**Program Flow**

**Our first turtle program**

|  |
| --- |
| import turtle #Allows us to use turtles  window = turtle.Screen() #Create a playground for turtles  amit = turtle.Turtle() #Create a turtle, assin to amit  amit.forward(50) #Tell amit to move forward by 50 units  amit.left(90) #Tell amit to turn by 90 degrees  amit.forward(30) #Complete the second side of a rectangle  window.mainloop() #Wait for user to close window |

The dot notation turtle.Turtle means “The Turtle type that is

defined within the turtle module”. (Remember that Python is case sensitive, so the module name, with a lowercase t, is different from the type Turtle.)

An object can have various methods — things it can do — and it can also have attributes — (sometimes called properties). For example, each turtle has a color attribute. The method invocation amit.color("red") will make amit red, and drawing will be red too. (Note the word color is spelled the American way!)

**Instances — a herd of turtles**

Just like we can have many different integers in a program, we can have many turtles. Each of them is called aninstance. Each instance has its own attributes and methods — so alex might draw with a thin black pen and be at some position, while tess might be going in her own direction with a fat pink pen.

**The for loop**

Python’s for loop is to be able to repeat some code, over and over again.

|  |
| --- |
| for friend in ['Amit', 'Mukul', 'Adi', 'Kshitij', 'Abhishek', 'Pari']:  invite = 'Hi ' + friend + '. Please come to my party!'  print(invite)  #More code can follow here... |

The variable friend in the for statement at line 1 is called the loop variable. We could have chosen any other variable name instead, such as broccoli: the computer doesn’t care.

Lines 2 and 3 are the loop body. The loop body is always indented. The indentation determines exactly what statements are “in the body of the loop”.

On each iteration or pass of the loop, first a check is done to see if there are still more items to be processed. If there are none left (this is called the terminating condition of the loop), the loop has finished. Program execution continues at the next statement after the loop body, (e.g. in this case the next statement below the comment in line 4).

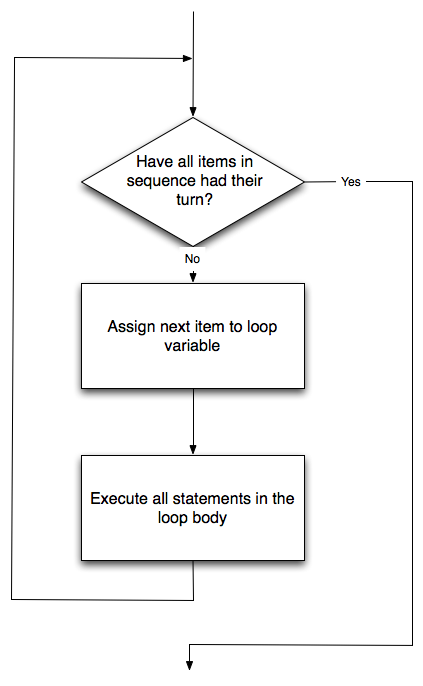
If there are items still to be processed, the loop variable is updated to refer to the next item in the list. This means, in this case, that the loop body is executed here 7 times, and each time friend will refer to a different friend.

At the end of each execution of the body of the loop, Python returns to the for statement, to see if there are more items to be handled, and to assign the next one to friend.

**Flow of Execution of the for loop**

As a program executes, the interpreter always keeps track of which statement is about to be executed. We call this the **control flow**, of the **flow of execution** of the program.

Control flow is often easy to visualize and understand if we draw a flowchart. This shows the exact steps and logic of how the for statement executes.



**The loop simplifies our turtle program**

|  |
| --- |
| for i in range(4):  #Excutes the body with i = 0, then 1, then 2, then 3  for x in range(10):  #Sets x to each of ... [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] |

range can deliver a sequence of values to the loop variable in the for loop. They start at 0, and in these cases do not include the 4 or the 10.

**A few more turtle methods and tricks**

Turtle methods can use negative angles or distances. So tess.forward(-100) will move tess backwards, and tess.left(-30) turns her to the right. Additionally, because there are 360 degrees in a circle, turning 30 to the left will get tess facing in the same direction as turning 330 to the right! (The on-screen animation will differ, though — you will be able to tell if tess is turning clockwise or counter-clockwise!)

**Conditionals**

**Boolean values and expressions**

A Boolean value is either true or false. It is named after the British mathematician, George Boole, who first formulated Boolean algebra — some rules for reasoning about and combining these values. This is the basis of all modern computer logic.

In Python, the two Boolean values are True and False (the capitalization must be exactly as shown), and the Python type is **bool**.

|  |
| --- |
| >>> type(True)  <class 'bool'>  >>> type(true)  Traceback (most recent call last):  File "<pyshell#1>", line 1, in <module>  type(true)  NameError: name 'true' is not defined |

A Boolean expression is an expression that evaluates to produce a result which is a Boolean value. For example, the operator == tests if two values are equal. It produces (or yields) a Boolean value:

The == operator is one of six common comparison operators which all produce a bool result; here are all six:

|  |
| --- |
| X == y #Produce True if … x is equal to y  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 or equal to y  x <= y #…x is less than or equal to y |

A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a comparison operator. Also, there is no such thing as =< or =>.

**Logical operators**

There are three logical operators, and, or, and not, that allow us to build more complex Boolean expressions from simpler Boolean expressions. The semantics (meaning) of these operators is similar to their meaning in English. For example, x > 0 and x < 10 produces True only if x is greater than 0 and at the same time, x is less than 10.

n % 2 == 0 or n % 3 == 0 is True if either of the conditions is True, that is, if the number n is divisible by 2 or it is divisible by 3. (What do you think happens if n is divisible by both 2 and by 3 at the same time? Will the expression yield True or False? Try it in your Python interpreter.)

Finally, the not operator negates a Boolean value, so not (x > y) is True if (x > y) is False, that is, if x is less than or equal to y. In other words: not True is False, and not False is True.

The expression on the left of the or operator is evaluated first: if the result is True, Python does not (and need not) evaluate the expression on the right — this is called short-circuit evaluation. Similarly, for the and operator, if the expression on the left yields False, Python does not evaluate the expression on the right.

**Truth Tables**

A truth table is a small table that allows us to list all the possible inputs, and to give the results for the logical operators.

Because the and and or operators each have two operands, there are only four rows in a truth table that describes the semantics of **and**.

|  |  |  |
| --- | --- | --- |
| a | b | a and b |
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

here is the truth table describing **or**:

|  |  |  |
| --- | --- | --- |
| a | b | a and b |
| F | F | F |
| F | T | T |
| T | F | T |
| T | T | T |

The third logical operator, not, only takes a single operand, so its truth table only has two rows:

|  |  |
| --- | --- |
| a | not a |
| T | F |
| F | T |

**Simplifying Boolean Expressions**

A set of rules for simplifying and rearranging expressions is called an algebra. the Boolean algebra — which provides rules for working with Boolean values.

First, the and operator:

|  |
| --- |
| x and False == False  False and x == False  y and x == x and y  x and True == x  True and x == x  x and x == x |

some corresponding rules for the or operator:

|  |
| --- |
| x or False == x  False or x == x  y or x == x or y  x or True == True  True or x == True  x or x == x |

Two not operators cancel each other:

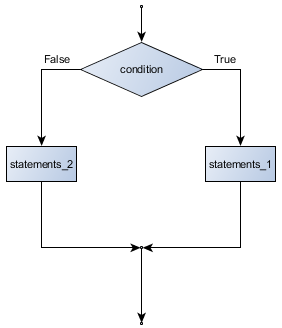
|  |
| --- |
| not (not x) == x |

**Conditional execution**

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the if statement:

The syntax for an if statement looks like this:

|  |
| --- |
| If <BOOLEAN EXPRESSION>:  <STATEMENT\_1> #Executed if condition evaluate to True  else:  <STATEMENT\_2> #Executed if condition evaluate to False |

****

The Boolean expression after the if statement is called the condition. If it is true, then all the indented statements get executed. If not, then all the statements indented under the else clause get executed.

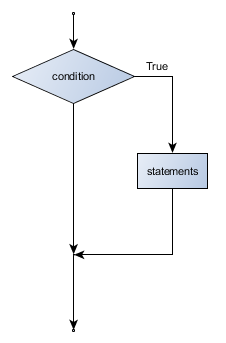
The header line begins with the keyword if followed by a Boolean expression and ends with a colon (:). The indented statements that follow are called a block. The first unindented statement marks the end of the block.

Each of the statements inside the first block of statements are executed in order if the Boolean expression evaluates to True. The entire first block of statements is skipped if the Boolean expression evaluates to False, and instead all the statements indented under the else clause are executed.

There is no limit on the number of statements that can appear under the two clauses of an if statement, but there has to be at least one statement in each block.

|  |
| --- |
| if True: #This is always True,  pass #So this is always executed, but is does nothing  else:  pass #And this is never executed |

**Omitting the else clause**

****

Another form of the if statement is one in which the else clause is omitted entirely. In this case, when the condition evaluates to True, the statements are executed, otherwise the flow of execution continues to the statement after the if.

|  |
| --- |
| import math  x = int(input("Enter the number: "))  if x < 0:  print("The negative number ", x, " x not valid here.")  x = 42  print("I've decided to use the number 42 instead.")  print("The square root of ", x, "is", math.sqrt(x)) |

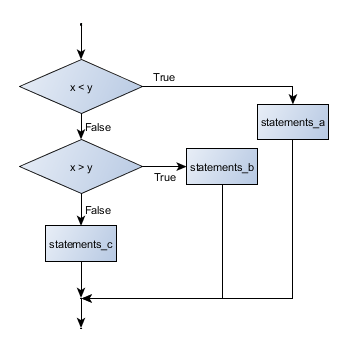
we have an import math statement, usually placed near the top of our script.

Notice too that else is not a statement. The if statement has two clauses, one of which is the (optional) else clause.

**Chained conditionals**

Sometimes there are more than two possibilities and we need more than two branches. One way to express a compu- tation like that is a chained conditional:

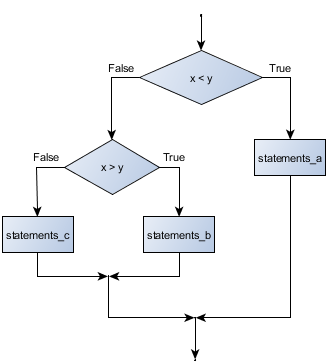
|  |
| --- |
| if x < y:  <STATEMENTS\_A>  elif x > y:  <STATEMENTS\_B>  else: # x == y  <STATEMENTS\_C> |



elif is an abbreviation of else if. Again, exactly one branch will be executed. There is no limit of the number of elif statements but only a single (and optional) final else statement is allowed and it must be the last branch in the statement:

**Nested conditionals**

One conditional can also be nested within another.



|  |
| --- |
| if x < y:  <STATEMENTS\_A>  else:  if x > y:  <STATEMENTS\_B>  else:  <STATEMENTS\_C> |

**Logical opposites**

Each of the six relational operators has a logical opposite:

Notice that the opposite of >= is <.

|  |  |
| --- | --- |
| operator | logical opposite |
| == | != |
| != | == |
| < | >= |
| <= | > |
| > | <= |
| >= | < |

Understanding these logical opposites allows us to sometimes get rid of not operators. not operators are often quite difficult to read in computer code, and our intentions will usually be clearer if we can eliminate them.

Two powerful simplification laws (called de Morgan’s laws) that are often helpful when dealing with complicated Boolean expressions are:

|  |
| --- |
| (not (x and y)) == ((not x) or (not y))  (not (x or y)) == ((not x) and (not y)) |

**Iteration**

Repeated execution of a set of statements is called iteration.

**Assignment**

As we have mentioned previously, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

Note too that an equality test is symmetric, but assignment is not. For example, if a == 7 then 7 == a. But in Python, the statement

a = 7 is legal and 7 = a is not.

**Updating variables**

When an assignment statement is executed, the right-hand side expression (i.e. the expression that comes after the assignment token) is evaluated first. This produces a value. Then the assignment is made, so that the variable on the left-hand side now refers to the new value.

|  |
| --- |
| airtime\_remaining = 15  print(airtime\_remaining)  airtime\_remaining = 7  print(airtime\_remaining) |
| 15  7 |

Note too that an equality test is symmetric, but assignment is not. For example, if a == 7 then 7 == a. But in Python, the statement a = 7 is legal and 7 = a is not.

It is especially important to distinguish between an assignment statement and a Boolean expression that tests for equality. Because Python uses the equal token (=) for assignment, it is tempting to interpret a statement like a = b as a Boolean test. Unlike mathematics, it is not! Remember that the Python token for the equality operator is ==.

**Updating variables**

When an assignment statement is executed, the right-hand side expression (i.e. the expression that comes after the assignment token) is evaluated first. This produces a value. Then the assignment is made, so that the variable on the left-hand side now refers to the new value.

If you try to get the value of a variable that has never been assigned to, you’ll get an error:

|  |
| --- |
| >>> w = x + 1  Traceback (most recent call last):  File "<interactive input>", line 1, in  NameError: name 'x' is not defined |

Before you can update a variable, you have to initialize it to some starting value, usually with a simple assignment:

|  |
| --- |
| runs\_scored = 0  ...  runs\_scored = runs\_scored + 1 |

updating a variable by adding 1 to it — is very common. It is called an increment of the variable; subtracting 1 is called a decrement. Sometimes programmers also talk about bumping a variable, which means the same as incrementing it by 1. This is commonly done with the += operator.

|  |
| --- |
| runs\_scored = 0  ...  runs\_scored += 1 |

**The for loop revisited**

Recall that the for loop processes each item in a list. Each item in turn is (re-)assigned to the loop variable, and the body of the loop is executed.

|  |
| --- |
| for friend in ["Joe", "Zoe", "Zuki", "Thandi", "Paris"]:  invite = "Hi " + friend + ". Please come to my party!"  print(invite) |

Running through all the items in a list is called traversing the list, or traversal.

**The while statement**

|  |
| --- |
| while <CONDITION>:  <STATEMENT> |

while <CONDITION> is True, <STATEMENT> is executed.

the while statement as if it were English. It means, while i is less than or equal to n, continue

executing the body of the loop. Within the body, each time, increment i. When i passes n, return your accumulated

sum.

Notice that if the loop condition is False the first time we get loop, the statements in the body of the loop are never

executed.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false

and the loop terminates. Otherwise the loop will repeat forever, which is called an infinite loop.