Modules and Functions

We have already seen that Python provides built-in function that we can use:

```python
>>> len('abcdef')
6
>>> float(5)
5.0
>>> int(5.8)
5
>>> int('125')
125
```  
These functions are available directly from the interactive shell. But …. say we want to get the square root of a number. It would seem reasonable that Python would provide a function to do that as well.

But when I try to use what I think should work, I get this:

```python
>>> sqrt(9)
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    sqrt(9)
NameError: name 'sqrt' is not defined
```  
But the following will work.

```python
>>> import math
>>> math.sqrt(9)
3.0
```  
Why?

“math” is a module (= a file) containing a number of mathematical functions.

The “import” statement instructs Python to “load” the module and make these functions available for use.
Which functions are available in the math module?

We do it with the “dir” command:

```python
>>> dir(math)
```

If we want to know what each one of these function does we use the “help” command:

```python
>>> help(math)
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available. It provides access to the
dependent mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
    acos(x)

    Return the arc cosine (measured in radians) of x.

    acosh(...)
    acosh(x)

    Return the hyperbolic arc cosine (measured in radians) of x.

    asin(...)
    asin(x)

    Return the arc sine (measured in radians) of x.

    asinh(...)
    asinh(x)

    Return the hyperbolic arc sine (measured in radians) of x.

    atan(...)
    atan(x)

    Return the arc tangent (measured in radians) of x.

    atan2(...)
```

And it goes on and on ..

You can also get help on a single function:

```python
>>> help(math.tan)
Help on built-in function tan in module math:

tan(...)
    tan(x)

    Return the tangent of x (measured in radians).
```
How do we use functions in a module?

There are three ways.

1. The way we just saw: This just makes the module available but we need to use the “dot” syntax to actually access the function.

```python
>>> import math
>>> math.sqrt(9)
3.0
```  

2. We can import a single function from a module. The function is then available to be used “directly” like the len() function.

```python
>>> from math import sqrt
>>> sqrt(9)
3.0
```  

3. We can import all the functions from a module at one time. We can then use the function name directly as in 2 above.

```python
>>> from math import *
>>> sqrt(9)
3.0
```  

Question:

What are the advantages and disadvantages of each of the methods above?

Answer:

We can also do this:

```python
>>> import math
>>> sqrt=math.sqrt
>>> sqrt(9)
3.0
```
We can create our own functions and modules!

But why would we want to? There are a number of reasons.

Answer:

How do we create functions? Easy. We use `def`:

```python
>>> def gt(x, y):
    if x>y:
        return True
    else:
        return False
```

```plaintext
>>> gt(3, 4)
False
>>> gt(4, 3)
True
```

**IMPORTANT: Terminology:** The `x` any above are called **parameters**, the 3 and 4 are called **arguments**. The arguments can be constants as in the above example, or variables as in the examples below.
Here is the syntax.

```python
def function_name( parameter list): # the parameter list could be empty – but still need ()
    code block
```

And the semantics:

1. A function needs to be defined, using the “def” construction above, before it is used. Otherwise Python will issue an error like this:

```python
>>> is_prime(5)
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    is_prime(5)
NameError: name 'is_prime' is not defined
```

2. The parameter list is a list of variables that will refer to local copies of the arguments “passed” from the calling code. For example:

```python
>>> def add(x, y):
...     x+=1
...     y+=1
...     print(x, y)

>>> a=10
>>> b=20
>>> add(a,b)
11 21
>>> a
10
>>> b
20
```
Question: How does a function return a value back to the “caller”?  

Answer: It uses the “return statement”. It has two forms:

- `return <some value>`
- `return`
- or … the function just “falls off” the last statement and implicitly returns to the caller.

The first form terminates the function and makes `<value>` available at the place from which the function was called. Program execution resumes from that point as well.

The second form just terminates the function, and computation resumes from the point from which the program was called.

The third “form” behaves just like the one above.

**Terminology:** When a function doesn’t return a value, but rather performs some function for us, we will sometimes call it a **procedure**.

```python
>>> type(len('asd'))
<class 'int'>
```

This tells us that the `len()` function returns an int.

```python
>>> type(print('asd'))
asd
<class 'NoneType'>
```

But the print function returns “NoneType”, a catchall that says that the function returns nothing. The print function is not computing a value for us, it’s basically “doing a job” for us, and then returning to the caller. We will call this a **procedure**.

**Problem:**

Write a function `is_even(x)` which returns True if `x` is even and False otherwise.

Answer:

**Problem:**
Write a function `is_leap(x)` which returns True if `x` is a leap year and False otherwise.

Answer:

Problem:

Write a function `is_prime(x)` which returns True if `x` is a prime number and False otherwise.

Answer:
Problem:
Write a program to ask the user for two integers, first and last. Write a program to print out all the primes between first and last (inclusive), five values per line.

Answer:

Problem:
Write a function sum_of_digits1(n) which returns the sum of the digits of n.

Answer:
Problem:

Write a function sum_of_digits2(n) which prints the sum of the digits of n.

Answer:

Just like there are conversions, int, float, str, there is a built in function called bin. The documentation says:

```
bin(x)

Convert an integer number to a binary string.
```

For example:

```
>>> bin(6)
'0b110'
```

Problem:

Write a function my_bin(n) which converts an integer number to a binary string representation of n. But Leave out the leading ‘0b’ returned by the built in function bin.

Answer:
Global Variables

**Question:** How can we get functions to change the values in variables outside of themselves?

We have already seen that variables in a function are local, so changing them doesn’t affect (in the instances we considered) the arguments passed over. But what about other variables, like in the following example?

```python
a=15

def change():
    a=100
    print(a)

change()
print(a)
```

We get:

```python
>>> 100 15 >>>
```

but if we add a `global` statement:

```python
a=15

def change():
    global a
    a=100
    print(a)

change()
print(a)
```

we get:

```python
>>> 100 100 >>>
```
Lists Strings, Tuples and Other Sequences

The following 2 pages are for reference.

**Sequences** represent ordered sets of objects indexed by nonnegative integers and include strings, (including Unicode strings) lists, and tuples. Strings are sequences of characters, and lists and tuples are sequences of arbitrary Python objects. Strings and tuples are immutable; lists allow insertion, deletion, and substitution of elements. All sequences support iteration.

Don’t worry! All the strange terms above will be explained below.

**Operations and Methods Applicable to All Sequences**

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s[i]</td>
<td>Returns element i of a sequence</td>
</tr>
<tr>
<td>s[i:j]</td>
<td>Returns a slice</td>
</tr>
<tr>
<td>s[i:j:stride]</td>
<td>Returns an extended slice</td>
</tr>
<tr>
<td>len(s)</td>
<td>Number of elements in s</td>
</tr>
<tr>
<td>min(s)</td>
<td>Minimum value in s</td>
</tr>
<tr>
<td>max(s)</td>
<td>Maximum value in s</td>
</tr>
</tbody>
</table>

**Operations Applicable to Mutable Sequences**

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s[i] = v</td>
<td>Item assignment</td>
</tr>
<tr>
<td>s[i:j] = t</td>
<td>Slice assignment</td>
</tr>
<tr>
<td>s[i:j:stride] = t</td>
<td>Extended slice assignment</td>
</tr>
<tr>
<td>del s[i]</td>
<td>Item deletion</td>
</tr>
<tr>
<td>del s[i:j]</td>
<td>Slice deletion</td>
</tr>
<tr>
<td>del s[i:j:stride]</td>
<td>Extended slice deletion</td>
</tr>
</tbody>
</table>

Lists are sequences of arbitrary objects.

**You create a list as follows:**

```python
names = [ "Dave", "Mark", "Ann", "Phil" ]
```

Lists are indexed by integers, starting with zero. Use the indexing operator to access and modify individual items of the list:

```python
a = names[2] # Returns the third item of the list, "Ann"
names[0] = "Jeff" # Changes the first item to "Jeff"
```

To append new items to the end of a list, use the append() method:

```python
names.append("Kate")
```
To insert an item in the list, use the insert() method:
```
names.insert(2, "Sydney")
```

You can extract or reassign a portion of a list by using the **slicing operator**:
```
b = names[0:2] # Returns ["Jeff", "Mark"]
c = names[2:] # Returns ["Sydney", "Ann", "Phil", "Kate"]
names[1] = 'Jeff' # Replace the 2nd item in names with 'Jeff'
names[0:2] = ['Dave','Mark','Jeff'] # Replace the first two items of
# the list with the list on the right.
```

Use the plus (+) operator to **concatenate** lists:
```
a = [1,2,3] + [4,5] # Result is [1,2,3,4,5]
```

Lists can contain any kind of Python object, including other lists, as in the following example:
```
a = [1,"Dave",3.14, ["Mark", 7, 9, [100,101]], 10]
```

Nested lists are accessed as follows:
```
a[1] # Returns “Dave”
a[3][2] # Returns 9
a[3][3][1] # Returns 101
```

### List Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list(s)</td>
<td>Converts s to a list.</td>
</tr>
<tr>
<td>s.append(x)</td>
<td>Appends a new element, x, to the end of s.</td>
</tr>
<tr>
<td>s.extend(t)</td>
<td>Appends a new list, t, to the end of s.</td>
</tr>
<tr>
<td>s.count(x)</td>
<td>Counts occurrences of x in s.</td>
</tr>
<tr>
<td>s.index(x [,start [,stop]])</td>
<td>Returns the smallest i where s[i] ==x. start and stop optionally specify the start- ing and ending index for the search.</td>
</tr>
<tr>
<td>s.insert(i,x)</td>
<td>Inserts x at index i.</td>
</tr>
<tr>
<td>s.pop([i])</td>
<td>Returns the element i and removes it from the list. If i is omitted, the last element is returned.</td>
</tr>
<tr>
<td>s.remove(x)</td>
<td>Searches for x and removes it from s.</td>
</tr>
<tr>
<td>s.reverse()</td>
<td>Reverses items of s in place.</td>
</tr>
<tr>
<td>s.sort([cmpfunc [, keyf [, reverse]]])</td>
<td>Sorts items of s in place. cmpfunc is a comparison function. keyf is a key function. reverse is a flag that sorts the list in reverse order.</td>
</tr>
</tbody>
</table>
We will start with

Lists

A list in Python is a mutable sequence of any time of Python object.

What does this mean??

Example:

\[ x=[1,23,“hello”, [3.4]] \]

\( x \) is the name of the list. It has 4 elements:

- the integer 1
- the integer 23
- the string “hello”
- the list [2,3]

It looks something like this in memory.
We create a new empty list in one of two ways:

- \( x=[\] \)
- \( x=list() \)

or, as above, we can create a list with elements just by listing the elements in the square brackets “[“ , “]”.

**Question:**

How do we access individual elements of a list?

**Answer:** There are two ways: beginning to end, end to beginning.

In both cases we use **square brackets with an integer** to “index” into the list. For example:

\[
\begin{array}{cccccc}
\text{x} & \text{x[-5]} & \text{x[-4]} & \text{x[-3]} & \text{x[-2]} & \text{x[-1]} \\
10 & 20 & 30 & 40 & 50 \\
\text{x[0]} & \text{x[1]} & \text{x[2]} & \text{x[3]} & \text{x[4]} \\
\end{array}
\]

Here is how the element labeling works: Assume the list has \( n \) elements.

- if we reference the element from the beginning to end: \( 1 \rightarrow n-1 \)
- if we reference the element from the end to beginning : \( -1 \rightarrow -n \)

**Some more examples:**

```python
>>> x=[1,23,"hello", [3,4]]
>>> print(x[0])
1
>>> print(x[3])
[3, 4]
>>> print(x[3][0])
3
>>> print(x[4])
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    print(x[4])
IndexError: list index out of range
```

Notice:

1. Going “from the left”, the positions in the list are numbered from 0 (not 1). Going from “the right”, from -1. In the above example, this means that the last element in the list is accessed as \( x[3] \) or \( x[-1] \).

2. Since \( x[3] \) in our example is the list \([3,4]\) we can access its elements by using a second index. That is why \( x[3][0] \) is the “zeroth” (i.e. first) element of \([3,4]\), which is 3.

3. Since there is no element in the list \( x[4] \) (they are \( x[0], x[1], x[2], x[3] \)) we are trying to access a nonexistent element and Python prints an error message.
Question:
Can we change (i.e. replace) elements of a list?

Answer:
Yes. Here is an example where we modify the list \( x \) above.

```python
>>> x[0]="Bob"
>>> x
['Bob', 23, 'hello', [3, 4]]
```

We can find out the size (=length) of a loop by using the \texttt{len()} function:

```python
>>> x
['Bob', 23, 'hello', [3, 4]]
>>> len(x)
4
```

We say that a list is \textit{mutable}. This means that is can be modified (i.e. “mutated”).

Question:
How can we add elements to an existing list?

Answer:
There are a number of different ways. We start with two functions:

- \texttt{append} – add “something” to the end of a list
- \texttt{extend} – add all the elements of some \texttt{sequence} at the end of a list
>>> a=[1,2,3]
>>> a
[1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> a.append("hello")
>>> a
[1, 2, 3, 4, 'hello']
>>> b=[5,6,7]
>>> a.append(b)
>>> a
[1, 2, 3, 4, 'hello', [5, 6, 7]]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 'hello', [5, 6, 7], 5, 6, 7]
>>> a.extend(8)
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    a.extend(8)
TypeError: 'int' object is not iterable
>>> a.extend([8])
>>> a
[1, 2, 3, 4, 'hello', [5, 6, 7], 5, 6, 7, 8]

Make sure the example above is absolutely clear!

**Lists and loops**

List and loops are made for each other!

```python
>>> s=[]
>>> for i in range(1,11):
    s.append(i)

>>> s
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Problem:

Write a program that creates a list with the integers 1 – 10. Using a for loop add up all the elements of the list and print the sum.
Problem:

Write a program that creates a list with the integers 1 – 10. Using a for loop, add up all the elements of the list that are even and print the sum.

Problem:

Write a program that creates a list with the integers 1 – 10. Using a for loop add up all the elements of the list that are odd and print the sum.

Problem:

Write a program that creates a list with the integers 1 – 10. Using a for loop add up all the elements of the list that are in even positions (0 is even) and print the sum.
When using lists it is often convenient to have Python generate some random values for us. Python provides a module called random that has some useful functions for this purpose. The two that we will use most are:

- `random` and
- `randint`

```python
>>> from random import random, randint
>>> random()
0.5697709209090985
>>> help(random)
Help on built-in function random:

random(...)  
    random() -> x in the interval [0, 1).

>>> randint(3, 45)
38
>>> help(randint)
Help on method randint in module random:

randint(self, a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.

>>> 
```

So..

Function “random()” generates a random **floating point number** from zero up to but not including one.

Function `randint(a, b)` generates a random **integer** in the range `a` to `b` inclusive. Note that `a` and `b` here are integers.

Problem:

Write a program to fill a list of size 10 with random integers in the range 1 – 10 and print it out.
Problem:

Modify the program above so that we print the list as well as the maximum integer in the list. Do this two ways.

Problem:

Modify the program above so that it also prints the position in the list where the maximum element was found.
Problem:

Using the code from the program above write a function

\[
\text{getmax}(x, i) \quad \# x \text{ is a list and } i \text{ is an integer}
\]

which will find and return the maximum element among the first \( i \) elements of list \( x \).

getmax will return two values:

- the maximum element found, and
- the position in list \( x \) where that element was found.

For example, say \( a=[4,2,7,1,45,23] \), then getmax(\( a, 4 \)) will search for the maximum element in the first 4 element of list \( a \).

So, in this case it will look at the following numbers: 2,4,7,1, and getmax(\( a, 4 \)) will return 7,2. This because in the first 4 elements, the largest is 7 and it is in position 2.

If we ran getmax(\( a, 6 \)) the function will return 45,4.
Problem:
Generate all the primes between 2 and 100.

Solution:

The **Sieve of Eratosthenes** provides an efficient solution. This algorithm is over **2200 years old**!


See there for an animation of the algorithm.

**To find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:**

1. Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).
2. Initially, let p equal 2, the first prime number.
3. Starting from p, enumerate its multiples by counting to n in increments of p, and mark them in the list (these will be 2p, 3p, 4p, etc.; the p itself should not be marked).
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.

When the algorithm terminates, all the numbers in the list that are not marked are prime.

**Why does it work?**

The main idea here is that as we go through the algorithm, every value for p is prime, because we will have already marked all the multiples of the numbers less than p. Note that some of the numbers being marked may have already been marked earlier (e.g. 15 will be marked both for 3 and 5).

**The program below is a slight modification of the above algorithm.**

This program will generate all primes in the range 2-100 using the sieve method.

We use two lists: x and primes.

**list x** will initially have the values 0-100 so that x[i]=i
**list primes**, initially empty, will grow as each new prime is found and appended onto it.

Each time a new prime p in x is located:

- p will be appended to list primes
- its place in list x will be set to zero,
- set all of its multiples in list x will be to zero.

When all elements in x are zero (we check this with the sum function), the main while loop terminates and the program prints list primes.
x=[]
for i in range(101):
    x.append(i)

primes=[2]
x[0]=x[1]=x[2]=0

p=2  # the first prime

while sum(x) != 0: # as long as sum(x)!=0, there are still non-zero entries in x.
    # Zero out all multiples of p
    i=1
    while i*p<=100:
        x[p*i]=0
        i=i+1

    # Now, look for the next prime
    p=p+1
    while x[p]==0: # it’s at the next non-zero position of x
        p=p+1

    # We found it. Add it to the list of primes, and zero out its position in x
    primes.append(p)
x[p]=0

# Done! Now print the list of primes.
print(primes)
Let’s take a break!

Here is a really interesting Microsoft/Google/Wall Street Interview Question!

1. Run the following program:

   from math import sqrt
   from random import random

   count=0

   for i in range(1000000):
       x=random()
       y=random()
       if sqrt(x*x+y*y)<1:
           count+=1

   print(4*(count/1000000))

2. What is it calculating?

3. How/why does it work? What is the theory behind this?
Slicing lists

What is a slice of a list?

If x is a list then the slice \texttt{x[a:b]} is the “sub-list” of the elements of the elements of a \texttt{from} index position \texttt{a} up to but not including index position \texttt{b}.

```python
>>> a=[1,2,3,4,5,6,7]
>>> b=a[3:5]
>>> b
[4, 5]
>>> .
```

If we want to indicate that the slice starts at the beginning of the list, we can leave out the start value:

```python
>>> c=a[:5]
>>> c
[1, 2, 3, 4, 5]
>>> .
```

If we want to indicate that the slice goes all the way to the end of the list, we can leave out the end value:

```python
>>> d=a[4:]
>>> d
[5, 6, 7]
>>> .
```

Leaving out both the start and end indexes is the same as saying the whole list. So:

```python
>>> e=a[:]
>>> e
[1, 2, 3, 4, 5, 6, 7]
>>> a
[1, 2, 3, 4, 5, 6, 7]
>>> .
```
What can we do with a slice of a list?

1. As we saw above, we can create a new list form a slice.

2. We can assign to a slice and thereby replace one sub-list by another.

```python
define a[2:5]=['a', 'b', 'c']
define a[2, 2, 'a', 'b', 'c', 6, 7]
```

Notice that this is a generalization of accessing and replacing one list element as in `a[1]=12` which just replaced a single list element.

**When we use a slice we can indicate a stride.**

**Huh?**

The stride is the length of the “step” that you take going from one element to the next when creating the slice.

In the following example 2 is the stride.

```python
define x=[10, 20, 30, 40, 50, 60, 70, 80, 90]
define y=x[1:8:2]
define y
```

We can assign a list to a slice with a stride, but the list on the right hand side of the assignment must be the same size as the list produced by the slice. In the following example, both are of size 4.

```python
define x[1:8:2]=['a', 'b', 'c', 'd']
define x
```

Note: The right hand side of a slice assignment can be any iterable (a string for example) as long as the lengths are the same.

```python
define a=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
define a
```

---

97