# A WORLD OF PYTHON

**What is a Program :**

* Sequence of instructions that specifies how to perform an operation.
* Like solving mathematical operation, replacing text in a file, processing image, playing a video file.
* Standard instruction in every program :
  + **Input :** from keyboard or any other device.
  + **Output :** Display some data on screen, save data in a file, send over a network etc.
  + **Math :** Perform basic mathematical operations.
  + **Conditional execution :** Check for conditions & run the code.
  + **Repetition :** Perform some action repeatedly with some variation.

**Running Python & first program :**

* Two types of python, python 2 and python 3.
  + Both are similar with few changes. You can learn either one of them and be able to catch up with the other.
  + These notes are based on python 3.
* Two ways to run a python program.
  + Use an online interpreter.
  + Download from python website for the machine.
* Once you start the interpreter : A program that reads and executes python code.
* Type in - >>> 1 + 1
  + Output : 2
* Traditionally writing a first program in any language is Hello World.
  + Type - >>> print(“ Hello, World ! ”)
  + Output : Hello, World !
  + The difference between python 2 and 3 lies here in python 2, you would write like this >>> print “ Hello,World ! “

**Arithmetic Operations :**

* Python provides operators which are special symbols that represent computation like addition and multiplication.
* >>> 5 + 5
  + Ouput : 10
* >>> 43 – 1
  + Output : 42
* >>> 3 \* 3
  + Output : 9
* >>> 10 / 5
  + Output : 2.0 ( Why 2.0 and not 2 ? Will be explained in the next section )
* >>> 2\*\*4 (This is an exponential operation mean 2 raised to 4)
  + Output : 16

**Value and Types :**

* A value is a basic thing that a program works with, like letter or number.
* Some values that we encountered before 42, 2.0, Hello,World ! etc.
* All these values have a type :
  + 42 is of type **integer**
  + 2.0 is of type **floating-point-number**
  + Hello,World ! is of type **String** letters that are strung along
* You can explicitly check the type of a value.
  + >>> type(2) – Ouput : <class ‘int’>
  + >>> type(2.0) – Output : <class ‘float’>
  + >>> type(‘Hello Worlds’) – Output : <class ‘str’>
  + The word “class” is used in the sense of category, a type of category of values.
  + Naturally 2 belongs to type **int**, 2.0 belongs to type **float** and ‘Hello World ’ belongs to type **str.**
  + Can we check for type(‘2’) or type(‘4.0’) they look like numbers but in quotation marks what would be their type ?

**Note :** Programming languages are formal language. That is they follow a certain syntax and rules that governs the language. Syntax rules come in two aspects one is Token & structure. Tokens are the basic element of the language such as words, numbers, chemical elements etc , the problem with 3 = 3$6 mathematical opr is that ‘$’ is an illegal token. The next thing is the token structure 3+ = 3 is illegal even though + and = are legal tokens is because they don’t form a proper structure, + doesn’t come before = for the operation.

**Note :** Programmers make mistake. For some reason the mistakes in the program are called **Bugs** and the process of tracking them down is called **Debugging.**

**Exercise :** whenever you are experimenting with things such as a simple print(“ Hello World ! ”) try and make mistake deliberately to see how the systems react, what messages they give and does the output remain same or different. It’s better to make mistakes deliberately in the start rather than making them later.

* Remove a parenthesis from a print statement.
* Check what happens when you type +2 or -2 or 2-----2 or 2+++2
* Check what happens when you type 02 or 0002 or 2 2 (value with no operator between them)
* Calculate the number of seconds in 42 min 46 seconds.
* How many miles in 10 kilometer. Hint: 1mile = 1.6 km

**Variables, Expression & Statements**

* One of the most powerful features of programming is to manipulate **variables**.
* A variable is a name that refers to a value.
* Assignment statements :
  + An **assignment statement** creates a new variable and assigns it value.
  + >>> message = ‘some random message container’
  + >>> n = 16
  + >>> pi = 3.141592653589793
  + The example above makes three assignments a string to a variable message, an integer to n and value of pi to a variable pi.
  + A common way to write variables with value on paper is to write their name with an arrow pointing to their value this is called a **State Diagram.** It shows what the variable holds in each state.
  + Message 🡪 ‘ some random message container’.
* **Variable names :**
  + Programmers generally choose variable names which are meaningful - they document what the variables are used for.
  + Variable name are arbitrarily long, they can contain bother letters and numbers but they have to begin with a letter.
  + Variable name can have uppercase letters but it is generally good idea to begin with a lowercase.
  + The underscore ‘ \_ ‘ character can appear in a variable name, they are often used for long variable names eg. my\_name, airspeed\_of\_plane\_boeing etc;
  + if you give variable an illegal name you will encounter a syntax error.
  + 76trombone = “parade” : syntax error
  + name@ = “hello” : syntax error illegal syntax
  + class = “ this is my class” : syntax error
  + The first two you can understand why there is an error but why ‘class’ ?
  + ‘class’ is one of the reserved python **keyword.** Python uses keywords to determine the structure of the program.
  + The keywords include :
    - and                 del             from             not           while
    - as                   elif             global           or             with
    - assert             else           if                   pass        yield
    - break              except       import           print
    - class               exec          in                  raise
    - continue          finally        is                  return
    - def                   for             lambda         try
    - in python3 exec is no longer a keyword by *nonlocal* is
* **Operators & operands and order of operations :**
  + Operators are special symbols that represent computation like addition, multiplication. The value the operators are applied to are called operands.
  + The operators like +,-,\*,/ and \*\* are addition,subtraction,multiplication and exponentiation
  + in python2 min = 59, min/60 will give result as : 0
  + in Python3 min/60 will give the result as 0.98333333
  + This is because python 2 on division does **floor** operation on the result whereas python 3 division is always a floating-point number. (recall 2.0)
  + When more than one operator appears in an expression the order of evaluation is according to rules of precedence. Remember **PEMDAS :**
    - **P**arenthesis : has the highest precedence can be used to evaluate in the order you want. Ex: 2\*(3+4) is 4, (1+1)\*\*(5-2) is 8.
    - **E**xponential : has the next higher precedence. so 2\*\*1+1 is 3 not 4, 3\*1\*\*3 is 3 not 27
    - **M**ultiplication & **D**ivision : have the same precedence.
    - **A**ddition & **S**ubtraction : have the same precedence.
* **String operation :** 
  + Generally we can’t perform mathematical operations on string even though they look like numbers .
  + ‘2’ – ‘1’ , ‘eggs’/ ‘omelet’ etc are illegal.
  + But there are two exceptions “ **+** “ and “ **\*** “.
  + The ‘+’ operator performs a string operation.
  + >>>First = ‘throat’
  + >>>Second = ‘ache’
  + >>>First + Second.
  + >>>throat ache
  + ‘\*’ also works on a string, it performs string repetition.
  + >>>’spam’\*3
  + >>’spamspamspam’
* **Comments :**
  + As problems get bigger and more complicated they get difficult to read.
  + Therefore it’s good to add notes explaining what the formal languages are doing.
  + These notes are called comments.
  + Eg:
    - #compute the percentage of time that has elapsed
    - Percentage = (min \* 100)/60 #you can also put a comment like this
  + Everything after a # is ignored and it has no effect on the execution of the program.
  + Comments are useful when they document non-obvious stuff, eg.
    - V = 5 #assign 5 to v this is a redundant comment
    - V= 5 # velocity in meters/second is a more useful comment
  + Good variables can
* **Debugging :**
  + Three kind of errors can occur in a program : Syntax error, Runtime error and Semantic error.
  + **Syntax Error :**
    - ‘Syntax’ refers to the rule of the program and the rules about the structure.
    - If there is a syntax error in your program then python displays an error message. Eg. (1 + 2) is legal but 1 + 2) is illegal because it should come with matching pairs.
    - At first you’ll make lot of syntax errors as and when you get experience you’ll make fewer of them.
  + **Runtime Error :**
    - The second type of error is called runtime because the error does not appear until after the program has started running.
    - They are also called exceptions because something exceptional has happened.
  + **Semantic Error :**
    - This type of error will not be indicated and the program will run.
    - But the program may not do as intended or the output will differ from what’s expected.
    - Basically the logic isn’t correct results to this error.

Exercise :

* Try making errors in the interactive mode :
  + We know what happens when n = 42 but what about 42 = n
  + How about x = y = 1
  + Some languages put semi-colon at the end of the statement, check what happens when you put ‘;’ and what about when you put a period ‘.’
  + Find the volume of the sphere with radius r = 5. Vol = (4/3)(pi)r\*\*(3)
  + Suppose the cover price of the book is $24.95, but a bookstore gets a 40% discount.The shipping cost is $3 for the first copy and 70 cents for each additional copy. What is the total wholesale cost for 60 copies.

**Functions**

* Function is a named sequence of statements that perform a computation.
* When you define a function you specify the name and sequence of statements.
* Later you call the function by name.
* **Function calls :**
  + We have already seen one type of function call : type(42)
  + >>> <class ‘int’>
  + The name of the function is ‘**type**’ and the expression in the function is called an ‘**argument**’.
  + It is common to say that function “*takes*” an argument and “*returns*” a result.
  + The result is also called “return value”
  + Python provides function that can convert one type of value into another.
    - **int(‘32’) >>> 32**
    - the int function takes an argument and converts it into an integer.
    - But it also has exceptions
    - int(‘hello’) >>> valueError : invalid literal for int()
    - int can also convert floating point values to int
    - **int(3.99999) >>> 3**
    - **int(-2.898738) >>> -2**
    - it doesn’t round of the number but slashes the fraction part.
    - We can also convert an integer into floating-point number
    - **float(42) >>> 42.0**
    - **float(‘3.14521’) >>>3.14521**
    - And we have a function that converts a value into string
    - **str(32) >>> ‘32’**
    - **str(3.14159) >>> ‘3.14159’**
* **Math functions :**
  + Python provides a math module to perform a series of mathematical computation.
  + A module is a file that contains a collection of related functions.
  + Before we can use the module we have to import it with an **import statement**.
  + >>> import math
  + <module ‘math’ (built-in)>
  + To access one of it’s functions and variable, you need to specify the name of the module and specify the name of the function or variable, seprated by a **dot** (also know as period ‘.’). This format is called **dot notation.**
  + Ratio = signal\_power / noise\_power
  + Decibels = 10\*math.log10(Ratio)
  + Radians = 0.7
  + Height = math.sin(Radians)
  + Math.sqrt(4)
  + >>>2.0
  + Math.pi
* **Function Definition and uses :**
  + So far we used functions that were built by python.
  + We can add our own functions as well.
  + A **function definition** specifies the name of the new function and sequence of statement that run when the function is called.
  + Ex. def print\_lyrics():
    - print(“Stairway to Heaven !”)
    - print(“I am on a Highway to Hell !”)
  + **def** is a keyword that indicates that this is a function.
  + The name of the function is print\_lyrics
  + The rules for the function name is same as variables, names,numbers and ‘\_ ‘ is legal but the name cannot start with a number and avoid having same name as a variable name.
  + The empty ‘()’ after the function name indicates that the function doesn’t take any arguments.
  + The first line of the function definition is called **header** and the rest is called **body.**
  + The function header ends with a ‘ **:** ’ **colon** and the body has to be **indented.**
  + By convention the indention is always 4 spaces.
  + Defining a function creates a function object which is of type *function.*
  + The syntax for calling a function is same as for built in function.
  + print\_lyrics()
  + Once you have defined a function you can use it inside another function.
  + Def repeat\_lyrics():
    - Print\_lyrics()
    - Print\_lyrics()
  + **Remember** that the function definition always needs to come before a function call otherwise python throws an error.
  + Flow of execution is top to bottom line by line.
* **Parameters and arguments :**
  + As we saw earlier with math.sqrt(4) takes in 4 as an argument to find the square root of the number 4.
  + We can define our own functions that takes in arguments.
  + Inside the function the arguments are assigned to a variable called **parameters.**
  + Example : def print\_twice(bruce):
    - print(bruce)
    - print(bruce)
  + The function assigns the argument to a parameter named *bruce*. So when the function is called it prints the value of the parameter twice.
  + Variable and parameters inside a function are **local.**
  + What we mean by that is they exist only within and for the function they were defined in and gets destroyed when the function is terminated.
  + **Useful to draw stack diagrams where each function is represented by a frame.**
  + Two types of function :
    - Functions that return value. (call them **Fruitful functions**)
    - Functions that don’t return any value (call them **Void functions**)
* **Why do we need functions in the first place ?**
  + Makes your program easier to read and debug.
  + Makes program small by eliminating repetitive code.
  + Well designed function can be useful for many other programs.
  + Easier to maintain and debug, if changes are needed then it can be done only at one place.
  + Encapsulation (wraps all the code in a single place) and Generalization (makes a general code that can be used to create anthing).

**Exercise :**

1. Write a function name *right\_justify* that takes a string named *S* as a parameter and prints the string with enough leading spaces so that the last letter of the string is in column 70 of the display:
   1. right\_justify(‘Tommy’)

Tommy

*Hint: use string repetition and concatenation, also python provides a built in function* ***len*** *that returns the length of the string like len(‘Tommy’) returns 5*

1. A function object is a value that can be assigned to a variable or pass as an argument. Example do\_twice is a function that takes a function as an argument and calls it twice
   1. def do\_twice(f):
   2. f()
   3. f()
   4. Here is an example that uses do\_twice to call a function named print\_spam() twice
   5. def print\_spam():
   6. print(‘spam’)
   7. do\_twice(print\_spam)
   8. Test this in your program.

**Conditionals and Recursion**

Main focus of this topic is the **if** statement which executes different code depending on the state of the program. But before we begin with that two new operators : floor division and modulus.

* **Floor Division and Modulus :** 
  + The floor division ‘ **//** ‘ divides two numbers and rounds down to an integer.
  + For example we want to know the run time of a movie with 105 min.
  + Minute = 105
  + Minute / 60
  + >>> 1.75
  + But what if we don’t want the fraction part since hour cannot be represented in decimal point.
  + Minute // 60
  + >>> 1
  + The **modulus %** operator divides the two numbers and returns the remainder.
  + Remainder = minute % 60
  + Remainder
  + 45
  + The modulus operator is more useful then it seems because you can check if one number say x is divisible by another number say y.
  + X%Y will be 0 if x is divisible by y.
  + You can also do x%10 to get the right most digit or x%100 to get the last two digits.
* **Boolean expression and Logical operators :**
  + A Boolean expression is an expression that is either true or false.
  + Example : the following operator “**==**” is used to compare two operands and result in true if they are equal else false.
  + 5 == 5 >>> true
  + 5 == 6 >>> false
  + True and false are special values and they belong to **type bool.**
  + The ‘ == ’ operator is one of the **relational operator** others are :
    - X != Y # X is not equal to Y
    - X > Y # X is greater than Y
    - X < Y # X is less than Y
    - X >= Y # X is greater than equal to Y
    - X <= Y # X is less than equal to Y
  + **Note :** The most common mistake programmers do is use ‘=’ instead of ‘==’. You need to understand that ‘=’ is an assignment and ‘==’ is comaprision.
  + There are three **Logical operators**
    - and, or and not and they are equivalent to their English meaning.
  + For example : x > 0 and x < 10 means x is greater than 0 *and* x is less than 10
  + N%2 == 0 or N%3 == 0 means if either of both are true then N is divisible by 2 or 3 or both.
  + Finally *not* operator is used to negate the Boolean expression. So for example : not(x > y) is true when x > y is false or x is less than y.
* **Conditional, Alternative and Chained Execution :**
  + In order to write useful programs we always need the ability to check for conditions and change the programs behavior accordingly.
  + **Conditional statements** gives us that ability. Here is a simple form of conditional statement :
  + if x > 0:
  + print(‘x is positive’)
  + The Boolean expression after if is called a **condition**. If it’s true the indented statement runs else nothing happens.
  + The *if* structure is similar to the function structure, followed by indented body.
  + There is no limit on the number of statements that can be present under the if statement.
  + These type of statements are called **Compound Statements.**
  + Sometime it’s useful to have statements with no body (placeholder for the code that will be written in the future) in that case we can use *pass* :
  + If x < 0:
  + pass #TODO: need to handle negative values.
  + Second type of if statement is the “alternative execution” in which there are two possible conditions.
  + if X%2 == 0:
  + Print(‘x is even’)
  + else:
  + print(‘x is odd’)
  + Sometimes there are more than 2 possibilities we need “chained conditionals” for that kind of scenario :
  + If x < y:
  + Print(‘x is less than y’)
  + elif x > y:
  + Print(‘x is greater than y’)
  + else:
  + Print(‘x and y are equal’)
  + ‘**elif** ’ is an abbreviation for else if. There is no limit on the number of elif statements.
  + We can also have a nested if conditions, that is if condition can run inside another if condition for example we could have written the above program like this as well :
  + If x == y
  + Print(‘x and y are equal’)
  + else:
  + if x < y:
  + Print(‘x is less than y’)
  + else:
  + Print(‘x is greater than y’)
  + Nested conditions become difficult to read as they become more comples, therefore it’s better to avoid them altogether.
* **Recursion :**
  + It is legal to call one function from another it also legal to call the function itself.
  + It might not be obvious why this is a good thing but it is one of the most magical thing a program can do. For example :
  + Def countdown(n):
  + If n < = 0:
  + Print(‘Blast off !’)
  + Else:
  + Print(n)
  + countdown(n-1)
  + If n is 0 or negative then it prints “Blastoff !”
  + Otherwise it ouputs n and calls itself with n-1 as an argument.
  + Let’s see what happens when we call the function as: countdown(3)
    - The execution of the countdown begins with n=3 since it’s greater than 0 it’ll output 3 then send 2 as a argument to itself.
      * Then execution of countdown begins with n=2, since 2 > 0 it’ll output 2 and send n-1 i.e 1 to itself
        + Then …… 1>0, outputs 1 and sends 0 to itself as an argument.

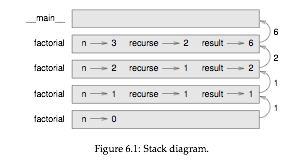
It get’s 0 so it’ll print “blastoff !” because equal to or less than 0.

* + - * + Retruns to n=1
      * Returns to n=2
    - Returns to n=3
  + Finally returns to the \_\_main\_\_.
  + A function that calls itself is **recursive** and the process of executing it is called **recursion.**
  + Let’s take another example:
  + def print\_n(s,n):
    - if n <= 0:
      * return
    - print(s)
    - print(s,n-1)
  + if n<=0 then return statement exits the function.
  + For simple example like these it’s better to use for loop but as you progress we’ll see examples that are hard to write in for loop but easier with recursion.
  + Infinite recursion : sometimes there isn’t a base case and the program runs on forever making recursive calls, this is generally not a good idea and is known as infinite recursion.
* **Keyboard input :**
  + We haven’t used any input from the user in the previous python program.
  + Python provides a built-in function called *input* that stops the program and waits for a user to type something.
  + When the user presses *Return* or *Enter* the program resumes and the input returns what the user has typed as a string.
  + ! in python 2 same function is called raw\_input()
  + Text = input()
  + What are you waiting for ?
  + >>> Text
  + >>> What are you waiting for ?
  + Before getting the input it’s a good idea to print a prompt telling user what to type. Input takes a prompt as an argument.
  + Text = input(“What is your name ? \n”)
  + >>> What is you name ?
  + My name is Tommy
  + >>> Text
  + >>> My name is Tommy
  + If you expect the user to type in an integer then you can convert the return value to an integer.
  + Number = input(“Enter a number : ”)
  + >> Enter a number : 42
  + >>> int(Number)
  + >>> 42

Exercise :

1. The Fermat’s theorem states that there are no +ve integers a,b and c such that
   1. A\*\*n + B\*\*n = C\*\*n for any value of n greater than 2.
   2. Write a function *check\_fermat* that takes in a,b,c,n as arguments and checks to see if the fermat’s theorem holds. The program should print “Holy Smokes ! Fermat was wrong” otherwise “No that does not work !”
   3. Write a function that prompts user to input value of a,b,c and n pass it to *check\_fermat* function.
2. If you are given three sticks there is a probability that you cannot form a triangle with the sticks. For example if length of one stick is 12 inches and length of other two sticks is 1 inch then you cannot form a triangle.
   1. Write a function named is\_triangle that takes 3 integers as arguments and prints “Yes” or “No” depending on whether you can or cannot form a triangle. (Hint : The sum of two sides of triangle is always greater than the third side, also check for the condition if the sum of two sides of a triangle is equal to the third that triangle is known as “Degenerate” triangle )
   2. Write a function that takes users input for three sides.

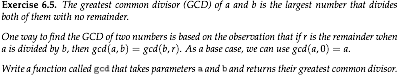
**Returning Functions or Fruitful functions**

* Many python functions have some sort of returning value. Like the math functions we encountered before.
* **Return Value :**
  + Calling a function usually generates a returning value which we usually assign to a variable or use as a part of an expression.
  + e = math.exp(1.0)
  + height = radius \* math.sin(radians)
  + The functions we wrote previously are know as void functions that is they don’t return an value.
  + Let’s see our first example about an area function which returns which returns the area of the circle
  + def area(radius):
    - a = math.pi\*radius\*\*2
    - return a
  + We could also return a expression like *return math.pi\*radius\*\*2* ,but it’s better to send a temporary variable as it makes it easier for debugging
  + Sometimes it’s useful to have multiple return statements on each conditional branch e.g.
  + def absolute\_value(x):
    - if x < 0:
    - return -x
    - else:
    - return x
  + Since the return statements are in alternate conditions only one runs.
  + Note that as soon as a *return* statement returns the function terminates without executing subsequent statements.
  + The code which is unreachable after a return statement is known as **dead code.**
  + Small Exercise : write a function that takes in 2 values x and y, returns 1 id x>y, 0 if x==y or -1 if x<y.
* **Incremental development :**
  + As we write larger functions we’ll find ourself doing more debugging.
  + To deal with debugging we might want to try a process called **Incremental development.**
  + The goal here is to avoid long debugging sessions by adding and testing only a small amount of code at a time.
  + The key aspects of this process are :-
    - Start with a working program and make small incremental changes. At any point if there is an error, we will have a good idea where it is.
    - use variables to hold intermediate values so we can display them and check them.
    - Once the program is working you may want to remove some of the scaffolding or consolidate multiple statements into compounded expression.
  + For example we want to find the distance between 2 points given the coordinates (x1,y1) and (x2,y2) using the Pythagorean theorem :
  + distance = math.sqrt((x2 - x1)\*\*2 +(y2-y1)\*\*2)
  + First we consider what are the inputs(parameters) and what is the output(return value).
  + def distance(x1,y1,x2,y2):
  + return 0.0
  + Obviously this version does not compute the distance but it’s syntactically correct.
  + Test the new function by calling with sample arguments >>> distance(1,4,6,8)
  + 0.0
  + Once we have confirmed that it’s syntactically correct we can start adding more code:
  + def distance(x1,y1,x2,y2):
  + dx = x2 - x1
  + dy = y2 - y1
  + print(‘dx is ’, dx)
  + print(‘dy is ’, dy)
  + return 0.0
  + If the function is working then it should display dx is 3 and dy is 4. If so then we know that function is getting the right argument and performing the right function otherwise only few lines to check for the error.
  + def distance(x1,y1,x2,y2):
  + dx = x2 - x1
  + dy = y2 - y1
  + dsquared = dx\*\*2 + dy\*\*2
  + print(dsquared)
  + return 0.0
  + Again you would check the result and see if it’s correct and finally write the last step.
  + result = math.sqrt(dsquared)
  + return result
  + remove the print part from the final function as only result should be sent back as return. If it works then you are basically done.
  + Code like the print statements are called **scaffolding** because they are useful for debugging but not in the final production.
  + Making temporary function is useful for debugging but once the program is working make the function more concise. (calling functions within functions)
  + Example returning a boolean function :
  + def is\_divisible(x,y):
  + if x%y == 0:
  + return true
  + else:
  + return false
  + it is important to give boolean functions a name that sounds like a question.
  + The result of the == operator is boolean therefore we can make the function more concise by :
  + def is\_divisible(x,y):
  + return x%y == 0
  + Boolean functions are often used in conditional statements.
  + if is\_divisible(x,y):
  + print(‘x is divisible by y’)
  + More Recursion :
  + calculating the factorial -
  + 0! = 1
  + n! = n(n-1)!
  + *def factorial(n):*
  + *if n == 0:*
  + *return 1*
  + *else:*
  + *recurse = factorial(n-1)*
  + *result = n \* recurse*
  + *return result*
  + The flow of the program is similar to the flow of the program countdown.

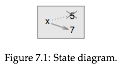
Exercise :

1. The Ackermann function A(m,n) is defined as :

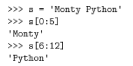
* Write a function that evaluates Ackermann function, use your function to evaluate A(3,4).

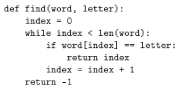
1. A palindrome is a word or a number that is spelled the same backward and forward. Example like “noon” and “redivider”. Recursively a word is a palindrome if the First and Last word is same and the middle word is a palindrome. The following functions take in string as an argument and return the first,last and middle word.
   1. def first(word):
   2. return word[0]
   3. def last(word):
   4. return word[-1]
   5. def middle(word):
   6. return word[1:-1]
      1. Type these functions into a file named palindrome.py , see what happens when you send two letters in the middle function ? One letter ? or an ‘ ’ empty string.
      2. Write a function named is\_palindrome that returns true if the word is palindrome else false.

**Iteration**

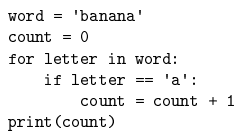
* The ability to run a block of statements repeatedly.
* We have seen a for loop statement, we saw recursion we will dwell into While statement out here.
* Before we get further in While statement, let’s begin with exploring a bit into reassignment.
* It s legal to make more than one assignment to one variable. A new assignment makes an existing variable refer to new value.
* >>> x = 5
* >>> x
* 5
* >>> x = 7
* >>> x
* 7
* This is what the state diagram of the reassignment looks like.
* >>> a = 5
* >>> b = a #now b is equal to a
* >>> a = 3 # a and b are no longer equal
* >>> b
* 5
* A common kind of reassignment is an **update** where the new value of the variable depends on the old.
* >>> x = x + 1 #get current value of x, add one and update the value of x
* If you try to update a variable without initialising it then python gives an error.
* We need to initialise first if we want to update :
* x = 0
* x = x + 1
* Updating a value by one is called an increment and reducing by one is called decrement.
* **While statement :**
  + Computers are often used to automate repetitive task. A repetition in a computer program is called **iteration**.
  + Example of while statement using the countdown example :
  + def countdown(n):
  + while n > 0:
  + print(n)
  + n = n - 1
  + print(‘ Blastoff !’)
  + Formally, here is the flow of execution for a while statement :
    - Determine whether the condition is true or false
    - If false exit the while statement and continue the execution of the next statement.
    - if the condition is true run the body and go back to step 1.
  + This type of flow is called a loop because at the end of statement it goes back to first step.
  + The body of the loop should change the variable of the condition so the loop eventually ends otherwise it’ll keep on running, and that is know as **infinite loop.**
* **Break statement :** 
  + Sometimes we don’t know when to exit/end the loop until we get half way through the body. In this case we use the *break* statement to exit the loop.
  + Suppose you want to take input from the user until they are done :
  + while true: #since ‘true’ cond always remains true this is an infinite loop
  + line = input(‘>’)
  + if line = ‘done’:
  + break
  + print(line)
  + print(Done!)

**Strings**

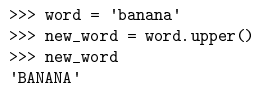
* A string is a sequence of letters
* A sequence means it’s an ordered collection of other values.
* We can access the characters one at a time with the use of brackets ‘[ ]’
* >>> fruit = 'banana'  
  >>> letter = fruit[1]
* The second statement takes letter 1 from *fruit* and assigns it to *letter*.
* The expression in the bracket is called **index** and indicates which character in the sequence we want.
* But you might be surprised as you might not get what you expect :
* >>> letter
* ‘a’
* For most people the first letter of ‘banana’ is ‘b’ but for the computer scientists, the index is an offset from the beginning of the string and the offset for the 1st letter is **zero**
* >>> letter = fruit[0]  
  >>> letter  
  'b'
* So ‘b’ is the 0th letter of banana, ‘a’ is the 1st letter of banana and ‘n’ is the 2nd letter of banana.
* As an index you can use expression that contains variables and operators.
* >>> i = 1  
  >>> fruit[i]  
  'a'  
  >>> fruit[i+1]  
  'n'
* But the expression has to be an integer otherwise you get an error.
* >>> letter = fruit[1.5]  
  TypeError: string indices must be integers
* **Length of string :** 
  + *‘len’* is a built in function that returns the number of characters in a string.
  + >>> fruit = 'banana'  
    >>> len(fruit)  
    6
  + To get the last letter of the string you might be tempted to try something like this :
  + >>> length = len(fruit)  
    >>> last = fruit[length]  
    IndexError: string index out of range
  + Because as stated earlier the index in ‘banana’ starts from 0 – 5, therefore there is no letter in ‘banana’ at index 6. So to get the last letter you have to subtract 1 from length.
  + >>> last = fruit[length-1]  
    >>> last  
    'a
  + Or you can count from backward from the end of the string. The expression *fruit[-1]* yields the last letter ‘a’, *fruit[-2]* yields 2nd last letter and so on.
* **Traversal with *for* loop :**
  + Often many computation requires accessing single character one at a time and doing something with that character till the end of character in string.
  + This pattern of processing is called **traversal.**
  + One way to write a traversal is a *while* loop.
  + while loop:  
    index = 0  
    while index < len(fruit):  
    letter = fruit[index]  
    print(letter)  
    index = index + 1
  + Mini Exercise: print characters line by line backwards.
  + Another way to write traversal is with **for** loop :
  + for letter in fruit:  
     print(letter)
  + Each time through the loop, the next character at
* **String slice :**
  + A segment of string is called **slice.** Selecting a slice is similar to selecting a character :
  + The operator [n:m] returns the part of the string from ‘nth’ character to ‘mth’ character, including the first but excluding the last.
  + If we omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string.
  + fruit = ‘banana’
  + >>> fruit[:3]
  + ‘ban’
  + >>>fruit[3:]
  + ‘ana’
  + if the first index is greater than or equal to the 2nd index then the result is an empty string.
  + >>> fruit[3:3]
  + ‘ ’
  + what about fruit[:] ?
* **Strings are immutable :** 
  + Immutable means something that cannot be changed.
  + The ‘object’ in this case is string and the ‘item’ is what we tried to assign.
  + The string being immutable it cannot be changed the best we can do is create new string variable.
  + Concatenates new letter onto slice of ‘greeting’ string.
* **Searching :**
  + The patter of computation - traversing a sequence and returning when we find what we are looking for is called **Searching.**
  + Example this :



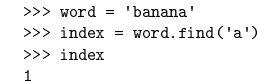
* + In this example we can see a *return* statement inside a loop. If the letter is found then the function breaks out of the loop and returns immediately.
* **Looping and counting :**
  + The following program counts the number of times a letter appears in the word :



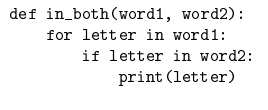
* + This program demonstrates another pattern of computation called **counter.**
  + The variable count is initialised to 0 and incremented each time the letter ‘a’ is found.
  + Mini exercise : Encapsulate the count in a function *count* which takes two arguments a word and the letter to be counted.
* **String Methods :**
  + Strings provides methods that provide a variety of useful operation.
  + **Method** is similar to function – it takes arguments and returns a value – but syntax is different.
  + It follows something called a dot notation.
  + For example let us consider a method *upper().* Takes a string and returns a new string in all upper case letters.
  + Instead of function syntax *upper(word)* it uses syntax *word.upper().*



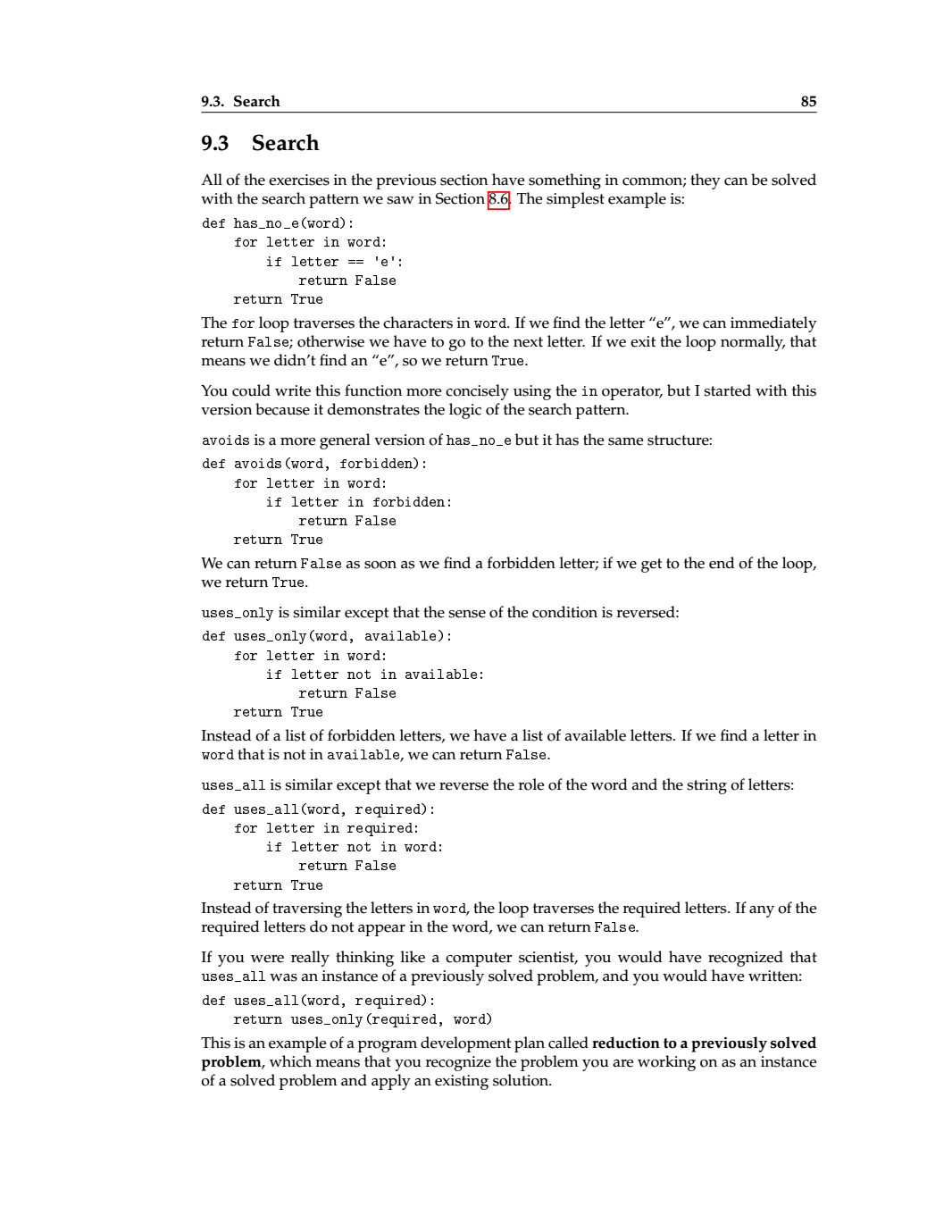
* + A method is called an **Invocation** that is we are invoking the method upper on word.
  + A dot notation syntax is in standard manner : The name of the variable/string/object on which the method is applied on(word), ’ . ‘ ,The name of the method (upper), The empty parenthesis indicating that it does not take any argument or also could be a nonempty parenthesis.
  + There also exist a method called **find()** that works like the function we saw above :



* + The find function is more general it can also find substrings :
  + >>> word.find(‘na’)
  + 2
  + But it can also take a second argument, which states which index to start looking from :
  + >>> word.find(‘na’,3)
  + 4
  + It also takes an optional argument which states where to look till.
  + >>> word.find(‘n’,1,2)
  + -1 #returns -1 since n does not appear between 1(inclusive) to 2
* **The** *in***operator :**
  + The **in** a Boolean operator that takes in 2 strings and returns *True* if the first appears as a substring the second string.
  + >>> ‘a’ in ‘banana’
  + True
  + >>> ‘seed’ in ‘banana’
  + False
  + For example the following function prints all the letters from word1 that appear in word2:

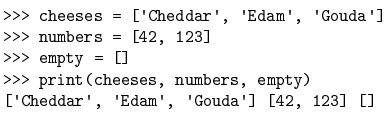


* + For some well-chosen variable names, Python sometimes reads like English. For example you could read the loop like :
  + “ for (each) letter in ( the first) word, if (the) letter (appears) in (the second) word, print (the) letter ”
* **String Comparison :**
  + The relational operators work on strings as well.
  + To see if two strings are equal –
  + If word == ‘banana’:
  + print(“ Banana’s ”);
  + Other relational operators are useful for putting words in Alphabetical order :
  + If word < ‘banana’:
  + Print(‘your word’ + word + ‘comes before banana’)
  + If word > ‘banana’:
  + Print(‘your word’ + word + ‘comes after banana’)
  + else:
  + print(‘Banana’sss’)
  + **Note:** Python does not handle uppercase and lowercase like normal people, All uppercase letters come before lowercase.

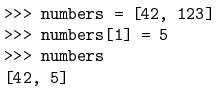


**Lists**

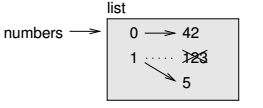
* A list is a sequence of objects.
* Like a string a list is a sequence of values.
* Except that values in a string are characters whereas the values in the list can be anything.
* The values in the list are called **elements** or sometimes **items.**
* There are several ways to create list but the easiest one is to enclose in brackets ([ and ]).
* [10,20,30,40,50] or [‘hello’,’crunchy frog’,’crap shit’]
* Here the first example contains a list of integers and the second contains list of string. But a list can contain values of multiple types too.
* Example the following list contains integers, float, string and another list ( A list within a list means **nested**)
* [‘spam’, 7, 2.0, [10, 20]]
* A list that contains no elements is called an empty list [ ]



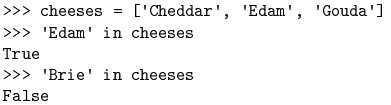
* **List are mutable :** 
  + Unlike strings the List is mutable .



* + Look at the stack diagram below :



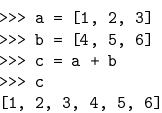
* + List indices work same way as string indices :
    - Any integer expression can be used as index eg. number[1].
    - If you try to read or write an element that does not exist, you get an *IndexError*.
    - If the index has a –ve value it counts from backwards of the list.
    - The **in** operator also works with list.



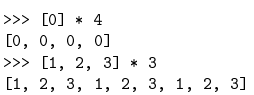
* **Traversing a list :**
  + The most common way to traverse through list is by using for loop.
  + The syntax is same as for strings – for cheese in cheese: …
  + This works well only if we want to read the elements of the list. If we want to update as well then a common way is to combine built-in function **len**  and **range.**



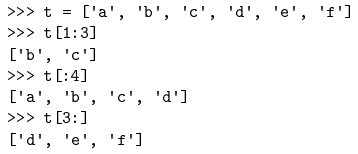
* + len – returns the length of the list or number of elements in the list.
  + range – returns a list of indices from 0 to n-1 where n is the length of list.
  + **Note :** Although a list can contain a nested list it is still counted as a single element.
* **List operations :** 
  + A ‘ **+** ‘ operator concatenates two lists :



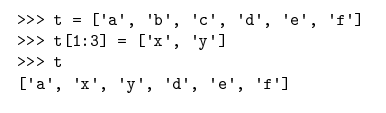
* + The ‘ **\*** ‘ operator returns number of times the list



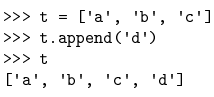
* + The slice operator also works on list :



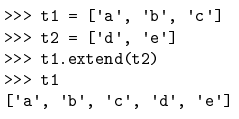
* + Since list are mutable it’s often useful to make a copy by using [:] before you make changes.
  + A slice operator on the left side of an element can update multiple elements :



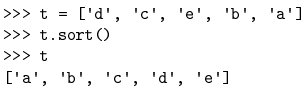
* **List Methods :** 
  + Python provides method that operate on list.
  + For example **append** function adds a new element in the end list.



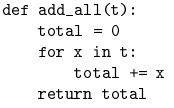
* + **extend** takes a list as an argument and appends the elements in the end of the list.



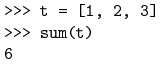
* + This example leaves t2 unmodified.
  + **sort** arranges the elements in the list from low to high :



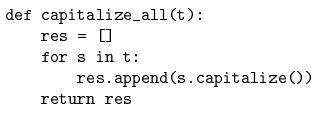
* + Most list methods only perform function and return void as in no value.
* **Map, Filter and Reduce :**
  + To add up elements in the list we can write a program like this :



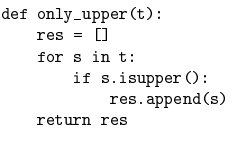
* + But adding up the list is such a common operation that python provides a built in function **sum()**



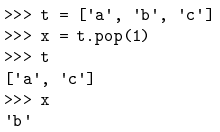
* + An operation like this which combine a sequence of elements into a single value is sometime called **Reduce.**
  + Sometimes we want to traverse one list while building another :
  + For example the following function takes a list of strings and returns a new list with all strings capitalized :



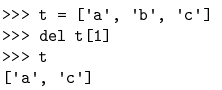
* + An operation like capitalize\_all is sometimes called **map** because it “maps” a function onto each element in the sequence.
  + Another common operation is to select some of the elements from the list and return a sublist.
  + For example the following function takes a list and returns a list that contains only upper case strings.



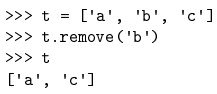
* + An operation like only\_upper is called a **filter** because it selects some of the elements and filters out the rest.
  + Most common list of operation can be expressed in combination of map, reduce and filter.
* **Deleting Elements :**
  + There are several ways to delete an element from a list.
  + If you know the index of the element then you can use **pop :**



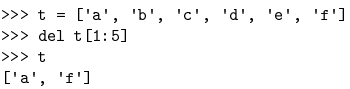
* + pops modifies the list and returns the element that was deleted.
  + If you don’t provide an argument it deletes and return the last element.
  + If we don’t need the removed value, we can use *del* operator :



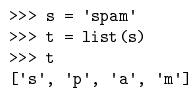
* + If we know the element we want to remove but not the index, we can use remove :



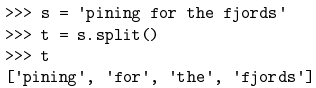
* + The return value from remove is *none*.
  + To remove more than one element we can use slice with del operator :



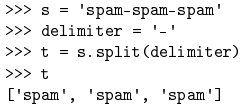
* **Lists and Strings :** 
  + A string is a sequence of char and list is a sequence of values, but a list of char is not the same as string.
  + To convert string into list of char we can use built in function *list.*



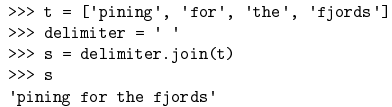
* + The list function breaks a string into individual char. To break a string into words use *split* method :



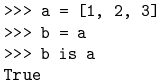
* + An optional argument **delimiter** specifies which character to use for word boundaries.



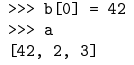
* + *Join* is inverse of split, it takes a list of strings and concatenates elements, since join is a string method we need to invoke on a delimiter.



* **Aliasing :** 
  + If a refers to an object and we assign b = a, then both variables refers to same object :



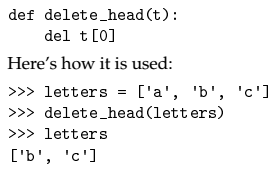
* + ‘**is**’ operator is used is used to compare equality between two objects.
  + The association of variable with an object is called **reference.**
  + In the above example there are two reference to the same object.
  + An object with more than one reference has more than one name, so we can say that object is **Aliased.**
  + If the Aliased object is mutable then changes made with one alias affect the other.



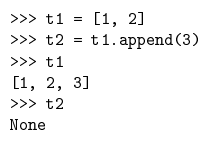
* + Avoid aliasing with mutable and for immutable objects aliasing is not much of a problem.



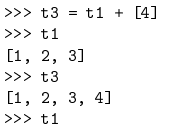
* **List Arguments :**
  + When we pass a list to a function, the function gets a reference to the list.
  + Any changes made to the list by the function, the caller can see the changes.
  + Example delete\_head function removes an item from the list.



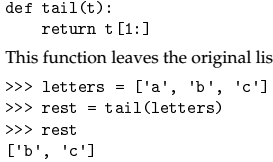
* + The parameter **t** and variable **letter** are aliases for the same object.
  + It is important to distinguish between and operation that modifies the list and one that creates a new one.
  + Example **append** method modifies the list where ‘**+**’ creates a new list.
  + The + operator creates a new list and leaves the original list unchanged.



* + + operator returns new list :



* + Alternative to write a function that modifies and returns a new list ;



**Dictionaries**

* A dictionary is a mapping, it’s like a list but more general.
* In a list indices have to be numbers but in a dictionary it can be anything.
* A dictionary contains collection of indices which are called **keys** and a collection of values.
* Each key is associated with a value.
* The association of key and a value is called a **key-value-pair** or sometimes an **item.**
* The function *dict()* creates a new dictionary with no items. It’s a built-in function therefore it cannot be used as a variable name.



* { } 🡨 This bracket represents an empty dictionary. To add an item in the dictionary you can use square brackets [ ].
* >>> eng2sp[‘one’] = ‘uno’
* >>> eng2sp
* {‘one’ : ‘uno’}
* We can even create a dictionary for multiple items at once :



* But if we print eng2sp on our machine the result will be different –



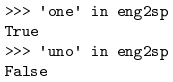
* The order does not matter as items in dict are not arranged in integer indices, it searches for key indices to retrieve the item.



* If the key isn’t in the dictionary then we get an error.
* The len function also works on dictionary, it gets the number of key-value pairs.



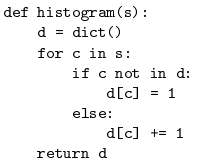
* The ‘in’ operator works in dictionary as well it checks if a ‘key’ is available in dictionary not the value.



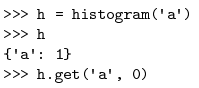
* To check if something appears as a value in the dictionary, we can use the method values, which returns the collection of values and then use the ‘in’ operator.



* **Dictionary as a collection of counter :**
  + Suppose we are given a string and we would like to count how many times does the string appear. There are several ways to do so :
    - Create a list of 26 variable and traverse the string and increment the count of each variable probably using a chained condition.
    - Create a list with 26 elements, convert each character into number (built-in func ord) use the number as an index into the list and increment the appropriate counter.
    - We could create a dictionary with character as keys and counters as the corresponding values. Every time we encounter a character we add to the dictionary if the character is not present already.
  + All of them do the same computation but differ in the implementation of the computation.
  + An **implementation** is a way of performing computation. Some implementation are better than others. Example in the dictionary implementation we don’t need to know the letters ahead of time, we only make room for the letters that do appear on the fly.



* + Dictionaries have a method called *get*. The method takes in a key and a value. If the key exists in the dictionary then it returns the corresponding value else it returns the default value.

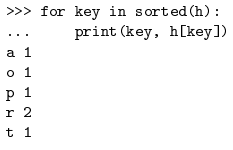




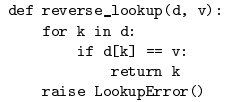
* **Looping and dictionary :** 
  + If we use a **for** loop for dictionary it traverses the key.
  + Example :



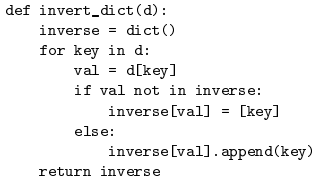
* + The keys are in no particular order therefore we can use *sorted* a built in function.



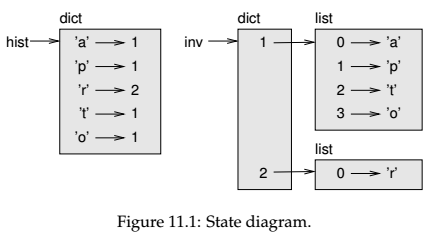
* **Reverse lookup :**
  + Given a dictionary d and a key k. It’s easy to find a corresponding value v = d[k]. This operation is called a **lookup.**
  + But what if we have *v* and want to find k. There might be 2 problems :
    - There can be same values for different keys.
    - No simple syntax for reverse lookup, will have to do search first.
  + Example here is a function that takes a value and returns the first key.



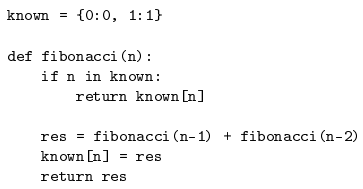
* + The **raise statement** causes an exception. Which causes a *lookup error* which is a built in function.
  + A reverse lookup is much slower than forward lookup, can cause performance problem.
* **Dictionaries and List :** 
  + List can also appear as value in a dictionary.
  + Example here is a function that maps frequencies with letters. As there might be many letters with same frequency :



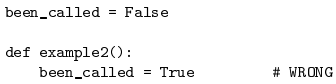
* + Here is a state diagram showing both Histogram and inverse



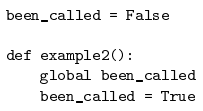
* + List can be values in a dictionary but they cannot be keys in a dictionary.
  + Since dictionary is made from hash tables the key needs to be hashable.
  + A hash is a function that takes in a value and returns a integer, dictionary makes use of this integer for finding key-value pairs.
  + A previously computed value that is stored for later use is called a **Memo.**
  + Here is an example for a faster version of computing Fibonacci sequence :



* + In this example ‘known’ is created in a special frame known as \_\_main\_\_
  + Variables created in \_\_main\_\_ is called **Global** variable because they can be accessed by any function.
  + It’s common to use Boolean flags as global variable.
  + But they cannot be changed inside a function (as variables reside locally in function ) example :



* + To reassign a global variable inside a function we have to declare global variable before.



* + This means when the function see ‘global’ it refers to global variable and not create a new local variable.

***Tupels***

* Tuples are immutable.
* The value can be any type and the index are integers.
* Syntactically a tuple is comma separated list of values :



* To create a tuple with a single element, we have to include final comma.



* A value in parenthesis is not a tuple.



* Another way of creating tuple is using the built-in function ‘*tuple* ‘.



* The relational operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).
* >>> (0, 1, 2) < (0, 3, 4)
* True
* >>> (0, 1, 2000000) < (0, 3, 4)
* True
* **Tuple assignment :** 
  + For swapping the values of two variable.
  + For the usual swapping method we use a temp variable, the tuple assignment is more elegant.
  + a , b = b , a
  + The left side is tuple variable and the right side is tuple value the left side variable is mapped respectively to right side value.
  + The number of variable on the left and the number of values on the right has to be same.



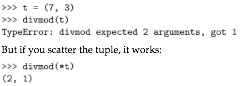
* + More generally **right** side can be any sequence (**list, string or tuple**)**.** Example splitting an email in to two domains.



* + The return element is list of two elements, monty is assigned to uname and python.org is assigned to domain.
* **Tuple as return value :**
  + A function can return only 1 value, but if the value is a tuple then it’s like returning multiple value.
  + for example dividing 2 values x and y and returning it’s remainder and quotient. Calculating x/y and x%y separately is inefficient, do it at the same time.
  + rem,quo = divmod(7,3)
  + print(rem) # 1
  + print(quo) #2
* **Variable Length argument :**
  + A parameter that begins with **\* gather** arguments into a tuple.

def printall(\*args):

print(args)

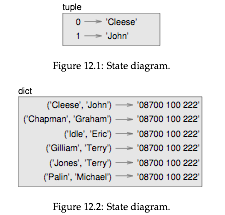
* + If you have a sequence of values and you want to pass it to function as multiple arguments, use **\*** operator.
* **Lists and Tuple :**
  + **Zip** is a built-in function,
    - takes in two or more sequence
    - returns a list of tuples where each tuple contains one element of each sequence

>>> s = 'abc'

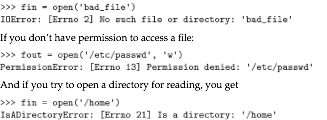
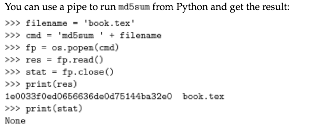
>>> t = [0, 1, 2]

>>> zip(s, t)

<zip object at 0x7f7d0a9e7c48>

* + - The result is a **Zip Object** that can be iterated through pair.
    - >>>for pair in zip(s,t):
      * >>> print(pair)
  + Zip is an **Iterator** object, they are like list but unlike the list you can’t access elements using Index.
  + We can make a zip object as list to use list features . **list(zip(s,t))**
  + If sequences are not same in zip() args then it considers the length of the shorter sequence
  + Combining zip, tuple and for is useful for traversing two or more sequence at a time.
* **Dictionary and tuple :**
  + Dictionary has a function called **items()** that return a sequence of tuples, where each tuple is a key-value pair.
  + You can also vice versa, create a dictionary from a list of Tuple. ‘dict(t)’
  + combining *zip* with *dict* yields a concise way of creating a dictionary.
    - d = dict(zip(‘abc’,range(3)))
  + dictionary method **update** also takes in key, value pair of list of tuple and adds in the existing dictionary.
  + It is common to use **Tuple** as a key value in dictionary.
  + We can use tuples as **keys** in dictionary (since we can’t use list)
  + directory[lastname,fistname] = number
  + for lstname,fstname in directory:
  + print(lstname,fistname,directory[lstname,fstname])
* **Sequences of Sequences :** 
  + Which sequence of sequences to use when ?
  + Strings
    1. They are more limited than most other sequences because they only contain characters.
    2. They are immutable
    3. If we want to change any characters in string we would have to use list (more efficient then creating a new string)
  + List
    1. List are more common then tuples since they are **Mutable**.
    2. There are cases where we would use Tuples instead.
  + Tuples
    1. Like *return* statements, it syntactically simpler to create a tuple then a list.
    2. If we want to use sequence as a dictionary **key,** we have to use an immutable type like Strings or tuple.
    3. If we are passing a sequence as an argument to a function, tuples reduces the potential for unexpected behaviour due to aliasing.
  + since tuple re immutable, they do not provide method such as *sort & reverse* since they modify the existing list.
  + Python provides a built in function called ***sorted & reversed*** that takes in any sequence and returns a new list.

**FILES**

* **Reading and Writing of files :**
  + A text file is a sequence of characters stored on a physical medium.
  + To write into a file we need to open it with mode ‘w’ as a second parameter : **fout = open(‘some.txt’,’w’)**
  + If the file already exist then it opens the file, clears the data and starts new, if the file doesn't exist then it creates a new one.
  + *open* returns a file object that provides method for working with files.
  + line1 = “my name is Nishank”
  + **fout.write(line1)**
  + 18
  + it returns number of words written to the file.
  + When we are done writing we should close the file.
  + **fout.close()** ! If we don’t close the file it gets closed for us.
* **Format operator :** 
  + The argument of the *write* function has to be a string.
  + So if we need to put any other value we need to convert it to string using **str**
  + **x = 52**
  + **fout.write(str(x))**
  + An alternative to use is the ‘**%**’ format operator.
  + Syntax : **<format string>, <second operator who is formatted >**
  + **‘%d’ % x** —> %d Means the second operator should be formatted as decimal integer in the string. Result is ’52’.
  + If there is more than one format sequence in the string, the second argument has to be a Tuple.
  + Example : ‘%d’ -> Decimal Integer, ‘%g’ -> Floating-point number, ‘%s’ -> String
  + **“In my %d years I have seen %g %s” %(10,4.0,’camels’)**
  + The number of elements in the tuple has to match the number of format sequence in the string.
  + The element type of the format string has to be same as the value.
  + A more powerful alternative is the string formatter method.
* **Filenames and paths :** 
  + Files are organised into directories (also called folders).
  + Every running program has a current working directory.
  + The **os** module provides function for accessing the file path and names.
  + The string like ‘/home/dinsdale’ that identifies a file or directory is called **Path**.
  + A simple path like ‘memo.txt’ is also considered a path but it’s known as **relative path** because it’s related to the current directory.
  + A path that does not depend on the current directory is called **absolute path**.
  + **os.path.abspath(‘memo.txt’)** #gets the absolute path of the file in cwd
  + **os.path.exists(‘memo.txt’)** #checks if the file exist returns true or false.
  + **os.path.isdir(‘/home/Mac-NB’)** #checks wether it’s a directory or not.
  + **os.path.isfile(‘memo.txt’)** #checks if its’s a file or not
  + **os.listdir(cwd)** #returns list of files in the directory
  + **os.path.join(dirname,name)** #joins the dirname and name and returns a complete path.
* **Catching Exceptions :**
  + A lot of things can go wrong when try to read and write into a file.
  + Like The file might not exist, or we may not have permission to access a file,or we are trying to open a directory instead etc.
  + To avoid these errors we can use os.path.isdir or os.path.exsists etc.
  + But we would have to check for lot of conditions, it’s better to use **Exception** syntax which is exactly what a **try** statement does.
  + Python starts by executing try statement, if everything is alright it skips the except: clause otherwise if there is a exception then it executes the except clause.
* **Pipes :**
  + Any programs that can be launched from our command-line shell of OS can also be launched from python using a **pipe object.**
  + Example using ls -l to open the list of files in long format using **os.popen** :
  + cmd = ‘ls - l’
  + fp = os.popen(cmd) #popen is deprecated now , use ***subprocess*** instead.
  + **fp** is a file object, we can read the output from the file object.
  + we can either read the output from a file object one line at a time using **readline** or all at a time using **read**. **res = fp.read()**
  + when we are done we close the pipe like a file . **stat = fp.close()**
* **Writing Modules :**
  + Any file that contains python code can be imported as a module.
  + Programs that will be imported as modules will follow this idiom :
  + if \_\_name\_\_ == ‘\_\_main\_\_’:
  + print(linecount(‘wc.py’))
  + Where \_\_name\_\_ is built in variable that is set when the program starts.
  + If the program is running as a script then ‘\_\_name\_\_’ has value ‘\_\_main\_\_’ ; in this case; the test code runs.
  + Otherwise if the module is being imported, the test code is skipped.

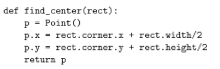
**CLASSES AND OBJECTS**

* A programmer defined type is called **class**.
* The syntax starts with *class <classname>:*
* *you can define variables and methods in the class.*
* Defining a class named *point* creates a **class object**
* <class ‘\_\_main\_\_.point’>
* Since point is defined at the top level it’s full name is \_\_main\_\_.point
* To create a point object we’ll call point as a function, class object is a factory for creating objects.
* Creating a new object is called **Initiation** and the object is an **Instance** of the class.

**Attributes**

* Assign values to an instance using dot notation.
* blank.x = 3.0
* blank.y = 4.0
* This behaviour is similar to selecting variables from modules like math.pi from math module, or string.whitespace.
* These elements are called attributes.
* You can read the value of the attribute or assign is to another variable.
* >>> blank.y
* 4.0
* >>> x = blank.x
* >>> x
* 3.0
* we can use dot notation as a part of any expression e.g.. dist = math.sqrt(blank.x\*\*2 + blank.y\*\*2)
* we can also pass an instance as an argument over a function.
* Inside the function is an alias of the instance so if the function modifies the instance then it also modifies the attributes of the original instance.
* We can have a attribute that is an instance of another class.

**Instance as a Return Value**

* Function can return instances.
* Eg. find\_center takes a Rectangle as an argument and returns a point, that contains the coordinates of the centre of the rectangle.

**Objects are mutable**

* The state of the object can be changed by making an assignment to one of it’s attributes.
* For example we can change the width and height of the rectangle object without changing it’s position.
* In addition we can modify the attributes eg .. box.height = box.height + 50

**Copying**

* Aliasing can make program difficult to read because changes in one place can have unexpected effects in another place.
* Copying an object is an alternative to aliasing. the *copy module* contains a function called **copy** that can duplicate any object :
* p1 = point
* p1.x = 2
* p1.y = 3
* import copy
* p2 = copy.copy(p1)
* p1 and p2 contain the same data but they are not the same point.
* For instance the default behaviour of ‘==‘ is same as ‘is’ operator even if the attributes have same value.
* p1 == p2 yields false just like p1 is p2
* The operation of copy is called **shallow copy**
* This is error prone as invoking a method within by the copy instance modifies both the instances.
* Fortunately we have a **deep copy** method that copies not only the object but also the objects it refers to.
* This operation is called **deep copy**. Eg. box3 = copy.deepcopy(box).

**Debuggin**

* type(p) #to check the type of the instance should be of <class >
* isInstance(p,point) #checks wether p is an instance of class point
* *True*
* If you aren’t sure wether the object has the a particular attribute, use the built in function hasattr(p,’x’) —> True

**CLASSES AND FUNCTIONS**