Computational Thinking and programming

**Session 1**

Overview of this course

* Learn computational modes of thinking.
* Master the art of computational problem solving.
* Make computer do what you want them to do.

In this course, I will learn in a particular language python but more importantly I will be learning how to think computationally to think algorithmically to think like a computer scientist and what is that mean it means when given a challenge how can I get the computer to solve the problem for me, how best can I describe the stages that I want to use to get this done in such a manner that I don’t have to do it but the computer can take care of it this is the notion of the computational thinking of algorithmic thinking.

Now this means I want the computer to work for me , it really means the computer will be a servant for me here. To do that we will cover the range of topics.

Range of topics

1. Represent knowledge with **Data structure**

* we want computer to compute something for us infer(deduce/conclude) some new knowledge for us that means we have to think about how we represent that knowledge and we will do that with particular things inside the machine called Data structures

1. **Iteration and recursion** as computational metaphors

* We want to infer(deduce/conclude) new information or find new information; we are going to see there are standard tools to make that happen things called iteration and recursion.

1. **Abstraction** of procedure and data types

* A big part of what we want to do inside the computer is to have it to be able to deal with things in a manner that we can see and understand, that where we are going to use the notion of abstraction to capture elements and then treat them as if they were primitives and reuse them

1. **Organize and modularization** systems using object classes and methods

* Abstraction then leads naturally to the idea of the modularization creating modules tokens elements that we can stitch together to come up with solutions to problems in interesting ways.

1. Different classes of **Algorithms**, searching and sorting

* Once we started learning how to build algorithms to think algorithmic we are going to see that there are standard classes of algorithms we are going to use those standard classes for common problems like searching and sorting

1. **Complexity of algorithm**

* We’re going to see as well that different algorithms have **different costs** and we want to see how to use that to reason about the expense of doing something and better ways of finding a solution to different problems

Computers can do only 2 things they perform the calculations and remember the results. now the question is what type calculation the computer does. Turns out these are very primitive calculation addition, multiplication, subtraction, division and simple logic operations compare true and false values and making decisions with that.

Now with the help of these simple operations we can define new more complex operations or calculations things we create. Then you can abstract, encapsulate them and treat them as primitive operations. But to start with a computer simply performs a lot of those calculations those simple primitive calculations very quickly is that enough might be , if that’s the case then we really don’t have to do a lot in terms of computation and I want to give you couple of examples to show you why even with the speed of modern computers you need to be able to think carefully cleverly algorithmically. Here are 2 examples for us to think about.

Ex 1: I would like to find a piece of information on the web something you do every day with a search engine

* Suppose if we are searching through the web how much could we do if we are using simple calculations, her is a computation I performed there are about 1 trillion pages right now on the world wide web.
* On average there are about 100 words on a page
* For the sake of argument lets assume if we want to find a word on a page its going to take us 10 operations to try and find them whether that word is on that page or not. We will see what these 10 operations are later.
* If I brute force try and search everything on the web to see if I can find the thing I am looking for its only going to take about 7.3 days to find something. Probably I wouldn’t wait that long even though we have very fast machine using simple calculation will not be enough.

Ex 2: I might want to play chess or have your computer or have the computer play the chess for me.

* An expert will tell you this on an average of 35 moves for every setting on the chess board until you get to the end game
* suppose you want to look ahead six moves in order to try and decide what you want to do in order to beat your opponent that say’s you got 1.8 billion boards that you need to check
* if its going to take you for example a hundred operations for every choice its going to take you about 30 mins to decide each move probably too slow.
* This is simply a way of saying that even with fast computers we need cleverness we need algorithmic thinking to take those simple computations and turn them into something more powerful.

That said good algorithmic design is going to be crucial and its one of the skill you will be learning throughout this course.

What about storage?

Ans: Storage in the machine why don’t we computer everything once and store it in the system and then just look it up. Lets go back to chess imagine I just want to look at all the possible chess games and store them away so that when I am in any move I will just know what I want to do in order to get to a winning position. Experts would suggest that there are something on the order of 10 to the 123 different possible chess games. And about 10 to the 80th atoms in the observable universe there’s no way that we can store all that information

We cant use brute force or pre-compute we need to be clever about how we come up with the solutions. Even with that are there going to be limits to the computation even if we build clever algorithms, in fact some of them suggest that there are still some limitations what a computer can do some problems are still at least at the moment to complex even with cleaver algorithms to come up with solutions fast enough.

Some problems are just fundamentally impossible to compute and the classic one from computer science is called the *touring halting problem and it simply says that if I want to write a piece of code a program that could take as the input from any other program and tell me whether it will always work, where it will always stop with an answer* it turns out you simply cant compute that in all cases, so there are going to be limits for computations but not to worry its going to be a lot of things we can do and that’s what we are going to do throughout this course.

**Session 2**

Types of Knowledge: what’s the knowledge that its going to use to do that computation and that actually leads to interesting discussion because we can divide knowledge into two types we will call declarative knowledge and imperative knowledge

Declarative Knowledge: these are statement of facts, statements there is candy taped to the underside of one chair. doesn’t tell you anything about how to find it, doesn’t tell you where to look for it. It is simply a state of fact.

Imperative knowledge: an imperative piece of information simply says there’s candy taped to the underside of a chair, Imperative knowledge is a recipe or ‘how-to-computation’ and this gives a sequence of steps to find a solution. its a recipe , follow the steps to find the candy in the room.

1. Face the students at the front of room
2. Count up to 3 rows
3. Start from the middle section’s left side
4. Count to the right 1 chair
5. Reach under chair and find it.

Another example is the square root of some number,

Fact: (declarative Knowledge), square root of a number x is y such that y\*y = x

**Imperative knowledge**

This is a very old recipe to find the square root of the a number by guessing the number nearest to the root. This is a very old algorithm its attributed to Alexandra of heron dating back to the second century BC. But here is the recipe to finding the square root.

1. Start with a guess ,G
2. If G\*G is close enough to x, stop and say G is the answer
3. Otherwise make a new guess by averaging G and x/G and a better averaging technique is (G+x/G)/2
4. Using the new guess, repeat process until close enough

|  |  |  |  |
| --- | --- | --- | --- |
| G | G\*G | x/G | (G+x/G)/2 |
| 3 | 9 | 5.333333 | 4.166666667 |
| 4.166667 | 17.36111 | 3.84 | 4.003333333 |
| 4.003333 | 16.02668 | 3.996669 | 4.000001388 |

A close up of text on a white background

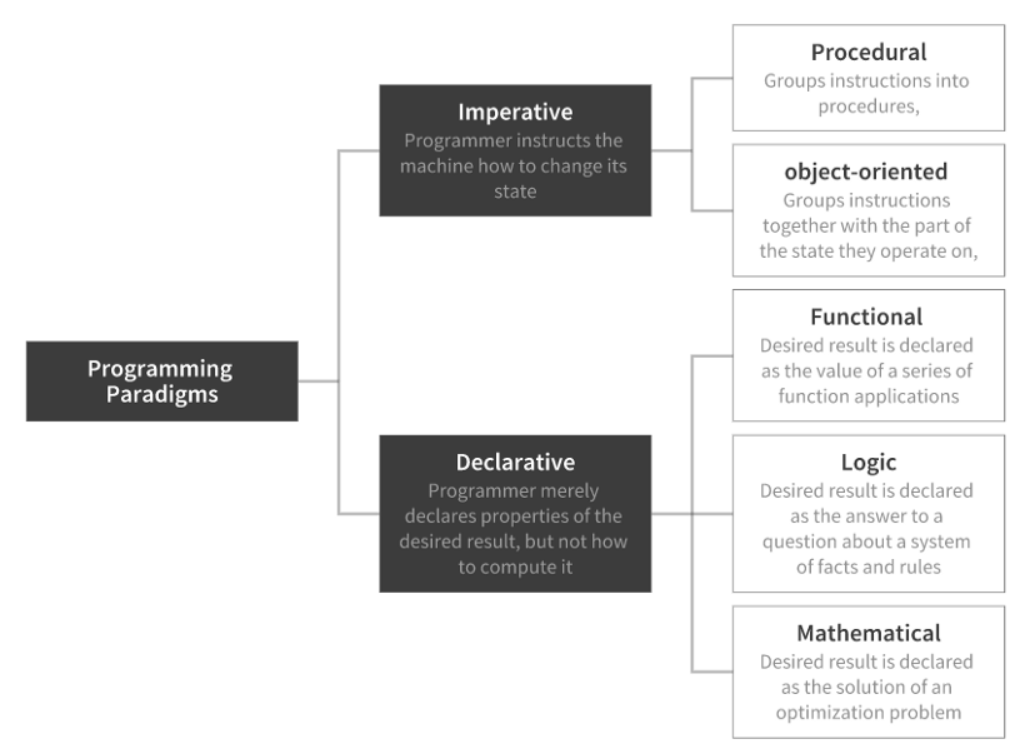
Description automatically generated

What’s the recipe:

1. ***Sequence of simple steps***
2. ***Flow of control process that specifies when each step is executed.***
3. ***A means of determining when to stop.***

Steps 1+2+3 = algorithm

These two kind of knowledge are important but only the second one (imperative knowledge) is valuable to us because that’s what going to help us get the computer to do something for us.



**Session 3**

We described a recipe, it’s a sequence of mechanical steps its going to capture the sequence of things I want to do in order to compute something, now let bring this recipe inside the computer. So the next question becomes how to capture the recipe inside a mechanical process. Let take the square root example I like to compute square roots for me historically there where 2 choices here

1. Fixed program: this would be the computer which would be designed to specifically to calculate the square root of a number. Ex: hand held calculators
2. Stored program: you can load into it that sequence of instruction that receipt and then inside that computer their going to set of parts that are going to do those instructions when I ask them to. In particular there is going to be a program inside the computer typically interpreter which once you started up is going to walk through each of those sequence of instructions in turn doing the computation I want.

**Session 4**

Now we got the idea of creating a recipe as a generic idea now lets see how we can create a recipe. Now we want to go from a description of a process to a specific set of statements that we can store in the machine so that interpreter that evaluator can then run those operations to use the primitives inside the machine to do the work for us. Below are the factor that will help us in creating recipes.

* A programming language provides a set of primitive operations
* Expressions are complex but legal combinations of primitive in a programming language
* Expressions and computations have values and meanings in a programming language.

Every programming languages have set of primitives a means of combinations a way of putting those primitives together to create new expressions and **a means of abstractions, way of taking some complex expression and treating it as it’s a primitive**

Lets take the analogy with the natural language such as English. In English we have constructs which are nothing but words lots of them some are more common than others. In a programming language we also have primitive constructs these are the atoms on which we’re going to build things in a programming language those are typically numbers strings or just sequence of characters and simple operations the thing that are provided to us by the manufacturer addition, subtraction comparison given those primitives we will put them together.

***Syntax:***

* English : “cat dog boy” -> not syntactically valid

“cat hugs boy” -> syntactically valid

* Programming language: “Hi”5 -> not syntactically correct

3.2\*5 -> syntactically valid

***Static semantics*** is which syntactically valid strings have meaning

* English: “I are hungry” ->syntactically valid, but static semantic error
* Programming Language: 3.2\*5 -> syntactically valid

3+”hi”->static semantic error

Once we have both syntactically correct and static semantically correct expressions we want them to know whats the meaning associated with them and we are going to start building this up.

Semantics is the meaning associated with a syntactically correct string of symbols with no static semantic errors

English : can have many meaning for a single statement:

“flying planes can be dangerous”

“this reading lamp hasn’t uttered a word since I bought it?”

But programming languages have only one meaning , and it may also be possible that is what the programmer may not have been intended to get the output.

Syntactic errors:

* Common and easily caught by most of the IDE’s.

Static semantic errors:

* Some language check for these before running program.
* Can cause unpredictable behavior.

Non semantic errors but different meaning than what programmer intended:

* Program crashes, stops running
* Program runs forever
* Program gives an answer but different than expected.

We are going to see patterns of computation we are going to build up computational modes of thought, different styles of solving problem because those styles are going to be common and can be easily reused when we see a new problem that fits into same category.

**Session 5**

Python Programs

A program is a sequence of definition and commands

* Definitions (functions) as we will see bit later on are ways of either assigning names to values or more importantly creating procedures that we are going to treat as it they are primitives those are being evaluated.
* Commands are simpler expressions that we can execute directly within python and we will do it in something called a shell(idle)

What are the fundamental primitives that represent data those we call objects and programs manipulate the objects in order to get out parts of those objects or to do something with those objects. Every object has a type is associated with it and tells us what kind of thing it is. Types tell programs whether they can act on it or not. If a program is expecting a number and we give it a string the program will throw an error, so the type of the object is really valuable.

Object can be scalars or non scalars (vectors). Scalars which says they cant be subdivided. Non scalars which are things that have internal structure into which we can pull out parts

Lets start with the simplest version of the scalar objects.

**Scalar objects:**

Int – represent integers, ex.5

Float – represent real numbers, ex. 3.27

Bool – represent Boolean values True and False

NoneType – special and has one value, None

Can use type() to see the type of an object

In[1]: type(5)

Out[1]:int

In[2]:type(3.0)

Out[2]: float

Some time we can change types specifically with numbers this is called Casting and we can cast different forms and we can see the different examples if we want to take an int such as 5 and turn it into a float we can do so by giving it the special command float the thing contained within parentheses and it converts it. The other direction also works we want to convert the float 3.5 to integer we will get 3, here we might have some choices but python simply gives the whole number part of it doesn’t round to the nearest integer it will simply truncate it.

Now we got the type lets put them in the expression 2+3 =5, that has the value associated with it we take 2 ,3 and apply the arithmetic operation of addition to them and when we evaluate it, it simply returns out 5. If we use the print statement to print out the result of addition 2+3 it will return the result. It is very important to note that the print is a function and it will return the operation. If no parameter is passed in the print function then no values is returned, print returns that none type.

In[1]: 2+3

Out[1]:5

Lets discuss on the different operators available in the python

|  |  |  |
| --- | --- | --- |
| operator | operation | Rresults |
| i+j | Sum | if both are ints, results is int if either or both are float , result is float |
| i-J | difference/subtraction |
| i\*j | product/Multiplication |
| i/j | division | result is float |
| i//j | int division | result is int , quotient without reminder |
| j%j | reminder | when i is divided by j |
| i\*\*j | power | i to the power of j |

There is a precedence to these operation and those are shown in the below.

Simple operations

Parentheses used to tell python to do these operations first

* + - 3\*5+1 evaluates to 16
    - 3\*(5+1) evaluates to 18

Operator precedence with out parentheses

* + - \*\*
    - \*
    - /
    - + and –
    - Executed left to right order, as appear in expression

BEDMAS – Brackets, Exponent, Division, Multiplication, Addition and Subtraction

**Session 6**

Abstractions : ways of giving name to things. So we can refer to the value just by its name reuse it. In python we do an assignment with the ‘Equals to’ operator as shown in the below example. An assignment that says I bind to our associate with the name pi the value 3.14159 once we have that we can use pi wherever we would like. We can also assign the approximation of the pi to a variable.

Ex: Pi = 3.14159, pi\_approx=22/7

Once we have that we can give more complicated expressions. One is we can reuse the name and not redo the computation.

Its also easier to understand. Its lets me think about what its telling me. And what this represents. Finally its going to be easier to change the code later if I have just got name and I want to change the binding of the name and reuse it.

So here is a simple example, we can bind the pi to a value I can give a radius say 2.2 and then I can compute the area. Pi\*(radius)2

That will give a simple little computation those expression also tell me something about what we are trying to capture and so those name turns out to be valuable. One of the things to keep in mind is that this is simply following of steps that we talked about. Its not like doing math in programming you do not solve for x.

One more concept we need to express here is

Radius = radius+1 is same as, radius +=1, it we increment the variable by the number.

Change the binding

* We can re-bind variable name using new assignment statements
* Previous values may still stored in memory but lost the handle for it.
* Value for area does not change until you tell the computer to do the calculation again.

another way of putting it is lets consider the this cloud as its memory,

3.14

2.2

3.2

115.1976

when we do the binding for ‘pi’ it associates the name pi with a value

pi pi

of in memory when we radius we get the same

thing and we compute for area it looks up radius

Radiuss\

squares it multiplies it by pi and then associates the name area with

with that value in memory and now if we change radius

it loses the first association (2.2) and creates a new association with

Area

value (3.2), but the area hasn’t changed. If I wanted to

to change the area too then I will have to recompute the formula to find the area of the circle.

**Session 7**

Let s add one more piece to above properties in which is the ability to make decision based on tests and for that we have to compare things.

Now lets see what are the comparison operators for integers and float, I and j are any variable names

|  |
| --- |
| i>j |
| i>=j |
| i<j |
| i<=j |
| i==j -> equality test, true if I equal j |
| i!=j ->Inequality test, true if I not equal to j |

Logical operators on bools, and b are any variable names

|  |
| --- |
| Not a -> True if a is False  False if a is True |
| A and b -> Ture if both are True |
| A or b -> True if either or both are Ture |

Branching Programs

The simplest branching statement is a conditional

* A test (expression that evaluates to True or False)
* A block of code to execute if the test is True
* An optional block of code to execute if the test is false.

This is nothing but the if else statement.

In python it is not necessary to have a false block aka else , as else is a optional part in the python. With this in mind we can start building a little more interesting program.

X = int(input(‘Enter an integer: ’)) #input is the keyword which allows me to send value into program In string

If x%2 == 0:

Print(‘’)

Print(‘Even’)

Else:

Print(‘’)

Print(‘odd’)

Print(‘done with conditional’)

How did python know those are the blocks because of the is indentation right after the colon the fact that those lines of code are in set are important because it tells us that’s an entire block of code and when branching is done here we are going to pick back up the rest of the code.

Observations of the above example.

* The expression x %2 ==0 evaluate to true when the remainder of x divided by 2 is 0
* Note that == is used for compression, since = is reserved for assignment
* The indentation is important – each indented set of expression denotes a block of instructions
  + - For example, if the last stamen were indented, it would be executed as part of the else block of code
* Note how this indentation provides a visual structure that reflects the semantics structure of the program

What will be the case when we have multiple condition or condition within condition, then we use elif and nested condition respectively. If x%2==0:

If x%3==0:

Print('Divisible by 2 and 3’)

Else:

Print(‘Divisible by 2 and not by 3’)

Elif x%3==0: # short hand for else if

Print(‘Divisible by 3 and not by 2’)

Control flow - Branching

|  |  |
| --- | --- |
| If <condition>:  <expression>  <expression>  …… | If <condition>:  <expression>  <expression>  ……  Elif<condition>:  <expression>  <expression>  …….  Else:  <expression>  <expression>  ……. |
| If <condition>:  <expression>  <expression>  ……  Else:  <expression>  <expression>  ……. |

* <condition> has a value True or False
* Evaluate expression in that block if <condition> is True
* In Python indention matters to identify the code.

**Session 8**

Strings : A string is a sequence of characters special characters, spaces, letters, digits etc, all these sequence of characters can be within double quotation or single quotation marks

Hi = ‘Hello world’

Greetings = “Hello”

Concatenation of strings

Name= “Eric”

Greet = hi + Name

Greeting = hi + ” ” + Name

Greet = hi + Name # notice concatenation just as with numbers could have multiple things in rows so we can concatenate multiple things together as we did here what we say when we talk about addition this way we say we have overloaded it. We have addition apply to not only to numbers but also to strings.

The type of an argument and the type of an object would tell an operation some important information, here it does that because its telling addition if the two things are strings then concatenate them its also telling addition if we give you two numbers just add them together using straight forward arithmetic so we can use addition in multiplication places to let the type of the object define what’s going to happen

Different operation on strings

* ‘ab’+’cd’ -> concatenation
* 3 \* ’abc’ -> successive concatenation / echo
* Len(‘eric’) -> the length
* ‘eric’[1] ->indexing #Begins with index 0 attempting to index beyond length -1 is an error
* ‘eric’[1:3]-> slicing
  + Extracts sequence starting at first index and ending before second index
  + If no value before : , starts from 0
  + If no value after: ends at length
  + If just : make a copy of entire sequence

Strings have things inside of them they are the first version of non scalar version object in that we can get out pieces with slicing and indexes.

**Session 9**

We need to be able to both read to **input** and print stuff out or return values (**output**).

So lets start with **output.** With the output I will want to be able to print something to the monitor as we are doing computation not just at the end of a computation. To do so here we have a keyword called as print.

X =1

Print(x)

Print(type(x))

X\_str= str(x)

In>> print(“my fav num is “,x,”.”,”x =”,x)

my fav num is 1 . x = 1

In the above statement we are trying perform a combination of sequence of expression with the int data type with the string. Now if we print it we get pretty funky. It says “my fav num is 1 . x \* 1” but then there is a space before the period and then it says x =1, print statement here is literally printing each element followed by the space. And particular x has a value that’s going to show up separately from the period

In>> Print(“my fav num is ”+x\_str+”. ”+”x = ”+x\_str)

my fav num is 1. x = 1

On the other hand if we give x as a string is simply going to put the two pieces together and so we can do a second kind of print which is we can say print("my fav num is "+x\_str+". "+"x = "+x\_str) here we are adding or concatenate the expression and we get the above result. As we can see the print statement here is not inserting a space at every expression its adding the expression as it is rather it.

So what is the conclusion of the above experiment is: in the case of print with multiple arguments it will print each one of them spaced apart. In the second case we are concatenating the entire expression into one long string and then printing it out and we can control the spacing around it. ‘,’ i.e comma will identify each expression separately and add spacing in between the expression but ‘+’ i.e addition or concatenation will convert all the expression into one long string and the print out the sentence as it is with out inserting any spacing in between.

Now lets look at **input**, the input keyword takes a single argument which is a string and its behavior is its going to print out whatever is inside of the those quotes, its going to wait for us to type something in when we hit enter its going to evaluate whatever we typed in and return that as the result of the input statement. Typically I want to bind that to a variable name like text and if

>>Text = input(“Type Something ”)

Type Something: Booo

>>Text

‘Booo’

>> print(5\*Text)

Booo Booo Booo Booo Booo

Input simply expects everything to be a string so we could. If we type any value as a input it will automatically read it in as the string. But that said if we want to get the number in then we will have to cast it to integer.

Num = input(“Enter a number”)

Print (5\*num)

**Session 10**

IDE’s comes with

* Text Editor: use to enter, edit and save your programs
* Shell- place in which to interact with and run your programs, standard methods to evaluate your program
* Integrated debugger (we will use it later)

**IDE (Integrated Development Environment)**

* Ananconda
* Jupiter
* Pycharm
* Intellij - Separate go through.

**Session 11**

Lets continue on the branching programs (conditional statements if and else) to next level which is nothing but iteration

Using control in loops.

* simple branching programs just make choices but path through code is still linear.
* Sometimes want to reuse parts of the code indeterminate number of times.

The second statement cannot be achieved with simple branching program we have to design the solution for the problem with the loop so that the program can iterate based on the number of time the control code is successful or the code exhaust the number of iteration.

To achieve this exhaust number of iteration we have **while loop**. Now we can control the flow with while loop

While <condition>:

<expression>

<expression>

* <condition> evaluates to a Boolean
* If <condition> is True, do all the steps inside the while code block
* Check <condition> again
* Repeat until <condition> is False

Below is the program to get out of the woods.

|  |
| --- |
| You are in the Lost forest.  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Go left or right? |

N = input(“you are in the Lost Forest. Go left or Right?”)

While N ==”right”:

N = input (“you are in the Lost Forest. Go left or Right?”)

Print(“you got out of the Lost Forest.”)

We can also achieve the same with for loop.

For n in range (5):

Print(n)

For loop has a particular syntax it has a variable ‘n’, it has a keyword ‘in’ and limit which is ‘range’, range is something that returns basically a sequence of integers starting at 0 and ending before 5 (0,1,2,3,4).

Lets recapture the general format of for loop

for <variable> in range (<some\_num>):

<expression>

<expression>

…

* Each time through the loop, <variable> takes a value
* First time, <variable> starts at the smallest value
* Next time, <variable> gets the range value +1
* Etc

We can give different set of range in for loop,

* Default values are start =0 and step =1 and is optional
* Loop until value is stop -1

Mysum = 0

For i in range (7,10): # loop starts @ 7 and ends @9 , in this case the range will give up to but not including 10

mysum +=i

Print(mysum)

# loop starts @ 5 and ends @10 , in this case the range will give up to but not including 11 and step = 2 so every time the loop runs it will jump 2 steps, first value being 5, 2nd turn value being 7, 3rd turn value is 9 and so on

>>> for i in range (5,11,2):

print(i)

5

7

9

Break statement

* Immediately exits whatever loop is in
* Skips remaining expressions in code block
* Exits only innermost loop

While <condition\_1>:

While <condition\_2>:

<expression\_a>

Break

<expression\_b>

<expression\_c>

If we are running through some code and execute or see rather that special keyword break it stops the execution of that loop at that point or that code block at that point and break out of it. Similarly it happens in for loop

mysum =0

for i in range (5,11,2):

mysum += i

if mysum ==5

break

print(mysum)

|  |  |
| --- | --- |
| For Loop | While Loop |
| * Know number of iterations * Can end early via break * User a counter * Can rewrite a for loop using a while loop | * Unbounded number of iteration * Can end early via break statement * Can use a counter but must initialize before loop and increment it inside the loop * May not be able to rewrite a while loop using a for loop |

**Session 12**

The notion of the for loop or a while loop extends the ability to write programs considerably for us that generalization of concept of iteration goes much beyond what we could do with simple branching programs.

With iteration we can do more, we can use the same code multiple times as with a branch we start with the test if that test is true then we are going to execute the loop body this is a repetitive action in the loop until the condition remain to be false.

An example to make it clear.

X =3

Ans = 0

iterLeft = x

while (iterLeft !=0):

ans = ans+x

iterLeft = iterLeft-1

print(str(x)+’\*’+str(x)+’=’+str(ans))

output: 3 \* 3 = 9

run it in IDE

**Session 13** (Very important – Iterative Algorithms)

**Guess and check algorithms**

the notion of iterative algorithm is really valuable it broadens dramatically the class of things that we can compute again if it just have branching things we can go through the code only once. But now with help of loops we can do much more iteration with the reusable branching code.

One of the things come out naturally once we have this idea of repeating things is set of methods that are sometimes called guess and check methods. They are very useful even though they are not most efficient technique. We want to extend that notion of iterative algorithms to capture examples of guess and check methods.

Lets take the example of square root mentioned earlier: square root of X was that Y such that Y squared was equal to X. we looked at an algorithm to actually computer it but now we can go back to that idea because if we had a good way to generate guesses we could use that declarative definition to check that’s the basis of guess and check. We are going to have to come up with a way of systematically making guesses for the possible answer check them and keep doing that in an iterative algorithm until we find a good solution lets see how we can do that.

Lets extend the example to finding the cube root of integer:

* One way to use this idea of generating guesses in order to find a cube root of x is to first try 0\*\*3, then 1\*\*3 and then 2\*\*3 and so on.
* Can stop when reach k such that k\*\*3>X.
* Only a finite number of cases to try.

|  |
| --- |
| x=int(input(**"Enter an positive integer"**)) ans = 0 while ans\*\*3<x:  ans = ans+1 *#this line generates guesses.*  if x-a\*\*3 <=0:  break if ans\*\*3 !=x: *#this is a declarative knowledge of testing cube root* print(str(x)+**' is not a perfect cube'**) else:  print(**'Cube root of '**+str(x)+**' is '**+ str(ans)) |

This above approach will be applicable to positive integers not for real number or negative numbers etc. let try to write the code for the above approach. Now let perform the modification for the possible negative scenarios.

|  |
| --- |
| x=int(input(**"Enter an positive integer"**)) ans = 0 while ans\*\*3<abs(x):  ans = ans+1 *#this line generates guesses.* if abs(x)-ans\*\*3<=0: *#this line is a decrementing function.*  break if ans\*\*3!=abs(x): *#this is a declarative knowledge of testing cube root* print(str(x)+**' is not a perfect cube'**) else:  if x<0:  ans = -ans #to write the ans in a positive or negative format  print(**'Cube root of '**+str(x)+**' is '**+ str(ans)) |

abs() is a built-in function to take the absolute value of X I am going to check the cube root against the absolute value of X then when we are done we can decide the negative or the positive version of the cube root function. As seen in the above code we can easily extend the code to build new versions of things to handle cases that we didn’t consider when writing it for the first time.

Loop Characteristics: now we know to do a loop that we need some characteristics

* we need a loop variable that’s initialized in outside loop.
* and the value changes with in the loop.
* test for termination after action are performed.

That notion we can also capture in a little bit more general form. Because it is useful to think about what’s happening inside the loop how do we know the loop actually going to stop and for that we often want to think about something we call a decrementing function, the idea is that we are going to map all of the program variables into an integer and when the loop is entered that value is going to be something that’s non- negative when we go through the loop is going to keep decreasing and when that value gets less than or equal to zero we have to terminate so this is the capturing of the loop variable it in terms of looking at what’s being done inside the loop.

In the above example we will use absolute value of X -ans\*\*3 as my decrementing function.

What could go wrong in the loops

* suppose we don’t initialize the variable?
  + Likely get a NameError: or worse use and expected value to initiate the computation
* Suppose we don’t change the variable inside the loop?
  + Will end up in an infinite loop, never reaching the terminating condition.

Guess and check

* You are able to guess a value for solution
* You are able to check if the solution is correct
* Keep guessing until find solution or guessed all values
* The process is exhaustive enumeration

These algorithms are called as exhaustive algorithms,

* You are going to exhaust all possible options to use
* This is going to take a while to run so you get tired waiting for it to finish but as computers get faster surprisingly many of these algorithms can actually work quite well

Let try this out with the for loop

|  |
| --- |
| cube = 8 for guess in range(cube):  if guess\*\*3 == cube:  print(**"cube root of "**,cube,**" is "**, guess) |

Extending the above code for other possible inputs and errors.

|  |
| --- |
| cube = 28 for guess in range(cube):  if guess\*\*3 == abs(cube):  break if guess\*\*3 !=abs(cube):  print(cube,**'is not a perfect cube'**) else:  if cube<0:  guess = -guess  print(**"cube root of "**,cube,**" is "**, guess) |

Exhaustive Enumeration

* Guess and check methods can work on problems with a finite number of possibilities.
* Exhaustive enumeration is a good way to generate guesses in an organized manner.

**Session 14**

Guess and check for square root

|  |
| --- |
| ans = 0 neg\_Flag = False x = int(input(**"Enter an integer: "**)) if x<0:  neg\_Flag = True while ans\*\*2<x:  ans = ans+1 if ans \*\* 2 ==x:  print(**"square root of "**, x,**"is"**, ans) else:  print(x,**"is not a perfect square"**)  if neg\_Flag:  print(**"just checking... did you mean"**,-x,**"?"**) |

Just a reminder her is something we have tried doing it earlier in the loops above is the example of some codes that’s going to compute a square root of something and it does it by running through a loop as we have done before it’s going to set up an initial value which it gets here by inputting something that enters an integer and then it’s going to run through a little while where it does guess and check.it keeps trying different versions of answer starting at 0 then 1 then 2going on until it sees whether that an answer squared is bigger than or equal to X once it gets at that point it checks to see if it actually the square is equal to X in which case it prints out that it found the square root and if not it tell us that it’s not a perfect square and just depending on whether it was a positive or negative thing it gives a little bit of additional information, this is the idea of guess and check.

**Reviewing Strings**

* Think of as a sequence of case sensitive character
* Can compare strings with ==,>,< etc
* Len() is a function used to retrieve the length of the string in the parentheses.
* Square brackets used to perfrom indexing into a string to get the value at a certtian index/position

S=”abc”

Indexing -> ‘a’ is 0, ‘b’ is 1 and ‘c’ is 2

Len(S) -> evaluates to 3

S[0] -> evaluates to a

S[1] -> evaluates to b

S[2] -> evaluates to c

S[3] -> trying to index out of bounds, error

* Can slice strings using [ start: stop: step]

S = “abcdefgh”

S[::-1] -> evaluates to ”hgfedcba”

S[3:6] -> evaluates to “def” emits the 6th or last index

S[-1] -> evaluates to “h”

* Strings are “immutable ” – cannot be modified

S = “hello”

S[0] = ‘y’ -> gives error

S = ‘y’ + S[1:len(S)] -> is allowed

S is a new object