# Algorithms and Programming I

Lecture#13

Spring 2015

# Conditional Statements

- Conditional execution in Python using the IF or if/else statement
- Indentation is used to indicate groups of statements that will be executed conditionally.
- A condition is an expression that can be true or false.

# 'if' statement; block structure

```
if (condition):
    action1
```

```
if (condition):
    action1
else:
    action2
```

```
if (condition):
    action1
elif (condition):
    action2
```

```
if (condition):
    action1
elif (condition):
    action2
else:
    action3
```

```
if (condition):
    action1
elif (condition):
    action2
elif (condition):
    action3
else:
    action3
```

- Block structure is determined by indentation

```
if (condition):
    action1
```

indentation

# Repetition statements-Loops(1)

- Repetition in python may be done using the while statement.
- Indentation is used to indicate groups of statements.
- Indentation will indicate the statement or group of statements that will be executed repeatedly.
- Programmers call repetition statements LOOPS!

# Repetition statements-Loops(2)

- **Loop** is a statement or group of statements that execute repeatedly until a terminating condition is satisfied
- **Infinite loop**: A loop in which the terminating condition is never satisfied

While statements have the form :

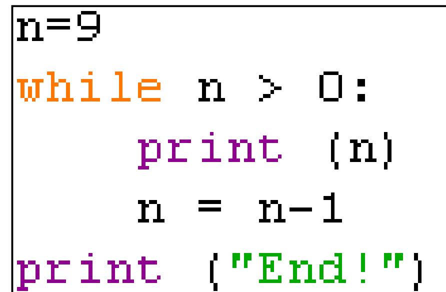
```
while condition:              # don’t forget the colon : after the condition
        statement_1           # execute if the loop condition is true
          ....
        statement _ n         #Go back to loop condition after this statement
Statement _ after _loop       # execute after the loop condition is false.
```

# Counter controlled loops

- Counter controlled use a counter variable that controls the iteration.
- Usually counter modification is the last thing in the loop body .
- Counter variable may count up, down, by ones or two, …. According to the counter variable modification and condition.

# `while` Statement

- Repetition of a block of statements
- Loop until test becomes false, or 'break‘



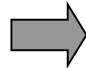
```
n=9
while n > 0:
     print (n)
     n = n-1
print ("End!")
```

- **Explanation**: “While n is greater than 0, continue displaying the value of n and then reducing the value of n by 1. When you get to 0, display the word End!”

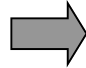
# `while` Statement (cont.)

- What are the outputs of the following programs?

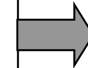
```
n=9
while n > 0:
    n = n-1
print (n)
print ("End!")
```



```
n=1
while (0<n<16):
    n = n + 2
    print (n)
print ("End!")
```



```
n=1
m=2
while ((n<10) and(m<20)):
    n = n + 2
    m = m + 3
    print ("n + m:", n+m)
print ("End!")
```



# For Loop

- `for` name `in range(`max`):`
-         statements

- Repeats for values 0 (inclusive) to max (exclusive)

```
>>> for i in range(5):
...     print(i)
0
1
2
3
4
```

python™

# For Loop variations

- `for` name `in range(`min`,` max`):`
- statements
- `for` name `in range(`min`,` max`,` step`):`
- statements
- Can specify a minimum other than 0, and a step other than 1

```
>>> for i in range(2, 6):
...     print(i)
2
3
4
5
>>> for i in range(15, 0, -5):
...     print(i)
15
10
5
```

# Built-in functions 'range(..)'

- It is used to terate over a sequence of numbers:
- Examples:
  - `range(10)` : generates a list of 10 values starting from 0 and incrementing by value 1 (Note that **10** is **not included**)
    - `[0,1,2,3,4,5,6,7,8,9]`

  - `range(0, 10, 2)` : generates values between 1 and 10 with increment value (or step value) 2
    - `[0,2,4,6,8]`

# Exercise

```
for j in range(50):
    if ((j%8)==0):
        print (j)
```

```
for j in range(0, 50, 8):
        print (j)
```

# Exercise 1

```
for i in range(3):
    for j in range(3):
        print (i+j)
```

# Exercise 2

- Write a program which prints the odd numbers between 1 and 150 (150 is not included)

```
for j in range(150):
    if((j%2) !=0):
        print (j)
```

# Exercise 3

- Write a program which prints the sum of numbers between 1 and 50 (50 is not included)

```
sum = 0
for i in range(50):
    sum = sum + i
print(sum)
```

# Exercise 4

- Write a program which prints the prime numbers between 2 and 100 (100 is not included)

```
prime = True
for i in range(2,20,1):
    prime = True
    for j in range(2,i,1):
        if (i != j):
            if((i%j)==0):
                prime= False

    if (prime):
        print(i)
```

So far we have the following in Python:

1. Assignments.
2. Conditionals. ( if statements)
3. Input / Output
4. Looping constructs ( For, While)

- Is that enough to write a piece of code?

# Functions

Functions:

(1) Allow us to break up into modules.

(2) Suppressed details.

(3) Create “new primitives” .

# Functions

- a **function** is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements..

# Python Functions

- There are two kinds of functions in Python.
  1. Built-in functions that are provided as part of Python – raw_ input(),input(), type(), float(), int() ...
  2. User Defined Functions that we define ourselves and then use.
- We treat the we “built function names” as "new" reserved words (i.e. we avoid them as variable names).

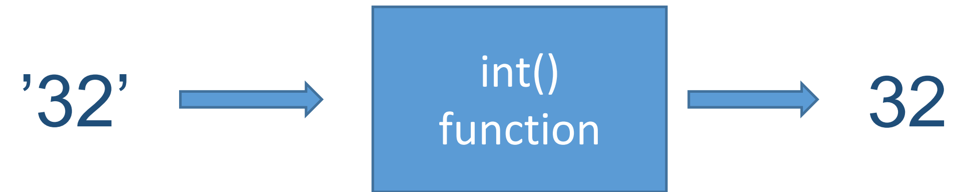
# Function Definition

- In Python a function is some reusable code that takes arguments(s) as input, does some computation and then returns a result or results.
- We define a function using the def ( case sensitive!) reserved word.
- We call/invoke the function by using the function name, parenthesis and arguments in an expression .
- We don’t have to pass argument when we call the function , but still we have to have the parenthesis ().

# Type conversion functions

- Python provides built-in functions that convert values from one type to another. The int function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

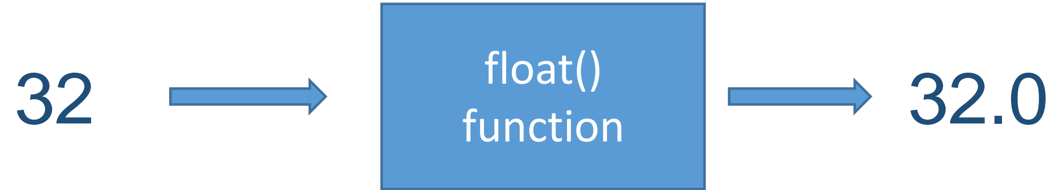



- int can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

# Type conversion functions…

- float converts Integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```




- Finally, str converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

# Math functions

- Python has a math module that provides most of the familiar mathematical functions. A **module** is a file that contains a collection of related functions. But before we can use the module, we have to import it:

>>> import math → This statement creates a **module object** named math.

- To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called **dot notation**.

# Math functions

- Example 1:

```
>>> ratio = signal_ power / noise _ power
>>> decibels = 10 * math.log10(ratio)
```

- Example 2:

```
>>> radians = 0.7
>>> height = math.sin(radians)
```

# Python Build _in_ functions ( python 2.75)

| | | Built-in Functions | | |
|---|---|---|---|---|
| abs() | divmod() | input() | open() | staticmethod() |
| all() | enumerate() | int() | ord() | str() |
| any() | eval() | isinstance() | pow() | sum() |
| basestring() | execfile() | issubclass() | print() | super() |
| bin() | file() | iter() | property() | tuple() |
| bool() | filter() | len() | range() | type() |
| bytearray() | float() | list() | raw_input() | unichr() |
| callable() | format() | locals() | reduce() | unicode() |
| chr() | frozenset() | long() | reload() | vars() |
| classmethod() | getattr() | map() | repr() | xrange() |
| cmp() | globals() | max() | reversed() | zip() |
| compile() | hasattr() | memoryview() | round() | __import__() |
| complex() | hash() | min() | set() | apply() |
| delattr() | help() | next() | setattr() | buffer() |
| dict() | hex() | object() | slice() | coerce() |
| dir() | id() | oct() | sorted() | intern() |

# Python Build _in_functions ( Python 3.4)

| | | Built-in Functions | | |
|---|---|---|---|---|
| abs() | dict() | help() | min() | setattr() |
| all() | dir() | hex() | next() | slice() |
| any() | divmod() | id() | object() | sorted() |
| ascii() | enumerate() | input() | oct() | staticmethod() |
| bin() | eval() | int() | open() | str() |
| bool() | exec() | isinstance() | ord() | sum() |
| bytearray() | filter() | issubclass() | pow() | super() |
| bytes() | float() | iter() | print() | tuple() |
| callable() | format() | len() | property() | type() |
| chr() | frozenset() | list() | range() | vars() |
| classmethod() | getattr() | locals() | repr() | zip() |
| compile() | globals() | map() | reversed() | __import__() |
| complex() | hasattr() | max() | round() | |
| delattr() | hash() | memoryview() | set() | |

# Building our Own Functions

- So far, we have only been using the functions that come with Python, but it is also possible to add new functions.

The syntax for a function definition is:

```
def NAME ( list of Parameters):
      statments
```

- We create a new function using the def keyword followed by optional parameters in parenthesis.
- We indent the body of the function. ( an indentation of two spaces will be used here)
- The list of parameters specifies what information ,if any, you have to provide in order to use the new function.
- This defines the function but *does not* execute the body of the function.
- The execution of a function introduces a new symbol table used for the local variables of the function.

# Definitions and Uses

- Example ( no parameters)

```
def newLine():
    print
```

- This is the store and reuse pattern.
- Once we have defined a function, we can call (or invoke) it as many times as we like.
- The syntax for calling the new function is the same as the syntax for built-in functions:

```
newline()
```

# Parameters and Arguments

**Argument:**

A value passed to a ***function*** (or ***method***) when calling the function.

**Parameter:**

A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept.

# Arguments

- An argument is a value we pass into the function as its input when we call the function
- We use arguments so we can direct the function to do different kinds of work when we call it at different times
- We put the arguments in parenthesis after the name of the function

```
big = max(1,2,1,0)
```

# Parameters

- A parameter is a variable which we use in the function definition that is a “handle” that allows the code in the function to access the arguments for a particular function invocation.

```
>>> def greet(lang):...
       if lang == 'es':...
           print 'Hola'...
      elif lang == 'fr':...
           print 'Bonjour'...
      else:...
           print 'Hello'...

>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
>>>
```

# Return Values

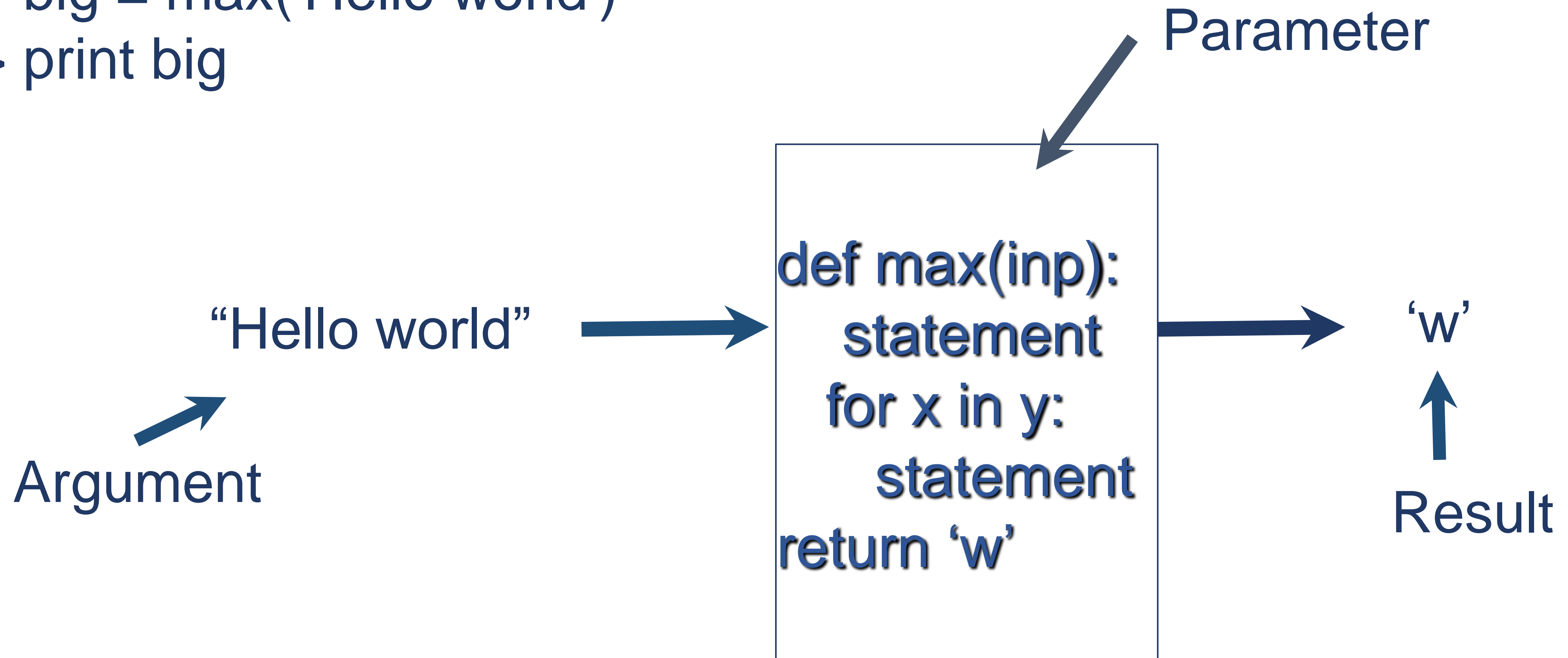
- Often a function will take its arguments, do some computation and return a value to be used as the value of the function call in the calling expression.  The return keyword is used for this.

# Return Value

- A “fruitful” function is one that produces a result (or return value)
- The return statement ends the function execution and “sends back” the result of the function

```
>>> def greet(lang):
        if lang == 'es' :
            return 'Hola'
        elif lang == 'fr'  :
             return 'Bonjour‘
        else:
            return 'Hello‘
 >>> print greet('en'),'Glenn'Hello Glenn
>>> print greet('es'),'Sally'Hola Sally
>>> print greet('fr'),'Michael'Bonjour
Michael
>>>
```

# Arguments, Parameters, and Results



# Multiple Parameters / Arguments

- We can define more than one parameter in the function definition
- We simply add more arguments when we call the function

```
def addtwo(a, b):
    added = a + b
    return added
x = addtwo(3, 5)
print x
```

# Void (non-fruitful) Functions

- When a function does not return a value, we call it a "void" function.

# Sequences

An ordered sets which support efficient element access using integer indices ( via the __getitem__() )special method and defines a len() method that returns the length of the sequence.

Some built-in sequence types are:

(1) lists,
(2) Strings
(3)  and tuple.

# Lists

- *compound* data types, used to group together other values.
- can be written as a list of comma-separated values (items) between square brackets.
- Lists might contain items of different types, but usually the items all have the same type.

Example:

squares = [1, 4, 9, 16, 25]

Like strings (and all other built-in *sequence* type), lists can be indexed and sliced:

```
>>> squares[0]      # indexing returns the item 1
```

If an index has a negative value, it counts backward from the end of the list:

```
>>> squares[-1]
25
>>> squares[-3:] # slicing returns a new list
[9, 16, 25]
```

# Nested list

- A list within another list is said to be **nested**.

Example:

m=[‘hello’,2.3,5,[10,20]]

Lists that contain consecutive integers are common, so Python provides a simple way to create them:

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

The range function takes two arguments and returns a list that contains all the integers from the first to the second, including the first but not including the second!

**Two other forms of range**

Rang(10) : start from zero

Range(1,10,3): specifies the space between successive values ( step size)

# Accessing elements of a list

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string  the bracket operator ([]).

Example:

```
>>>Numbers=[2,2,3,4]
>>>Number[0]
2
```

# List length

- The function *len* returns the length of a list.

It is a good idea to use this value as the upper bound of a loop instead of a constant. That way, if the size of the list changes, you won't have to go through the program changing all the loops; they will work correctly for any size list:

Example:

```
names= [“sam", “nat", “jack", “john"]
i = 0
while i < len(names):
  print names[i]
  i = i + 1
```

# List membership

- in is a Boolean operator that tests membership in a sequence.

Example:

```
>>>names= [‘sam’, ‘nat’, ‘jack’, ‘john’]
>>>’sam’ in names
True
>>>’carol’ in names
false
```

# Lists and for loops

The generalized syntax of a for loop is:

```
for VARIABLE in LIST:
      BODY
```

```
>>> numbers=[1,2,2,4,5,6]
>>> for numbers in numbers:
        print numbers
```

# List operations

**The + operator concatenates lists:**

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

**The * operator repeats a list a given number of times**

```
>>> [0] * 4
[0,0,0,0]
>>> [1,2,3]* 3
[1,2,3,1,2,3,1,2,3]
```

# List slices

- A segment of a list is called a **slice**.

Example:

```
>>> list=[‘a’, ’b’ ,’c ,’d ’ , ’e ’, ’ f,’]
>>> list  [1:3]
[’b’ , ‘c’]
>>> list[: 4]
[‘a’, ’b’ ,’c ,’d ’]
```

# List methods

- Python provide methods that operate on lists:

1. **append** : which add a new element to the end of a list.

example:

```
>>>t=[‘a’,’s’,’d’]
>>>t.append(‘x’)
>>> print t
t = [‘a’ , ‘s’, ‘d’, ‘x’]
```

**2. Extend**: takes a list as an argument and appends all of the elements.

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print t1
['a', 'b', 'c', 'd', 'e']
```

(3) Sort: arrange the elemnts of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print t
['a', 'b', 'c', 'd', 'e']
```

# List are mutable

- lists are mutable, which means we can change their elements.

immutable

An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered.
A new object has to be created if a different value has to be stored. They play an important role in places where a
constant hash value is needed,
for example as a key in a dictionary.

mutable

Mutable objects can change their value but keep their id()

iterable

An object capable of returning its members one at a time.
Examples of iterable include all sequence types (such as list, str, and tuple) and some non-sequence types like dict and file.