2
```
The \( y(t,v0) \) function took two arguments

Could implement \( y(t) \) as a function of \( t \) only:

```python
>>> def yfunc(t):
...     g = 9.81
...     return v0*t - 0.5*g*t**2
...     
...     yfunc(0.6)

NameError: global name 'v0' is not defined

v0 must be defined in the calling program program before we call yfunc

```python
>>> v0 = 5
>>> yfunc(0.6)
1.2342
```

v0 is a global variable

Global variables are variables defined outside functions

Global variables are visible everywhere in a program

\( g \) is a local variable, not visible outside of \( yfunc \)
Say we want to compute \( y(t) \) and \( y'(t) = v_0 - gt \):

```python
def yfunc(t, v0):
    g = 9.81
    y = v0*t - 0.5*g*t**2
    dydt = v0 - g*t
    return y, dydt

# call:
position, velocity = yfunc(0.6, 3)
```

Separate the objects to be returned by comma.

What is returned is then actually a tuple:

```python
>>> def f(x):
...     return x, x**2, x**4
...
>>> s = f(2)
>>> s
(2, 4, 16)
>>> type(s)
<type 'tuple'>
>>> x, x2, x4 = f(2)
```
The function

\[ L(x; n) = \sum_{i=1}^{n} \frac{1}{i} \left( \frac{x}{1+x} \right)^i \]

is an approximation to \( \ln(1 + x) \) for a finite \( n \) and \( x \geq 1 \)

Let us make a Python function for \( L(x; n) \):

```python
def L(x, n):
    x = float(x)  # ensure float division below
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1+x))**i
    return s

x = 5
from math import log as ln
print L(x, 10), L(x, 100), ln(1+x)
```
We can return more: also the first neglected term in the sum and the error \( \ln(1 + x) - L(x; n) \):

```python
def L2(x, n):
    x = float(x)
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1+x))**i
    value_of_sum = s
    first_neglected_term = (1.0/(n+1))*(x/(1+x))**(n+1)
    from math import log
    exact_error = log(1+x) - value_of_sum
    return value_of_sum, first_neglected_term, exact_error
```

# typical call:
x = 1.2; n = 100
value, approximate_error, exact_error = L2(x, n)
Functions do not need to return objects

Let us make a table of $L(x; n)$ versus the exact $\ln(1 + x)$

The table can be produced by a Python function

This function prints out text and numbers but do not need to return anything – we can then skip the final `return`

```python
def table(x):
    print '\nx=%g, ln(1+x)=%g' % (x, log(1+x))
    for n in [1, 2, 10, 100, 500]:
        value, next, error = L2(x, n)
        print 'n=%-4d %-10g (next term: %8.2e ' '
       'error: %8.2e)' % (n, value, next, error)
```

Output from `table(10)` on the screen:

```
x=10, ln(1+x)=2.3979
n=1   0.909091 (next term: 4.13e-01   error: 1.49e+00)
n=2   1.32231  (next term: 2.50e-01   error: 1.08e+00)
n=10  2.17907  (next term: 3.19e-02   error: 2.19e-01)
n=100 2.39789 (next term: 6.53e-07   error: 6.59e-06)
n=500 2.3979   (next term: 3.65e-24   error: 6.22e-15)
```
Consider a function without any return value:

```python
>>> def message(course):
...   print "\%s is the greatest fun I’ve ever experienced" % course
... >>> message(‘INF1100’) ‘INF1100’ is the greatest fun I’ve ever experienced >>> r = message(‘INF1100’) # store the return value ‘INF1100’ is the greatest fun I’ve ever experienced >>> print r None
```

None is a special Python object that represents an "empty" or undefined value – we will use it a lot later.
Functions can have arguments of the form name=value, called *keyword arguments*:

```python
>>> def somefunc(arg1, arg2, kwarg1=True, kwarg2=0):
    print arg1, arg2, kwarg1, kwarg2

>>> somefunc('Hello', [1,2])  # drop kwarg1 and kwarg2
Hello [1, 2] True 0         # default values are used

>>> somefunc('Hello', [1,2], kwarg1='Hi')
Hello [1, 2] Hi 0           # kwarg2 has default value

>>> somefunc('Hello', [1,2], kwarg2='Hi')
Hello [1, 2] True Hi        # kwarg1 has default value

>>> somefunc('Hello', [1,2], kwarg2='Hi', kwarg1=6)
Hello [1, 2] 6 Hi           # specify all args

If we use name=value for *all* arguments, their sequence can be arbitrary:

```python
>>> somefunc(kwarg2='Hello', arg1='Hi', kwarg1=6, arg2=[2])
Hi [2] 6 Hello
```
Consider a function of $t$, with parameters $A$, $a$, and $\omega$:

$$f(t; A, a, \omega) = Ae^{-at} \sin(\omega t)$$

We can implement $f$ in a Python function with $t$ as positional argument and $A$, $a$, and $\omega$ as keyword arguments:

```python
from math import pi, exp, sin

def f(t, A=1, a=1, omega=2*pi):
    return A*exp(-a*t)*sin(omega*t)

t1 = f(0.2)
t2 = f(0.2, omega=1)
t2 = f(0.2, 1, 3)  # same as f(0.2, A=1, a=3)
t3 = f(0.2, omega=1, A=2.5)
t4 = f(A=5, a=0.1, omega=1, t=1.3)
t5 = f(t=0.2, A=9)
```
Python convention: document the purpose of a function, its arguments, and its return values in a *doc string* – a (triple-quoted) string written right after the function header

**Examples:**

```python
def C2F(C):
    """Convert Celsius degrees (C) to Fahrenheit.""
    return (9.0/5)*C + 32

def line(x0, y0, x1, y1):
    """
    Compute the coefficients a and b in the mathematical expression for a straight line y = a*x + b that goes through two points (x0, y0) and (x1, y1).
    """
    a = (y1 - y0)/(x1 - x0)
    b = y0 - a*x0
    return a, b
```
A function can have three types of input and output data:
- input data specified through positional/keyword arguments
- input/output data given as positional/keyword arguments that will be modified and returned
- output data created inside the function

_All output data are returned, all input data are arguments_

Sketch of a general Python function:
```python
def somefunc(i1, i2, i3, io4, io5, i6=value1, io7=value2):
    # modify io4, io5, io7; compute o1, o2, o3
    return o1, o2, o3, io4, io5, io7
```

- `i1, i2, i3, i6`: pure input data
- `io4, io5, io7`: input and output data
- `o1, o2, o3`: pure output data
The main program

A program contains functions and ordinary statements outside functions, the latter constitute the *main program*

```python
from math import *  # in main

def f(x):  # in main
    e = exp(-0.1*x)
    s = sin(6*pi*x)
    return e*s

x = 2  # in main
y = f(x)  # in main
print 'f(%g)=%g' % (x, y)  # in main
```

The execution starts with the first statement in the main program and proceeds line by line, top to bottom.

*def* statements define a function, but the statements inside the function are not executed before the function is called.
Math functions as arguments to Python functions

- Programs doing calculus frequently need to have functions as arguments in other functions
- We may have Python functions for
  - numerical integration:  $\int_a^b f(x) \, dx$
  - numerical differentiation: $f'(x)$
  - numerical root finding: $f(x) = 0$
- Example: numerical computation of $f''(x)$ by
  $$f''(x) \approx \frac{f(x - h) - 2f(x) + f(x + h)}{h^2}$$

```python
def diff2(f, x, h=1E-6):
    r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
    return r
```
- No difficulty with $f$ being a function (this is more complicated in Matlab, C, C++, Fortran, and very much more complicated in Java)
Application of the \texttt{diff2} function

Code:

```python
def g(t):
    return t**(-6)

# make table of $g''(t)$ for 14 $h$ values:
for k in range(1,15):
    h = 10**(-k)
    print 'h=%.0e: %.5f' % (h, diff2(g, 1, h))
```

Output ($g''(1) = 42$):

```
h=1e-01: 44.61504
h=1e-02: 42.02521
h=1e-03: 42.00025
h=1e-04: 42.00000
h=1e-05: 41.99999
h=1e-06: 42.00074
h=1e-07: 41.94423
h=1e-08: 47.73959
h=1e-09: -666.13381
h=1e-10: 0.00000
h=1e-11: 0.00000
h=1e-12: -666133814.77509
h=1e-13: 66613381477.50939
h=1e-14: 0.00000
```
For $h < 10^{-8}$ the results are totally wrong

We would expect better approximations as $h$ gets smaller

Problem: for small $h$ we add and subtract numbers of approx equal size and this gives rise to round-off errors

Remedy: use float variables with more digits

Python has a (slow) float variable with arbitrary number of digits

Using 25 digits gives accurate results for $h \leq 10^{-13}$

Is this really a problem? Quite seldom – other uncertainties in input data to a mathematical computation makes it usual to have (e.g.) $10^{-2} \leq h \leq 10^{-6}$
Sometimes we want to perform different actions depending on a condition.

Consider the function

\[ f(x) = \begin{cases} \sin x, & 0 \leq x \leq \pi \\ 0, & \text{otherwise} \end{cases} \]

In a Python implementation of \( f \) we need to test on the value of \( x \) and branch into two computations:

```python
def f(x):
    if 0 <= x <= pi:
        return sin(x)
    else:
        return 0
```

In general (the `else` block can be skipped):

```python
if condition:
    <block of statements, executed if condition is True>
else:
    <block of statements, executed if condition is False>
```
We can test for multiple (here 3) conditions:

```python
if condition1:
    <block of statements>
elif condition2:
    <block of statements>
elif condition3:
    <block of statements>
else:
    <block of statements>
<next statement>
```
Example on multiple branches:

\[
N(x) = \begin{cases} 
0, & x < 0 \\
x, & 0 \leq x < 1 \\
2 - x, & 1 \leq x < 2 \\
0, & x \geq 2 
\end{cases}
\]

```python
def N(x):
    if x < 0:
        return 0
    elif 0 <= x < 1:
        return x
    elif 1 <= x < 2:
        return 2 - x
    elif x >= 2:
        return 0
```
A common construction is

```python
if condition:
    variable = value1
else:
    variable = value2
```

This test can be placed on one line as an expression:

```python
variable = (value1 if condition else value2)
```

Example:

```python
def f(x):
    return (sin(x) if 0 <= x <= 2*pi else 0)
```
If tests:

if x < 0:
  value = -1
elif x >= 0 and x <= 1:
  value = x
else:
  value = 1

User-defined functions:

def quadratic_polynomial(x, a, b, c):
    value = a*x*x + b*x + c
    derivative = 2*a*x + b
    return value, derivative

# function call:
x = 1
p, dp = quadratic_polynomial(x, 2, 0.5, 1)
p, dp = quadratic_polynomial(x=x, a=-4, b=0.5, c=0)

Positional arguments must appear before keyword arguments:

def f(x, A=1, a=1, w=pi):
    return A*exp(-a*x)*sin(w*x)
An integral
\[ \int_a^b f(x) \, dx \]
can be approximated by Simpson’s rule:
\[ \int_a^b f(x) \, dx \approx \frac{b - a}{3n} \left( f(a) + f(b) + 4 \sum_{i=1}^{n/2} f(a + (2i - 1)h) + 2 \sum_{i=1}^{n/2-1} f(a + 2ih) \right) \]

Problem: make a function `Simpson(f, a, b, n=500)` for computing an integral of \( f(x) \) by Simpson’s rule. Call `Simpson(...)` for \( \frac{3}{2} \int_0^\pi \sin^3 x \, dx \) (exact value: 2) for \( n = 2, 6, 12, 100, 500 \).
def Simpson(f, a, b, n=500):
    """
    Return the approximation of the integral of f
    from a to b using Simpson’s rule with n intervals.
    """

    h = (b - a)/float(n)

    sum1 = 0
    for i in range(1, n/2 + 1):
        sum1 += f(a + (2*i-1)*h)

    sum2 = 0
    for i in range(1, n/2):
        sum2 += f(a + 2*i*h)

    integral = (b-a)/(3*n)*(f(a) + f(b) + 4*sum1 + 2*sum2)
    return integral
def Simpson(f, a, b, n=500):

    if a > b:
        print 'Error: a=%g > b=%g' % (a, b)
        return None

    # Check that n is even
    if n % 2 != 0:
        print 'Error: n=%d is not an even integer!' % n
        n = n+1  # make n even

    # as before...
    ...
    return integral
def h(x):
    return (3./2)*sin(x)**3

from math import sin, pi

def application():
    print 'Integral of 1.5*sin^3 from 0 to pi:'
    for n in 2, 6, 12, 100, 500:
        approx = Simpson(h, 0, pi, n)
        print 'n=%3d, approx=%18.15f, error=%9.2E' %

application()
The program: verification

Property of Simpson’s rule: 2nd degree polynomials are integrated exactly!

```python
def verify():
    """Check that 2nd-degree polynomials are integrated exactly.""
    a = 1.5
    b = 2.0
    n = 8
    g = lambda x: 3*x**2 - 7*x + 2.5  # test integrand
    G = lambda x: x**3 - 3.5*x**2 + 2.5*x  # integral of g
    exact = G(b) - G(a)
    approx = Simpson(g, a, b, n)
    if abs(exact - approx) > 1E-14:  # never use == for floats!
        print "Error: Simpson’s rule should integrate g exactly"

verify()
```