

# CS 1301 STUDY GUIDE

## Welcome to Programming!

- What is a Program?

A program is a sequence of instructions that specifies how to perform a computation. Though the details look different in different languages, a few basic instructions appear in just about any language:

Input, Output, Math, Conditional Execution and Repetition.

- Python is a **High Level Language** (i.e. a language that is designed to be easy for humans to read and write).
- Computers can only execute programs written in a **Low Level Language** (i.e. a language that is designed to be easy for a computer to execute; also called machine language or assembly language).
- So how does the computer understand the code I have written in a high level language?  
The **compiler** does that important job! A compiler is a special program (or a set of programs) that transforms the source code written in a high level programming language (source language) into the object code written in a low level computer language (target language) that the computer's processor can use.
- What is debugging?  
Programming errors are called bugs and the process of tracking them down and correcting them is called debugging. Three kinds of errors can occur in a program:
  1. **Syntax Error**: Syntax refers to the structure of a program and the rules about that structure. (Think of it as the grammar of a programming

language). A syntax error is an error in the syntax of a program that makes it impossible to parse and compile.

2. **Runtime Error**: An error that does not occur until the program has started to execute but that prevents the program from continuing.
3. **Semantic Error**: In this error, your program runs successfully but the meaning of the program (its semantics) is wrong and it does something other than what you asked it to do.

- **Comments**

A comment in a computer program is text that is intended only for the human reader – it is completely ignored by the interpreter.

In Python, “#” token starts a comment.

- What is an algorithm?

It is a set of specific steps for solving a category of problems.

## VALUES, VARIABLES and DATA TYPES

- A value is one of the fundamental things that a program manipulates. They are classified into **different classes** or **data types**.

- Some basic data types are:

1. int (integer): 1, 0, -2, 600, -78....
2. float (a number with a decimal point): 2.0, -3.0, 0.0, 67.5.....
3. bool (boolean): True, False, any expression that evaluates to a Boolean value e.g. `2 == 2`
4. NoneType: the default data type returned when a function stores nothing
5. str (String): Though it is not a basic data type, it is a very common data type. It is essentially a sequence of characters enclosed in quotes; even numbers when enclosed within quotes are classified as strings

- “**type**” function: To find out what “class” a value falls into use the “type” function.

```
>>>type("Hello World")
<type 'str'>
```

```
>>>type(15)
<type 'int'>
```

```
>>>type(True)
<type 'bool'>
```

```
>>>type(78.8)
<type 'float'>
```

```
>>>type("78.8")
<type 'str'>
```

- Strings in Python can be enclosed in either single quotes ('which can have double quotes inside them'), double quotes ("which can have single quotes inside them") and triple quotes (""" which can have either single or double quotes and can even span multiple lines""")

- **Type Conversion**

1. The **int** function can take a floating point number or a string and turn it into an int. For floating point numbers, it discards the decimal portion of the number – a process we call **truncation towards zero** on the number line.

```
>>>int(-3.4)
```

-3 (Note that the result is closer to zero)

```
>>>int(2.999)
```

2 (It does not round up to the closest integer)

```
>>>int("2345")
```

2345

```
>>>int("23 bottles")
```

**Invalid**

2. The type converter **float** can turn an integer, a float or a syntactically legal string into a float.

```
>>>float(17)
```

```
17.0
```

```
>>>float("123.45")
```

```
123.45
```

3. The type converter **str** turns its argument into a string.

```
>>>str(17)
```

```
"17"
```

- **Variable:** A variable is a name that refers to a value.

#### I. Variable Naming Rules:

- ✓ Variable names can be arbitrarily long.
- ✓ Variable names can contain letters, underscores and digits, but they have to begin with a letter or underscore.
- ✓ "case" matters when naming variables.

#### II. The assignment operation:

The assignment statement gives a value to a variable.

```
>>>message = "What's up?" ("=" is the assignment operator)
```

**Note:** The assignment statement binds a name (on the left) to a value (on the right). It evaluates the expression on the right first and then makes the assignment of that value to the variable.

```
>>>n = 5
```

```
>>>n = n+1
```

This statement may look weird if you think about it as a mathematical expression. How can "n" equal "n+1"? Understand that the "=" token is not the usual equal to symbol in math, here it assigns the value on the right to the variable on the left. So it evaluates the expression on the right first (i.e.  $n+1 = 5+1 = 6$ ) and now reassigns the value 6 to "n". So, after executing the two statements above "n" holds the value 6.

- **Mathematical Operation**

The order of evaluation depends on the "Rules of Precedence".

- a) Order: left to right  
b) Priority: PEMDAS (Parentheses > Exponentiation > Multiplication = Division = Modulo > Addition = Subtraction)

When two operations have the same precedence, then we evaluate from

Left to Right.

- Addition: +
- Subtraction: -
- Multiplication: \*
- Integer Division: // (answer will be an integer)

The integer division (or floor division) always gives a whole number as a result and in order to adjust, it always moves to the left of the number.

- Floating Point Division: / (answer will be a float)
- Power: \*\*
- Modulo: %

This operator works on integers and yields the remainder when one number is divided by another.

# Only when all the numbers used in the operation are integers, the result is an integer.

# If any of the numbers is float, the result will be a float.

HOWEVER, using integer division (//) with a float will yield the integer answer as a floating point rather than the correct decimal approximation. In other words, it will do integer division, but return it as a float (see example for better understanding).

e.g.:  $11/3 = 3.6$

$11/3.0 = 3.6$

$11//3 = 3$

$11//3.0 = 3.0$  \*instead of 3.6, integer division gives the integer answer (3) converted to a float (3.0)

- Operations on Strings

- ❖ In general, you cannot perform mathematical operations on strings, even if the strings look like numbers.

❖ The “+” operator does work with strings causing **concatenation** which means joining the two operands by linking them end to end.

❖ The “\*” operator also works on strings; it performs repetition.

```
>>> "Fun"*3
```

FunFunFun (One of the operands has to be a string and the other has to be an integer)

- Two more useful functions:

✚ **len** function: It returns the number of characters in a string.

```
>>> len("Happy Birthday")
```

14 (Note that it also counts the space as a character)

✚ **input** function: It is a built-in function in Python for getting input from the user.

```
>>> n = input("Enter your name")
```

On entering the test as a response, it gets returned from the input function as a string and in this case, is assigned to the variable “n”.

Even if you asked the user to enter their age, you would get back a string like “18” and it would be your job as a programmer to convert that string into a float or an int, using the converter functions.

## FUNCTIONS

- In Python, a function is a named sequence of statements that belong together and their primary purpose is to help us organize programs into chunks that match how we think about the problem.
- Syntax for defining a function:

```
def NAME (Parameter):  
    STATEMENTS
```

- There can be any number of statements inside a function but they have to be indented from “def”.

- There can be any number of parameters separated by commas and enclosed in parentheses. The parameter specifies what information, if any, we have to provide in order to use the new function.

```
def addTwoNums (x,y) :  
    sumOfNum = x + y  
    print ("The sum of the two given numbers is " + sumOfNum + ".")
```

- **Note that defining a new function does not make a function run.**  
To do that we need a **function call**. Function calls contain the name of the function being executed, followed by a list of values called **arguments** which are assigned to the parameters in the function definition.

```
>>>addTwoNums(5,10) ← Function call  
The sum of the two given numbers is 15. ← Output
```

- **Variables and Parameters are local.**  
When we create a local variable inside a function, it only exists inside the function definition and we cannot use it outside.  
In the example above, if we try to use **x** or **y** outside the function, we'll get an error.  
>>>a  
Name Error: name 'x' is not defined  
The variables **x** and **y** are local to **addTwoNums** function and are not visible outside the function.  
When execution of the function terminates, the local variables are destroyed.
- Functions that return a value are called fruitful functions.

```
def name():  
    name = input("Enter your name")  
    return name
```

Since this function returns a value stored in the variable “name”, it means that when I call this function it will return/store a value (in this case, a string) and to use that value further in my program I can assign it to a variable.

The return statement is followed by an expression that will be evaluated and returned to the caller as the “fruit” of calling this function.

```
>>>result = name()
>>>print("The result is " + result)
```

- The opposite of a fruitful function is a void function – one that is not executed for its resulting value but is executed because it does something useful.

```
def age():
    age = 18
    print(age)
```

This function displays the value stored in the variable “age” on the screen but does not store/return it.

- Even though void functions are not executed for their resulting value, Python always wants to return something. So, if the programmer doesn’t arrange to return a value, Python will automatically return the value “None”.

```
>>>result = age()
```

```
>>>print(result)
```

```
18
```

```
None
```

Huh? What happened? It should print just 18 why does it print None as well?

To understand this subtle point, we need to understand the difference between print and return (more detailed in next section).

**Explanation:** In Step 1 - The function `age()` is being called and the value it returns is being assigned to `result`. On calling `age()`, the function gets executed (recall that the right side is evaluated first) and on executing the

function age(), 18 gets printed on the screen. And since the function does not return anything, by default it returns `None`.

Now in Step 2 – The value stored in result (i.e. `None`) is printed on the screen and therefore, `None` gets printed on the second line.

## PRINT vs RETURN

<ul style="list-style-type: none"><li>• In Python 3, print is a function so when you call print you need to use parentheses. (like you call functions)</li><li>• It displays stuff to the screen. <pre>&gt;&gt;&gt;print("Hey")</pre>Hey (gets rid of the quotes when displaying the output)</li><li>• It can be used outside a function.</li><li>• It does not stop the execution of further statements if the function hits a print statement. As a result, we can print multiple things in a function.</li><li>• The print statement simply displays the argument to the screen, it does not return anything so the default value returned by the print statement is <code>None</code>.</li></ul>	<ul style="list-style-type: none"><li>• In Python 3, return is a statement so when you call return you may or may not use parentheses.</li><li>• It returns a value to a function (in a way, stores the value to a function)</li><li>• It cannot be used outside a function.</li><li>• Once the function hits the return statement, it stores the value returned back to the function and stops executing any further statements in the function definition. As a result, a function can return one thing.</li><li>• Since it stores the value returned by the function, the function call can be assigned to a variable.</li></ul>
--	---

- Example:

```
def tester():  
    print ("Hey")  
    return ("Bye")
```

```
>>>tester()
Hey #Notice no quotes when printed
'Bye' #Notice quotes when returned
>>>x = tester()
Hey #Notice, now only Hey is printed because the value returned is
assigned to a variable
>>>x
'Bye' #Here it returns the value stored in x, hence the quotes
>>>print(x)
Bye #Now the value stored in x, i.e. the value returned by the function is
printed hence no quotes
```

- Example:

```
def testing():
    print("1")
    return 2
    print("3")
```

```
>>>testing()
1 #Notice no quotes since it is printed
2 #Notice no quotes since the value returned is not a string, it is an int
#Notice 3 is not printed to the screen because once the function hit the return
statement, it stopped executing further statements
>>>y = testing()
1 #Notice now only 1 is printed because the value returned is assigned to
a variable
>>>print(y)
2 #Now the value stored by the function is printed to the screen
```

- Example:

```
def test():
    print("Hello World")
```

```
>>>result = test()
```

Hello World #Notice when this assignment is made, the right side is evaluated first and test() function is called as a result Hello World is printed to the screen

```
>>>print(result)
```

None #Now the value stored in the function is printed to the screen. Since this function does not contain any return statement it implicitly returns None and therefore None is printed to the screen

## CONDITIONALS

- A Boolean value is either True or False.
- A Boolean expression is an expression that evaluated to produce a result which is a Boolean value.

```
>>>2 == 3
```

False

- **Logical Operators**

There are three logical operators - and, or, not; they allow us to build more complex Boolean expressions from simpler Boolean expressions.

➤ and

	True	False
True	True	False
False	False	False

➤ or

	True	False
True	True	True
False	True	False

➤ not

not True = False

not False = True

- **Order and Priority**

Order: left to right

Priority: parentheses > not > and > or

- **Conditional Statement:** A statement that controls the flow of execution depending on some condition.

❖ if

```
if (aBooleanExpression):  
    statements
```

The statements will be executed when the boolean expression is True.

❖ elif

```
elif (aBooleanExpression):  
    statements
```

The statements will be executed only when the boolean expression is True and all other previous boolean expressions in the same if group are False.

❖ else

```
else:  
    statements
```

The statements will be executed when all the previous Boolean expressions in the same if group are False.

❖ if Family

- An “if” family always begins with an “if” statement, which is the only “if” statement in the family.
- It might have some “elif” statements, and at most one “else” statement at the end.
- All the conditional statements in the same “if” group must have same indentation.
- At most one block of statements will be executed in an “if” family. The first block which evaluates to True in an “if” family gets executed irrespective of whether the next blocks in the same family evaluate to True or not.
- Note: you do not need to have an “elif” or else statement accompanying every “if” statement.

- Example:  
def tester():  
 x = 3

```
y = 6
if (x > y) :           #if family 1
    print("Python is hard")
elif (x == y):       #still if family 1
    print("Python? The snake right?")
else:                #yup if family 1
    print("Python is cool")
if (x == 3):         #if family 2
    print("Hey")
elif (x == 4):       #if family 2
    print("Yo")
if (y == 3):         #if family 3
    print("Sup?")
else:                #if family 3
    print("I love CS 1301")
```

```
>>>tester()
Python is cool
Hey
I love CS 1301
```

- Short Circuit Evaluation:
  - The expression on the left of the “or” operator is evaluated first; if the result is True, Python does not (and need not) evaluate the expression on the right.
  - For the “and” operator, if the expression on the left yields False, Python does not evaluate the expression on the right.
  - This is called Short-Circuit evaluation.

## ITERATIONS

- **Iteration:** Repeated execution of a set of programming statements.
- **for loop**

The for loop processes each item in a sequence. Each item in turn is (re-) assigned to the loop variable and the body of the loop is executed.

```
for item in aSequence:  
    statements
```

The statements will be executed for every item in aSequence. For each execution, the variable identifier defined by the programmer (item) will be assigned to point to the next item in the sequence, and may be used to refer to it in the block of statements. At the end of the for loop, the variable will remain pointing at the last item in the sequence.

- Example:

```
>>>for i in ["Joe", "Sam", "Jacob"]:  
    print(i)  
  
Joe  
Sam  
Jacob
```

- while loop

```
while aBooleanExpression:  
    statements
```

The statements will be executed repeatedly until the Boolean expression becomes False.

- When using while loop, do not forget to initialize the counter before the loop and do increment/decrement in the loop.

Example:

```
>>>x = 0          #where to start  
>>>while x < 5:  #end as x becomes greater than or equal to 5  
    print("Hey")
```

```
x = x + 1 #this increment helps the loop reach end
```

This prints:

```
Hey      #x = 0
Hey      #x = 1
Hey      #x = 2
Hey      #x = 3
Hey      #x = 4
```

- The body of the loop should change the value of one or more variables so that eventually the condition becomes False and the loop terminates. Otherwise, the loop will repeat forever which is called an infinite loop.
- If the condition is False the first time we go to the loop, the statements in the body of the loop are never executed.
- **Choosing between for and while:**
  - ✓ Use a “for” loop if you know, before you start looping, the maximum number of times that you’ll need to execute the body. It is called **Definite Iteration** – we know ahead of time some definite bounds for what is needed.
  - ✓ By contrast, if you are required to repeat some computation until some condition is met and you cannot calculate in advance where (or if) this will happen, you’ll need a while loop. It is called **Indefinite Iteration** – we are not sure how many iterations we’ll need.
- The **break** statement:  
The break statement is used to immediately leave the body of the loop. The next statement to be executed is the first one after the body.

```
>>>for i in [12,16,17,24,29]
    if i % 2 == 1:
        break
    print(i)
print("done")
```

This prints:

```
12
16
done
```

- The `continue` statement:

This is a control flow statement that causes the program to immediately skip the processing of the rest of the body of the loop, for the current iteration.

But the loop still carries on running for its remaining iterations.

```
>>>for i in [12,16,17,24,29,30]
    if i % 2 == 1:
        continue          #If the number is odd
        print(i)          #Don't process it
    print("done")
```

This prints:

12

16

24

30

Done

- The `range` function

 `range(x)` – returns a list of numbers from 0 up to but not including x.

```
>>>range(4)
```

```
[0,1,2,3]
```

 `range(start,stop)` – returns a list of numbers from start up to but not including stop.

```
>>>range(5,11)
```

```
[5,6,7,8,9,10]
```

 `range(start,stop,step)` – returns a list of numbers from start up to but not including stop with an increment of step.

```
>>>range(5,11,2)
```

```
[5,7,9]
```

## BINARY CONVERSIONS

- Definition

- ❖ Binary (base 2):  $10011 = 1 \cdot 2^{**4} + 0 \cdot 2^{**3} + 0 \cdot 2^{**2} + 1 \cdot 2^{**1} + 1 \cdot 2^{**0}$
- ❖ Octal-decimal (base 8):  $725 = 7 \cdot 8^{**2} + 2 \cdot 8^{**1} + 5 \cdot 8^{**0}$
- ❖ Decimal (base 10):  $342 = 3 \cdot 10^{**2} + 4 \cdot 10^{**1} + 2 \cdot 10^{**0}$
- ❖ Hexadecimal (base 16):  $C1A = 12 \cdot 16^{**2} + 1 \cdot 16^{**1} + 10 \cdot 16^{**0}$

- General process:

Decimal  $\longleftrightarrow$  Binary  $\longleftrightarrow$  Octal-decimal / Hexadecimal

\*\* You can do octal-decimal / hexadecimal to decimal conversion by a general process (convert to binary version first and then to decimal).

- Decimal  $\longrightarrow$  Binary

Example: 156(base 10) to base 2

Step 1: Divide 156 by 2. Write down the result and the remainder as following:

Handwritten division process for converting 156 to binary:

$2 \overline{)156}$	Remainder:
$2 \overline{)78}$	0
$2 \overline{)39}$	0
$2 \overline{)19}$	1
$2 \overline{)9}$	1
$2 \overline{)4}$	1
$2 \overline{)2}$	0
$2 \overline{)1}$	0
	1

**$156_{10} = 10011100_2$**

Step 2: Keep doing this for all the quotients until you get 0 for quotient.

Step 3: Copy the remainders in a reverse order.

- Binary  $\longrightarrow$  Decimal

Example: 1011101 (base 2) to base 10

Step1: calculate by definition  $1011101$  (base 2) =  $1 \cdot 2^{**6} + 0 \cdot 2^{**5} + 1 \cdot 2^{**4} + 1 \cdot 2^{**3} + 1 \cdot 2^{**2} + 0 \cdot 2^{**1} + 1 \cdot 2^{**0}$

Result: 1011101 (base 2) is 93 (base 10)

## TRY AND EXCEPT

```
try:  
    block1  
except:  
    block2
```

block1 will be executed first. If an error occurs during block1's execution, the flow of execution will immediately jump to block2 (skipping any remaining statements in block1). If no error occurs, block2 will be skipped.

## COMPOUND DATA TYPES

### I. STRINGS

- Strings are made up of smaller strings each containing one character.
- **Index Operator**

The indexing operator (Python uses square brackets to enclose the index) selects a single character substring from a string.

```
>>>fruit = "orange"
```

```
>>>x = fruit[1]
```

```
>>>print(x)
```

r

The expression in the brackets is called the index, which specifies a member of an ordered collection.

- **Slices**  
A substring of a string is obtained by taking a slice.

➤ aVariable [:]

Slice everything

E.g. x = "python"

x[:] = "python"

- aVariable [ start : ]  
Slice everything after start  
E.g. x = "python"  
x[2:] = "thon"
- aVariable [ : stop ]  
Slice everything before stop  
E.g. x = "python"  
x[:4] = "pyth"
- aVariable [ start : stop ]  
Slice from start (included) to stop  
E.g. x = "python"  
x[1:5] = "ytho"
- aVariable [ start : stop : step ]  
Slice from start to stop with common difference step  
E.g. x = "python"  
x[1:5:2] = "yh"
- aVariable [ : : step ]  
Slice everything with common difference step  
E.g. x = "python"  
x[::-1] = "nohtyp"  
x[::2] = "pto"

\* All the starts are included and stops are excluded.

- **String Formatting**

Format operator: The % operator takes a format string and a tuple of values and generates a string by inserting the data values into the format string at the appropriate locations.

```
>>>print("My name is {} and I weigh {:.2f} pounds".format("Aaron",109.9622))
```

# 2 is the number of decimal places that you want the number to be rounded to.

My name is Aaron and I weigh 109.96 pounds

Types:

 d and i for decimal integer

 .nf for float with n decimal places ( .n is optional)

 s for string

- Strings are immutable which means you cannot change an existing string. The best you can do is create a new string that is a variation on the original and re-assign it to the same variable.

```
>>>x = "Hello World"
```

```
>>>y = "J" + x[1:]
```

```
>>>x = y
```

```
>>>print(x)
```

```
Jello World
```

- The "in" and "not in" operator

The "in" operator tests for membership. When both the arguments to "in" are strings, "in" checks whether the left argument is a substring of the right argument.

```
>>> "ap" in "apple"
```

```
True
```

```
>>> "pa" in "apple"
```

```
False #The order of characters must also match
```

The "not in" operator returns the logical opposite results of "in".

```
>>> "x" not in "apple"
```

```
True
```

- Useful Methods

➤ upper: It is a method that can be invoked on any string object and create a new string, in which all characters are in uppercase while the original string remains unchanged.

```
x = "hello"
```

```
y = x.upper()
```

```
>>>y
```

```
HELLO
```

```
>>>x
```

hello

- lower: It is a method that can be invoked on any string object and create a new string, in which all characters are in lowercase while the original string remains unchanged.

```
x = "HELLO"
```

```
y = x.lower()
```

```
>>>y
```

```
hello
```

```
>>>x
```

```
HELLO
```

- len: The len function when applied to a string, returns the number of characters in a string.

```
>>>fruit = "banana"
```

```
>>>len(fruit)
```

```
6
```

- find: Strings already have their own built-in find method. It returns the index position on which the string argument is present in the original string. It returns -1 if the original string does not contain the argument string.

```
>>> "banana".find("nan")
```

```
2
```

- split: It splits a single multi-word string into a list of individual words, based on a token string which is called the delimiter which acts as a boundary marker between substrings (by default it splits by spaces) and it removes all delimiters between the words.

```
>>>a = "I love CS 1301"
```

```
>>>b = a.split()
```

```
>>>b
```

```
["I", "love", "CS", "1301"]
```

#Notice no spaces in the words

```
>>>x = "So, what's up, I don't know."
```

```
>>>y = x.split(",")
```

```
>>>y
```

["So", " what's up", " I don't know."] #Note that when we change the delimiter to something other than the default value then spaces are also added to our words in the list.

## II. LISTS

- A list is an ordered collection of values. The values that make up the list are called its **elements** or its **items**.

```
aList = ["Bob", 12, [1, 2, 3]]
```

- Lists are similar to strings, which are ordered collections of characters, except that the **elements of a list can be of any type**.
- Lists and strings and other collections that maintain the order of their items are called **sequences**.
- Lists are mutable which means that you can change the elements of a list by a simple assignment.

Example,

```
aList = ["Hey", "How", 3]
```

```
aList[0] = 1
```

```
>>>aList
```

```
[1, "How", 3]
```

- A list within another list is said to be a nested list. You can do double indexing to access an element of the nested list.

Example,

```
aList = ["a", "b", [1, 2, 3, 4]]
```

```
aList[2][0] = "c"
```

```
>>>aList
```

```
["a", "b", ["c", 2, 3, 4]]
```

- **Accessing the elements**

You index through a list like you do for a string. Any expression evaluating to an integer can be used as an index.

If you try to access or assign to an element that does not exist, you get a runtime error.

- **List Operations**

- The “+” operator concatenates two lists to give a new list. The original lists are not affected.

```
>>>aList = [1,2,3]
>>>bList = [4,5,6]
>>>cList = aList + bList
>>>cList
[1,2,3,4,5,6]
```

- The “\*” operator repeats a list a given number of times to give a new list. The original list is not affected.

```
>>>aList = [0]
>>>bList = aList*4
>>>bList
[0,0,0,0]
```

- **List Slices**

The slice option lets us work with sublists.

```
>>>aList = ["a", "b", "c", "d"]
>>>bList = aList[1:3]
>>>bList
["b", "c"] #Note that again start is included and the end is excluded
```

- Lists are **mutable**.

Unlike strings, lists are mutable which means we can change their elements. Using the index operator on the left side of an assignment, it changes one of the elements in the list.

```
>>>aList = [1,2,3,4]
>>>aList[2] = 10
>>>aList
[1,2,10,4]
```

- **List Deletion**

```
>>>aList = ["a", "b", "c"]
>>>del aList[1] #Notice that we give it the index and not the element
>>>aList
["a", "c"]
```

- **List Methods:** The dot operator can be used to access built-in methods of list objects.
  - `(nameOfList).append` – It is a list method which adds the argument passed to it to the end of the list.
  - `(nameOfList).index(element)` – Finds the index of the first occurrence of element in the list.
  - `(nameOfList).reverse()` – Reverses the list.
  - `(nameOfList).sort()` – Sorts the list.
  - `(nameOfList).remove(element)` – Removes the first occurrence of the element in the list.
- **Alias vs Clone**
  - `aList = [1,2,3,4,5,6]`
  - `bList = aList`      *#This is an ALIAS*  
 Since variables refer to objects, if we assign one variable to another, both refer to the same object.  
 Here because the same list has two different names, i.e. `aList` and `bList`, we say that it is aliased.  
**Note that changes made with one alias affect the other.**  
**So, changing `bList` will also change `aList` because both of them refer to the same list object.**
  - `cList = aList[:]`      *#This is a CLONE (copy)*  
 When we want to modify a list and keep a copy of the original as well, we need to be able to make a copy of the contents of the list and not just the reference of the list. This process is called cloning. Now we are free to make changes to `cList` without worrying that we will inadvertently be changing `aList`.

### III. TUPLES

- Tuples are used for grouping data. A tuple can be used to group any number of items into a single compound value.
- **Syntactically, a tuple is a comma-separated sequence of values.** Though not necessary, it is conventional to enclose them in parentheses.  
`>>>aTup = 5,      #Correct`

```
>>>aTup = (5,)      #Correct
```

```
>>>aTup = (5)      #Wrong because there is no comma! Without a  
comma Python treats it as an integer inside parentheses.
```

```
>>>aTup = ("string", 2, False)    #Correct, a tuple can hold any data type
```

- **Tuples are immutable but if there is a list inside a tuple, it can be modified because the list inside the tuple is mutable. So be careful!**

Example,

```
aTup = ("String", 2, [1, 2, 3, 4])
```

```
aTup[0] = "Sam"      #Wrong!!
```

```
aTup[2][0] = "a"     #This works!!
```

```
>>>aTup
```

```
("String", 2, ["a", 2, 3, 4])
```

- **Tuple Operations**

- Slices

```
aTup = (2, "Hey", "You", 89)
```

```
bTup = aTup[1:3]      #Note the end index is excluded
```

```
>>>bTup
```

```
("Hey", "You")
```

- You can use the "+" operator to add tuples and assign it to a new variable; note that the original tuple is unchanged.

Example,

```
aTup = (2, 3, 4)
```

```
bTup = aTup + (5,)
```

```
>>>bTup
```

```
(2, 3, 4, 5)
```

## IV. DICTIONARY

- Dictionaries are Python's built-in **mapping** type. They map keys (**which can only be immutable type**) to values (**which can be of any type**).
- To create a dictionary:

- ❖ One way to create a dictionary is to start with the empty dictionary and add key - value pairs.

Example,

```
aDict = {}  
aDict["one"] = 1  
aDict["two"] = 2  
aDict["three"] = 3  
>>>aDict  
{'two':2, 'three':3, 'one':1}
```

#Note that the order in which the key - value pairs are present does not matter. Dictionaries are not sequences and therefore, it makes no sense to index into a dictionary because the elements don't have any fixed index in a dictionary.

**The values of a dictionary are accessed through keys.**

```
>>>aDict["one"]  
1
```

- ❖ Another way to create a dictionary is to provide a list of key-value pairs using the same syntax as previous output.

Example,

```
>>>aDict = {"one" : 1, "two" : 2, "three" : 3}
```

- Dictionaries are mutable.
- **Dictionary Operations and Methods**
  - The "del" statement removes a key-value pair from a dictionary.  

```
>>>del aDict["one"]
```
  - aDict.get(key, default)  
For 'key' key, returns value or default if key is not in dictionary.  
The default is usually None; also it is not required to give the second parameter (i.e. the default value).
  - aDict.has\_key(key)  
Returns True if key is in dictionary aDict, otherwise returns False.
  - aDict.keys()  
Returns a list of all the keys in the dictionary.
  - aDict.values()  
Returns a list of all the values in the dictionary.
  - aDict.items()

Returns a list of all the key-value pairs in the dictionary. Each item in the list is a tuple in format ( key, value ).

## FILE INPUT/OUTPUT

### • Reading

- `myFile = open ( filename, "r" )`
  - Open the file for reading purpose.
    - ❖ `myFile.readline()`  
Return the next line of the file.
    - ❖ `myFile.readlines()`  
Return all the lines in the file as a list of strings.
    - ❖ `myFile.read()`  
Return all of the contents of the file as a single string.
- \*Default mode of file I/O is "r"

### • Writing

- `myFile = open ( filename, "w" )`
  - Open the file for writing purpose.
    - ❖ `myFile.write ( aString )`  
Write a string to the file.
- \*If a file already exists with the same filename, the old file will be erased and substituted with the newly opened one.

### • Appending

- `myFile = open ( filename, "a" )`
- Open the file for appending purpose.
  - ❖ `myFile.write(aString)`  
Writes to an already existing file and adds to it.

### • Closing

- `myFile.close()`
- Close the file. You must do this every time!

## RECURSION

- **Recursion** means “defining something in terms of itself” usually at some smaller scale, perhaps multiple times, to achieve your objective. For example, we might say “A human is someone whose mother is a human being”, or “a directory is a structure that holds files and smaller directories”.
- **Three Elements of recursion:**
  - ✓ **Base Case:** Also known as terminating condition, is a conditional statement that stops recursion at some point and avoids infinite execution.
  - ✓ **Recursion Call:** Call the function itself inside the function.
  - ✓ **Recursive Step:** The process of approaching the base case. Usually increment or decrement.
- Recursion usually works as iteration. Do not use “for” loop or “while” loop together with recursion unless you understand exactly what you are doing.
- Example,  
The Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 134 ...  
If we number the terms of the sequence from 0, we can describe each term recursively as the sum of the previous two terms:

```
def fib(n):  
    if n <= 1:          #Base Case  
        return n  
    t = fib(n-1) + fib(n-2) #Recursion call; also n-1 & n-2 are Steps  
    return t
```

## FUNCTIONAL PROGRAMMING

- **Map**
  - map ( aFunction, aSequence )
  - map applies the function to all the elements in the sequence and returns a list that has the same length as the original sequence.
  - aFunction must take in one element.
  - map returns a list that has the same length as the original sequence, but the elements are modified.

➤ Note that the original sequence is not affected at all.

➤ Example,

```
>>>numbers = [1,2,3,4]
```

```
>>>newList = map(lamda i : i + 3, numbers)
```

```
>>>newList
```

```
[4,5,6,7]
```

- Reduce

➤ reduce ( aFunction, aSequence )

➤ reduce applies the function to the first two elements in the sequence first, and then repeatedly takes in the result that the function returns and the following element as parameters to reduce the length of the sequence, and finally returns one result.

➤ aFunction must take in two elements and return one element.

➤ reduce returns only one element.

➤ Note that the original sequence is not affected at all.

➤ Example,

```
>>>numbers = [1,2,3,4]
```

```
>>>sumOfNum = reduce(lambda i,j : i + j, numbers)
```

```
>>>sumOfNum
```

```
10
```

- Filter

➤ filter ( aFunction, aSequence )

➤ filter applies the function to every elements in the sequence and gets a Boolean. It keeps the element if the Boolean is True and removes the element if the Boolean is False.

➤ aFunction must take in one element and return a boolean.

➤ filter returns a new list that is shorter or has the same length as the original sequence, but each element is not modified.

➤ filter may return something other than a list. For example, if you filter a string it will return a string.

➤ Example,

```
>>>numbers = [1,2,3,4]
```

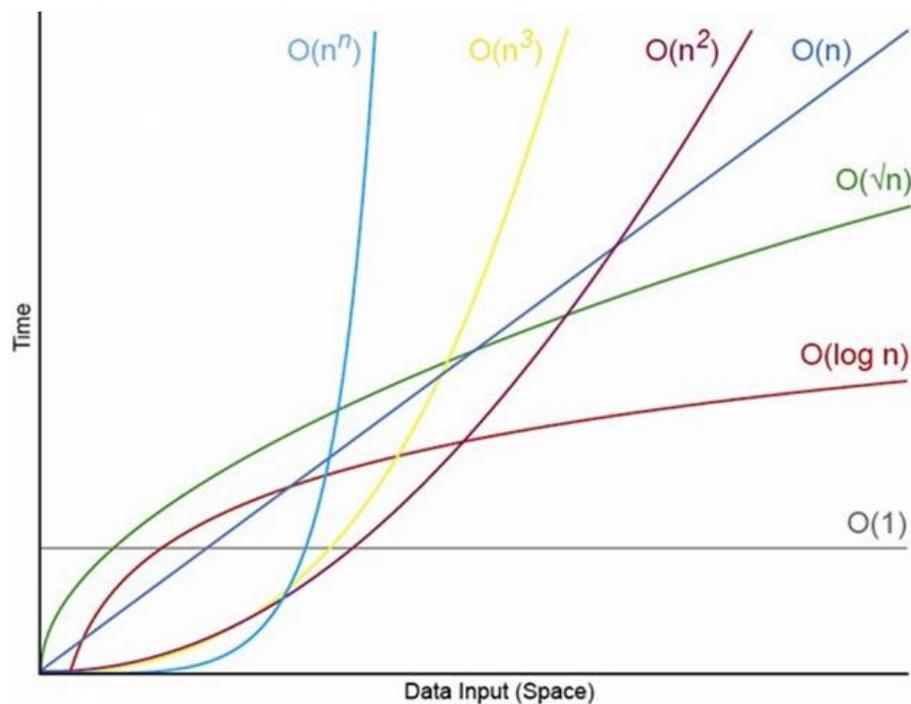
```
>>>newList = filter(lambda i : i%2 == 0, numbers)
```

```
>>>newList  
[2,4]
```

## SEARCHING AND SORTING

- **BigO Complexity**

BigO notation is used to describe how the work an algorithm does grows as the size of the input grows. In general, you ignore constants when calculating the BigO time complexity of an algorithm.



- **Search**

- ❖ **Linear Search**

- Search one by one
    - bigO:  $N$  (Examine each of the  $N$  elements in the list)

- ❖ **Binary Search**

- Compare the target value to the mid-point of the list. If the mid-point is not the target, divide the list in half and try again, searching only the correct half. Repeat until either there are no more elements to check, or until the target is found in the list
    - This algorithm can only be performed on sorted lists.
    - Example: Search 2 in list [1, 2, 3, 5, 8, 10, 15, 25]

- Round 1:  
Mid-point: 8  
 $2 < 8$   
New list: [1, 2, 3, 5]
- Round 2:  
Mid-point: 3  
 $2 < 3$   
New list: [1, 2]
- Round 3:  
Mid-point: 2  
 $2 = 2$   
Done
- bigO:  $\log N$  (  $\log_2 N$  rounds. 1 comparison each round.)

- Sort

- - ❖ Selection Sort

- Select the smallest number (if sort increasingly) in the list and append it to the result list.
- Example: Sort [3, 1, 4, 2] increasingly
- Round 1:  
Minimum: 1  
Result list: [1]  
New list: [3, 4, 2]
- Round 2:  
Minimum: 2  
Result list: [1, 2]  
New list: [3, 4]
- Round 3:  
Minimum: 3  
Result list: [1, 2, 3]  
New list: [4]
- Round 4:  
Minimum: 4  
Result list: [1, 2, 3, 4]

Done

- bigO:  $N^2$  ( N rounds. At most N comparisons each round to find out the smallest element.)

#### ❖ Insertion Sort

- Get the first element in the list. Insert it in the right place in the result list.

- Example: Sort [3, 1, 4, 2] increasingly

- Round 1:

Element: 3

Result list: [3]

New list: [1, 4, 2]

- Round 2:

Element: 1

Result list: [1, 3]

New list: [4, 2]

- Round 3:

Element: 4

Result list: [1, 3, 4]

New list: [2]

- Round 4:

Element: 2

Result list: [1, 2, 3, 4]

Done

- bigO:  $N^2$  (N rounds. At most N comparisons each round to find out the correct location.)

#### ❖ Bubble Sort

- Pass through the list of elements, and swap adjacent elements if they are not in the correct order. It must repeat the pass N-1 times to guarantee the entire list is sorted (If the smallest element is at the end of the list, it will take N-1 passes to swap it down to the front of the list.)

- Example: Sort [3, 1, 4, 2] increasingly

- Round 1:

[3, 1, 4, 2]  [1, 3, 4, 2]

[1, 3, 4, 2]  [1, 3, 4, 2]

[1, 3, 4, 2]  [1, 3, 2, 4]

- The last element in the list is guaranteed to be correct after the first round.

- Round 2:

[1, 3, 2, 4]  [1, 3, 2, 4]

[1, 3, 2, 4]  [1, 2, 3, 4]

- The last two elements in the list is guaranteed to be correct after the second round.

- Round 3:

[1, 2, 3, 4]  [1, 2, 3, 4]

- The last three elements in the list is guaranteed to be correct after the third round.

Done

- bigO:  $N^2$  (N-1 rounds. Each round takes N-1 comparisons. Hence we have  $(N-1)*(N-1)$ . Because we ignore constants, this is  $N^2$ )

### ❖ Merge Sort

- Divide the original list into small lists. Merge the small lists.

- Example: Sort [3, 1, 4, 2] increasingly

- Division stage:

- Round 1: [3, 1, 4, 2]  [3, 1] [4, 2]

- Round 2: [3, 1] [4, 2]  [3] [1] [4] [2]

- Merge stage:

- Round 1: [3] [1] [4] [2]  [1, 3] [2, 4]

- Round 2: [1, 3] [2, 4]  [1, 2, 3, 4]

- bigO:  $N \log N$  ( $\log(2N)$  rounds and at most N divisions each round in the division stage.  $\log(2N)$  rounds and at most N comparisons each round in the division stage.)

### ❖ Quick Sort

- Select element as pivot every round and compare the rest

elements to the pivot. Elements that are less than the pivot are collected into an unsorted list on the left of the pivot.

- Elements that are greater than or equal to the pivot are collected into an unsorted list to the right of the pivot.
- Repeat for the left and right hand collection of numbers until the size of each collection is one, at which point the entire list of numbers is correctly ordered.
- Example: Sort [3, 1, 4, 2] increasingly
- Round 1:  
Pivot: 4 (random choice)  
New list: [3, 1, 2, 4]
- Round 2:  
Pivot: 1 (random choice)  
New list: [1, 3, 2, 4]
- Round 3:  
Pivot: 2 (random choice)  
New list: [1, 2, 3, 4]
- bigO: depends on pivot choices  
N\*logN (average)  
N\*\*2 (maximum)  
(Average log (2N) rounds, at most N rounds. At most N comparisons each round.)