WHAT WE HAVE LEARNED

How to handle sequences - lists, strings, tuples
TODAY’S CONTENTS

More about Strings and Functions
Strings are **immutable sequences**

```python
def is_palindrome(s):
    for i in range(len(s) /2):
        if s[i] != s[len(s) - i - 1]
            return False
    return True
```

The `in` operator for strings:

```none
>>> "abc" in "01234abcde"
True
>>> "abce" in "01234abcde"
False
```

(For tuple and list, `in` tests whether something is equal to an element of the list or tuple)
# STRING SPECIAL OPERATORS

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Concatenation - Adds values on either side of the operator</td>
<td>a + b will give HelloPython</td>
</tr>
<tr>
<td>*</td>
<td>Repetition - Creates new strings, concatenating multiple copies of the same string</td>
<td>a*2 will give HelloHello</td>
</tr>
<tr>
<td>[]</td>
<td>Slice - Gives the character from the given index</td>
<td>a[1] will give e</td>
</tr>
<tr>
<td>[ : ]</td>
<td>Range Slice - Gives the characters from the given range</td>
<td>a[1:4] will give ell</td>
</tr>
<tr>
<td>in</td>
<td>Membership - Returns true if a character exists in the given string</td>
<td>H in a will give 1</td>
</tr>
<tr>
<td>not in</td>
<td>Membership - Returns true if a character does not exist in the given string</td>
<td>M not in a will give 1</td>
</tr>
<tr>
<td>r/R</td>
<td>Raw String - Suppress actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter &quot;r,&quot; which precedes the quotation marks. The &quot;r&quot; can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quotation mark.</td>
<td>print r'\n' prints \n and print R'\n' prints \n</td>
</tr>
<tr>
<td>%</td>
<td>Format - Performs String formatting</td>
<td>See at next section</td>
</tr>
</tbody>
</table>
String objects have many useful methods:
- `upper()`, `lower()`, and `capitalize()`
- `isalpha()` and `isdigit()`
- `startswith(prefix)` and `endswith(suffix)`
- `find(str)`, `find(str, start)`, and `find(str, start, end)`
  (return -1 if `str` is not in the string)
- `replace(str1, str2)`
- `rstrip()`, `lstrip()` and `strip()` to remove white space on the right, left, or both ends.

String methods for converting between lists and strings:
- `split()` splits with white space as separator
- `split(sep)` splits with given separator `sep`
- `join(l)` concatenates strings from a list `l`
- `repr()` or `str()` converts any value to a string

```python
>>> str(39.95)
'39.95'
>>> repr(39.95)
'39.950000000000003'
```
FUNCTION ARGUMENTS

We have learnt about parameters and function arguments:

```python
def create_sun(radius, color):
    sun = Circle(radius)
    sun.setFillColor(color)
    sun.setBorderColor(color)
    sun.moveTo(100, 100)
    return sun

sun = create_sun(30, "yellow")
```

Arguments are mapped to parameters one-by-one, left-to-right.
We can provide default parameters:

```python
def create_sun(radius = 30, color = "yellow"):
    ...
```

Now we can call it like this:

```python
sun = create_sun()
star = create_sun(2)
moon = create_sun(28, "silver")
```

But not like this:

```python
moon = create_sun("silver")
```
Default parameters have to locate after normal parameters

```python
def avg(data, start = 0, end = None):
    if not end:
        end = len(data)
    return sum(data[start:end])/float(end-start)
```

```python
>>> d = [ 1, 2, 3, 4, 5 ]
>>> avg(d)
3.0
>>> avg(d, 2)
4.0
>>> avg(d, 1, 4)
3.0
```
The default values are evaluated at the point of function definition in defining scope:

```python
i = 5
def f(arg=i):
    print arg

i=6
f()
```

What is the result?
NORMAL AND DEFAULT PARAMETERS

The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```python
def f(a, L=0):
    L += a
    return L
print f(1)
print f(2)
print f(3)

def f(a, L=[]):
    L.append(a)
    return L
print f(1)
print f(2)
print f(3)

def f(a, L=None):
    if L is None:
        L = []
        L.append(a)
    return L
print f(1)
print f(2)
print f(3)
```

What are the results?
We can include the name of the parameter in the function call to make the code clearer. The order of arguments does not matter.

```python
def avg(data, start = 0, end = None):
    if not end:
        end = len(data)
    return sum(data[start:end])/float(end-start)
```

```>>> avg(d, end=3)
2.0
>>> avg(data=d, end=3)
2.0
>>> avg(end=3, data=d)
2.0
>>> avg(end=3, d)
SyntaxError: non-keyword arg after keyword arg
```
def subtract(a, b):
    return a-b
print subtract(5, 2)
print subtract(b=5, a=2)
# print subtract(a=5, 2)
def add(a, b=3):
    return a+b
print add(5, 2)
print add(5)
# print add(b=3, 5)
# print add(b=3, 5)
print add(a=3, b=3)
PASS BY REFERENCE VS VALUE

All parameters (arguments) in the Python are passed by reference

def changeme(mylist):
    #This changes a passed list into this function
    mylist.append([1,2,3,4]);
    print "Values inside the function: ", mylist
    return
    # Now you can call changeme function

mylist = [10, 20, 30]
changeme(mylist);
print "values outside the function: ", mylist

What are the results?
What’s the result of this code?

```python
def changeme(mylist):
    #This would assign new reference in mylist
    mylist = [1, 2, 3, 4];
    print "Values inside the function: ", mylist
    return
    # Now you can call changeme function

mylist = [10, 20, 30]
changeme(mylist);
print “values outside the function:”, mylist
```
You may need to process a function for more arguments than you specified while defining the function.

These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

The general syntax for a function with non-keyword variable arguments:

```python
def funcName([formal_args], *var_args_tuple):
    #function_body
    ...
```

An *asterisk* (*) is placed before the variable name that will hold the values of all non-keyword variable arguments.
def printinfo( arg1, *vartuple):
    print "Output is: "
    print arg1
    for var in vartuple:
        print var
    return;

printinfo( 10 );
printinfo( 70, 60, 50 );

What are the results?
**ANONYMOUS FUNCTIONS**

**Lambda** keyword is used to create small anonymous function
- No return statement. Instead, lambda includes expression that is returned
- Lambda is an expression, not a statement
- Can access variables that locates in its own local namespace only

**Syntax**
- `lambda[arg[,arg2, ...argn]]: expression`

**Example**
```python
def make_incrementor(n):
    return lambda x: x+n
f = make_incrementor(42)
print f(0)
print f(1)`
```
1) 
   \[ L = [ \text{lambda } x: x^2, \text{lambda } x: x^3, \text{lambda } x: x^4 ] \]

   for f in L :
       print(f(3))

2) 
   \[ \text{min} = (\text{lambda } x, y: x \text{ if } x<y \text{ else } y) \]
   print min(101*98, 102*99)
BUILT-IN FUNCTIONS FOR LIST USING LAMBDA

```python
>>> data = [1, 10, 9, 23, 17, 24, 7, 12, 30]
>>> print filter(lambda x: x%3 == 0, data)
[9, 24, 12, 30]
>>> print map(lambda x: x*2 +10, data)
[12, 30, 28, 56, 44, 58, 24, 34, 70]
>>> print reduce(lambda x, y: x+y, data)
133
```
BUILT-IN FUNCTIONS FOR LIST

Compute prime numbers in Python

```python
import math

def primes(n):
    nums=range(2,n)
    sqrtn=math.sqrt(n)
    i=0
    while((nums[i] <= sqrtn) and (i < len(nums))):
        nums=filter(lambda x: (x==nums[i] or x%nums[i] !=0), nums)
        print nums
        i += 1
    return nums

primes(30)
```