# Functions

- A function is a group of statements that are run together as a unit.
- Every function begins with a header of the form:

```
def name(list_of_parameters):
```

- Examples of valid function headers are:

```
def f(x):
def toFahrenheit(celsius):
def search(array,target):
def displayTerminationMessage():
```

- Every function header begins with **def**
- **name** can be any sequence of letters, digits and underscores that follow the rules for a variable name
- **list_of_parameters** is a list of zero or more variables names separated by commas
- **If there are no parameters the parentheses () are still required!**

- Write a function that is given (passed) a temperature in Celsius and returns a temperature in Fahrenheit.
- Each statement in the body of the function must be indented the same amount.

```
In [1]: def toFahrenheit(celsius):
               return 9./5 * celsius + 32
In [2]: print(toFahrenheit(20))
68.0
```

- The function is run/executed by using its name (calling the function).
- When a function is called it must be given (passed) as many values (arguments) as there are parameters.
- Each parameter is assigned the value of the matching argument.
- The arguments match the parameters by position, the value of the first argument is assigned to the first parameter, the value of the second argument is assigned to the second parameter, etc. The names of the arguments and corresponding parameters do not matter and may be different.

- Write a function to evaluate $f(x) = ax^2 + bx + c$

```
In [1]:  def evalQuadratic(a, b, c, x):
             print('a = %g, b = %g, c = %g,\
                   x = %g' % (a, b, c, x))
             return a*x**2 + b*x + c

In [2]: print(evalQuadratic(3, -1, 10, 2))
a = 3, b = -1, c = 10, x = 2
20

In [3]: aa= -2; bb = 3; cc = -1; xx=3
In [4]: print(evalQuadratic(aa, bb, cc, xx))
a = -2, b = 3, c = -1, x = 3
-10
```

- Notice how the arguments match the parameters by position and not by name.

- Most functions end with a **return** statement of the form:

  **return** expression

- The value of the expression is the value returned (given back) by the function.

  e.g.

```
    return 9/5 * celsius + 32
```

- A function does not have to return a value, for example a function may simply display some output.

```
In [1]: from time import ctime
In [2]: def displayTerminationMessage():
            print("""
        Programmed by Stew Dent.
        Date: %s.
        End of processing.""" % ctime())
            return # optional as the function does
                   # not return a value

In [3]: displayTerminationMessage()
Programmed by Stew Dent.
Date: Sun Jun 3 07:08:10 2018.
End of processing.
```

- Any of the arguments may be an expression.

```
In [1]: aa= -2; bb = 3; cc = -1; xx=3
In [2]: print(evalQuadratic(aa*3, bb-1, (cc+10)/
                            bb, xx))
a = -6, b = 2, c = 3, x = 3
-45
```

- The result returned by a function is usually assigned to a variable.

```
In [3]: fx = evalQuadratic(aa*3, bb-1, (cc+10)/
                           bb, xx)
a = -6, b = 2, c = 3, x = 3
In [4]: print(fx)
-45
```

- The arguments may be expressions that contain the value returned by a function.
- The value returned by a function may be used anywhere a variable of the same type is used.

- Use the result of a function as an argument to another function:

```
In [1]: aa = -2; bb = 3; cc = -1; xx = 3
In [2]: def double(value):
            return 2*value

In [3]: fx = evalQuadratic(aa*3, bb-1, (cc+10)/bb,
                           double(xx))
a = -6, b = 2, c = 3, x = 6
In [4]: print(fx)
-201
```

- Use the result of a function in an expression:

```
In [5]: expr = evalQuadratic(1, 2, 3, 4) *
               evalQuadratic(4, 3, 2, 1) / 10.
a = 1, b = 2, c = 3, x = 4
a = 4, b = 3, c = 2, x = 1
In [6]: print(expr)
24.3
```

- Consider the following function:

```
In [1]: def evalQuadratic(a, b, c, x):
              print('a = %g, b = %g, c = %g, x = %g' %
                                                  (a, b, c, x))
                                     return a*x**2 + b*x + c
```

- The following function calls are valid:

```
In [2]: print(evalQuadratic(4,3,2,1))
a = 4, b = 3, c = 2, x = 1
9
In [3]: print(evalQuadratic(x=1,a=4,c=2,b=3))
a = 4, b = 3, c = 2, x = 1
9
In [4]: print(evalQuadratic(4,3,x=1,c=2))
a = 4, b = 3, c = 2, x = 1
9
```

- The following function call is NOT valid:

```
In [5]: evalQuadratic(a=4,c=3,2,1)
SyntaxError: non-keyword arg after keyword arg
```

- Keyword arguments MUST come after all of the positional arguments.

- The result of a function can be used in list comprehension:

```
In [1]: def toFahrenheit(celsius):
            return 9./5 * celsius + 32
In [2]: celsius = [c * 5 - 20 for c in range(9)]
In [3]: print(celsius)
[-20, -15, -10, -5, 0, 5, 10, 15, 20]
In [4]: fahrenheit = [toFahrenheit(c) for c in
                      celsius]
In [5]: print(fahrenheit)
[-4.0, 5.0, 14.0, 23.0, 32.0, 41.0, 50.0, 59.0,
68.0]
```

- **Local variables** are those variables defined within a function. Local variables are known only within the function in which they are declared.
- **Global variables** are those variables that are NOT defined within any function. Global variables can be accessed anywhere within a program (script).

- A local variable with the same name as a global variable hides the global variable and the local variable is used within a function.

```
In [1]: a=20; b=-2.5 # global variables
In [2]: def fcn1(x):
                   a = 10 # local variable a hides
                                 # global variable a
               return a * x * b # use local variable a
                                       #and global
variable b

In [3]: print(fcn1(5))
-125.0
```

- -125 = 10 * 5 * -2.5, where 10 is the value of the local variable **a**, -2.5 is the value of the global variable **b** and 5 is the value of the parameter **x**.
- All parameters are <u>local</u> variables, known only within the function in which they are defined.

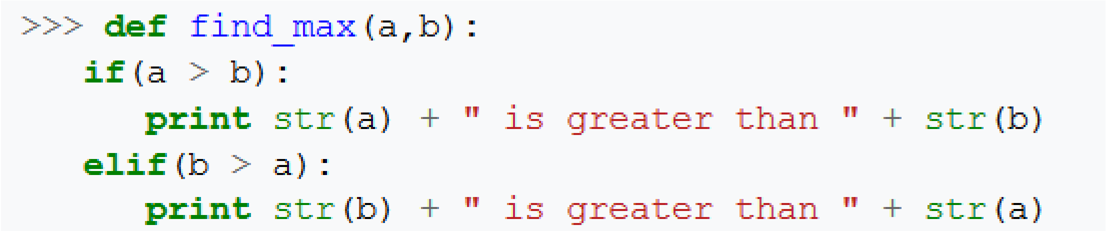
- To change the value of a global variable within a function the variable must be declared as **global** inside the function.

```
In [1]: a=20; b=-2.5 # global variables
In [2]: def fcn2(x):
            global a
            a = 10 # change value of
                   # global variable a
            return a * x * b

In [3]: print(fcn2(5))
-125.0
In [4]: print(a)
10
```

- Notice that the function changed the value of the global variable **a**

- Write a program that grab two inputs and return the maximum one



```
>>> def find_max(a,b):
   if(a > b):
      print str(a) + " is greater than " + str(b)
   elif(b > a):
      print str(b) + " is greater than " + str(a)
```

- Write a program that grab three inputs sort them descending in the output

- **Accessing and / or changing the values of global variables from within a function is considered to be the worst thing a programmer can do!**
- Communication with a function should always be through the parameters and the value returned (if any).

- The value returned from a function may be any type, including a tuple of values.
- In this case the result of the function must be assigned to a tuple or used where a tuple is required.

- e.g. a function that returns multiple values:

```
In [1]: def f(x):
            return x, x**2, x**3, x**4

In [2]: s = f(2)
In [3]: print(s)
(2, 4, 8, 16)
In [4]: print(type(s))
<type 'tuple'>
In [5]: x1, x2, x3, x4 = f(3)
In [6]: print(x1, x2, x3, x4)
3 9 27 81

In [7]: x1, x2, x3 = f(2)
Traceback (most recent call last):
  File "<pyshell#98>", line 1, in <module>
    x, x1, x3 = f(2)
ValueError: too many values to unpack (expected 3)

In [8]: x1, x2, x3, x4, x5 = f(2)
Traceback (most recent call last):
  File "<pyshell#99>", line 1, in <module>
    x, x1, x3, x4, x5 = f(2)
ValueError: not enough values to unpack (expected 5, got 4)
```

- What happens if you try to assign the result of a function that does not return anything?

```
In [1]: def displayOutput(data):
            print('OUTPUT: %s' % data)

In [2]: displayOutput('hello')
OUTPUT: hello
In [3]: output = displayOutput('hello')
OUTPUT: hello
In [4]: print(output)
None
In [5]: print(type(output))
<class 'NoneType'>
In [6]: output is None
True
In [7]: output is not None
False
```

- When a function is defined some of the parameters may be given default values, these parameters are known as keyword parameters. The other parameters are known as positional parameters.
- The keyword parameters must come after the positional parameters in the function header.

```
In [1]: def myFunc(posn1, posn2, key1='hello’,
                   key2=0):
            format = 'posn1 = %r, posn2 = %r, ' +\
                     'key1 = %r, key2 = %r’
            print(format % (posn1,posn2,key,key2))
            return


In [2]: myFunc('hi', 10)
posn1 = 'hi', posn2 = 10, key1 = 'hello', key2 = 0
In [3]: myFunc(15, -2.5, (1, 2), 'fini')
posn1 = 15, posn2 = -2.5, key1 = (1, 2), key2 = 'fini'
```

- If the names of the parameters are used the arguments may be specified in any order.

```
    In [1]: myFunc(key1='hi',posn1=29,key2=-17,posn2=0)
    posn1 = 29, posn2 = 0, key1 = 'hi', key2 = -17
```

- A **Doc String** is a string enclosed in triple quotes and may span several lines.
- A **Doc String** is often used to document a function

```
In [2]:
def toFahrenheit(celsius):
                              """
    celsius -> fahrenheit.

    Given a temperature in degrees
                              celsius convert it to degrees
    fahrenheit
                              and return that value.


                              :param float celsius: a temperature
    in celsius.
                              :return float: corresponding
    temperature in fahrenheit.
                              """
                              return 9/5 * celsius + 32
```

- The Doc String MUST occur **before** any other statement in the body of the function.

- To print out the Doc String for a function use **functionName.__doc__**
- e.g. print the Doc String for the function toFahrenheit

```
In [1]: print(toFahrenheit.__doc__)
 celsius -> fahrenheit.


Given a temperature in degrees
celsius convert it to degrees fahrenheit
and return that value.


:param float celsius: a temperature in celsius.
:return float: corresponding temperature in fahrenheit.
```

Often the output of an interactive session using the function is included in a function's Doc String as an example of how to use the function.

- The name of a function can be used as an argument to another function.

```
In [1]: def areaRectangle(length, width):
            return length * width
In [2]: def areaRightTriangle(height, width):
            return height * width / 2.
In [3]: def area(fx, v1, v2):
            return fx(v1, v2)
```

- Have **area** call **areaRectangle** to compute the area of a rectangle.

```
In [4]: print(area(areaRectangle, 2, 3))
6
```

- Have **area** call **areaRightTriangle** to compute the area of a right angle triangle.

```
In [5]: print(area(areaRightTriangle, 4, 12))
24.0
```

- Many functions are simple and contain only one line.

```
def f(x):
        return x**2 + 4
```

- This function can be written as a **lambda** function as follows:

```
f = lambda x: x**2 + 4
```

- **f** is the name of the function
- **x** is the parameter of the function
- **x******2 + 4** is the expression to be evaluated and whose value will be returned
- Lambda functions are often used to define a function as an argument to another function.

```
In [1]: print(area(lambda l, w: l * w, 3, 4))
12
```

- Lambda functions can also use keyword parameters.

```
In [2]: print(area(lambda l=1, w=2: l * w, 4, 2))
8
```

- The **main** program is the collection of all statements outside the functions plus the definitions of all the functions.
- Execution of a program begins with the first statement in the **main** program.
- The statements in a function are executed only when that function is called.

```
# main.py
from math import pi, sin  # in main

def mySin(degrees): # in main
    radians = degrees * pi / 180
    return sin(radians)

angle = 90 # degrees (in main)
sine = mySin(angle)  # in main, mySin is called here
print('sin(%g) = %g' % (angle, sine)) # in main
```

# if statement

- Consider the function:

- A python function to implement f(x) is:

```
from math import pi, sin # ifElse1.py

def f(x):
    if 0 <= x <= pi/2:
        result = sin(x)
    else:
        result = 0.0
    return(result)
```

- An **if** statement is of the form:

```
if condition:
   <block of statements> (if clause)
else:
   <block of statements>  (else clause)
```

- If the ***condition*** is True then the statements in the ***if clause*** are executed.
- If the ***condition*** is False then the statements in the ***else clause*** are executed.
- The block of statements in **one and only one** of the *if* and *else clauses* is executed.
- An alternate form of the **if** statement with no *else clause* is:

```
if condition:
   <block of statements> (if clause)
```

- The function:

Could also be written as:

```
from math import pi, sin # ifElse2.py

def f(x):
    result = 0.0
    if 0 <= x <= pi/2:
        result = sin(x)
    return(result)
```

- An ***elif*** ladder is used to select from one of many alternatives and is of the form:

```
if condition:
   <block of statements>
elif condition:
   <block of statements>
elif condition:
   <block of statements>
   .
   .
   .
else:
   <block of statements>
<next statement>
```

- Consider the function:

```
def fx(x):  # elif1.py
    if x < -10:
        result = 0
    elif x < 0:
        result = -x
    elif x <= 10:
        result = x
    else:
        result = 0
    return(result)
```

```
In [1]: print(fx(-11))
0
In [2]: print(fx(-10))
10
In [3]: print(fx(-1))
1
In [4]: print(fx(0))
0
In [5]: print(fx(1))
1
In [6]: print(fx(10))
10
In [7]: print(fx(11))
0
```

- Consider the function:

This could also be written as:

```
def fx(x):   # elif2.py
    if x < -10 or x > 10:
        result = 0
    elif x < 0:
        result = -x
    else:
        result = x
    return(result)
```

```
In [1]: print(fx(-11))
0
In [2]: print(fx(-10))
10
In [3]: print(fx(-1))
1
In [4]: print(fx(0))
0
In [5]: print(fx(1))
1
In [6]: print(fx(10))
10
In [7]: print(fx(11))
0
```

- The inline if test is of the form:

```
value1 if condition else value2
```

- It is often used to assign a value to a variable

- Recall that for a quadratic equation, which is of the form $ax^2+bx+c$

```
In [1]: r1 = (-b - sqrt(b*b - 4*a*c))/(2*a)
In [2]: r2 = (-b + sqrt(b*b - 4*a*c))/(2*a)
In [3]: root1 = r1 if abs(r1) > abs(r2) else r2
In [4]: root2 = (c/a)/root1
```

- This does the same thing as:

```
In [5]: r1 = (-b - sqrt(b*b - 4*a*c))/(2*a)
In [6]: r2 = (-b + sqrt(b*b - 4*a*c))/(2*a)
In [7]: if abs(r1) > abs(r2):
            root1 = r1
        else:
            root1 = r2
In [8]: root2 = c/a/root1
```

- Inline if tests that are used repeatedly are often used in lambda functions.

```
In [1]: max0 = lambda x: x if x > 0 else 0
In [2]: print(max0(0))
0
In [3]: print(max0(1))
1
In [4]: print(max0(-1))
0
```

- Suppose we have a variable and if the type of the variable is *int* or *float* we want to return a string that indicates if the value of the variable is negative, positive or zero. If the variable is of any other type return 'invalid type'.
- To do this requires the use of nested if statements.

```
def testNum(x):  # nestedif.py
    if type(x) is int or type(x) is float:
        if x == 0:
            result = 'zero'
        elif x < 0:
            result = 'negative'
        else:
            result = 'positive'
    else:
        result = 'invalid type'
    return(result)
```

```
from time import ctime

def threeNplus1(number):
    result = [number]
    while number != 1:
        if number % 2 == 0:
            number = number // 2
        else:
            number = 3 * number + 1
        result.append(number)
    return result

def displayList(theList, number):
    print('\nThe 3N+1 sequence for %d follows:' % number)
    for index, element in enumerate(theList):
        if index % 6 == 0 and index != 0:
            print()
        print('%10d' % (element), end = ' ')

def main():
    print('-------------------------------------------------------\n')
    number = int(input('Enter a positive integer: '))
    if number > 0:
        displayList(threeNplus1(number), number)
    else:
        print('Number must be greater than zero!')
    print("""
\nProgrammed by Stew Dent.
Date: %s.
End of processing.""" % ctime())

main()
```

- Sample output:

```
----------------------------------------------------

Enter a positive integer: 3

The 3N+1 sequence for 3 follows:
      3      10        5      16         8       4
      2       1

Programmed by Stew Dent.
Date: Sun Jun 3 11:06:34 2018.
End of processing.
```

- After this shell script is run using all of the functions in the script are available in the interactive shell.
- To run the program one or more additional times:

```
In [1]: main()
   --------------------------------------------------

   Enter a positive integer: 101

   The 3N+1 sequence for 101 follows:
       101     304     152      76      38      19
        58      29      88      44      22      11
        34      17      52      26      13      40
        20      10       5      16       8       4
         2       1
   Programmed by Stew Dent.
   Date: Sun Jun 3 11:50:43 2018.
   End of processing.
```