**Unit 2**

**Conditional Executions and Functions**

**2.1 Comparison Operators**

These operators will allow us to compare variables and output a Boolean value (True or False).

If you have any sort of background in Math, these operators should be very straight forward.

First we'll present a table of the comparison operators and then work through some examples:

## Table of Comparison Operators

In the table below, a=3 and b=4.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | If the values of two operands are equal, then the condition becomes true. | (a == b) is not true. |
| != | If values of two operands are not equal, then condition becomes true. | (a != b) is true |
| > | If the value of left operand is greater than the value of right operand, then condition becomes true. | (a > b) is not true. |
| < | If the value of left operand is less than the value of right operand, then condition becomes true. | (a < b) is true. |
| >= | If the value of left operand is greater than or equal to the value of right operand, then condition becomes true. | (a >= b) is not true. |
| <= | If the value of left operand is less than or equal to the value of right operand, then condition becomes true. | (a <= b) is true. |

Examples of each of these.

#### Equal

1. 2 == 2

Output= True

1. 1 == 0

Output= False

**Note** that == is a comparison operator, while = is an assignment operator.

#### Not Equal

1. 2 != 1

Output= True

1. 2 != 2

Output: False

#### Greater Than

1. 2 > 1

Output= True

1. 2 > 4

Output= False

#### Less Than

1. 2 < 4

Output= True

1. 2 < 1

Output= False

#### Greater Than or Equal to

1. 2 >= 2

Output= True

1. 2 >= 1

Output= False

#### Less than or Equal to

#### 2 <= 2

Output= True

1. 2 <= 4

Output=False

# 2.2 Chained Comparison Operators

An interesting feature of Python is the ability to chain multiple comparisons to perform a more complex test. You can use these chained comparisons as shorthand for larger Boolean Expressions.

In this lecture we will learn how to chain comparison operators and we will also introduce two other important statements in Python: **and** and **or**.

Let's look at a few examples of using chains:

# 1 < 2 < 3

# The above statement checks if 1 was less than 2 **and** if 2 was less than 3.

Output= True

1. **1<2 and 2<3**

The above checks if 3 is larger than both of the other numbers

Output= True

1. **1<3 and 3>2**

Output= True

It's important to note that Python is checking both instances of the comparisons.

1. **1==2 or 2<3**

Output= True.

Note how it was true; this is because with the **or** operator, we only need one or the other to be true.

1. **1==1 or 100==1**

Output= True

# 2.3 Errors and Exception Handling

# 

Note how we get a SyntaxError, with the further description that it was an EOL (End of Line Error) while scanning the string literal. This is specific enough for us to see that we forgot a single quote at the end of the line. Understanding these various error types will help you debug your code much faster.

This type of error and description is known as an Exception. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal.

## try and except

The basic terminology and syntax used to handle errors in Python are the try and except statements. The code which can cause an exception to occur is put in the try block and the handling of the exception is then implemented in the except block of code. The syntax follows:

try:

You do your operations here...

...

except ExceptionI:

If there is ExceptionI, then execute this block.

except ExceptionII:

If there is ExceptionII, then execute this block.

...

else:

If there is no exception then execute this block.

In Python, exceptions can be handled using a try statement.

The critical operation which can raise an exception is placed inside the try clause. The code that handles the exceptions is written in the except clause.

We can thus choose what operations to perform once we have caught the exception. Here is a simple example.

import sys

randomList = ['a', 0, 2]

for entry in randomList:

try:

print("The entry is", entry)

r = 1/int(entry)

break

except:

print("Oops!", sys.exc\_info()[0], "occurred.")

print("Next entry.")

print()

print("The reciprocal of", entry, "is", r)

**Output**

The entry is a

Oops! <class 'ValueError'> occurred.

Next entry.

The entry is 0

Oops! <class 'ZeroDivisionError'> occured.

Next entry.

The entry is 2

The reciprocal of 2 is 0.5

## finally

The finally: block of code will always be run regardless if there was an exception in the try code block. The syntax is:

try:

Code block here

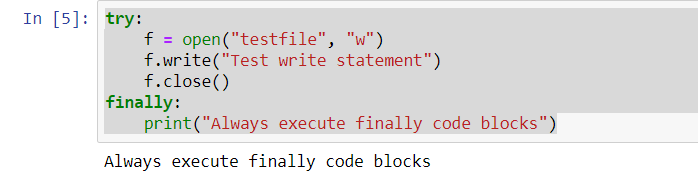
...

Due to any exception, this code may be skipped!

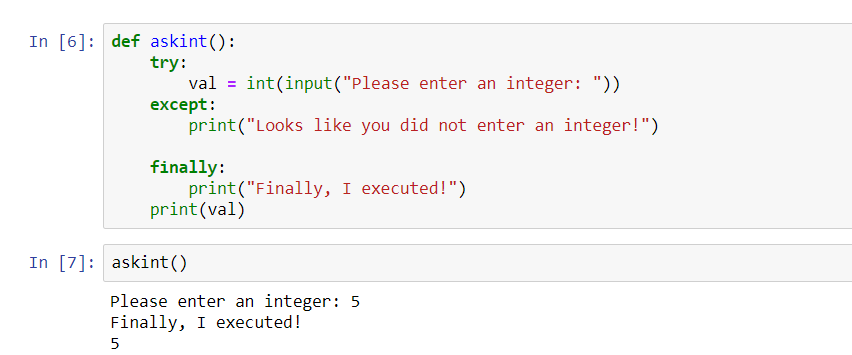
finally:

This code block would always be executed.

Example 1:



Example 2:



# 2.4 Introduction to Python Statements

In this lecture we will be doing a quick overview of Python Statements. This lecture will emphasize differences between Python and other languages such as C++.

There are two reasons we take this approach for learning the context of Python Statements:

1.) If you are coming from a different language this will rapidly accelerate your understanding of Python.

2.) Learning about statements will allow you to be able to read other languages more easily in the future.

## Python vs Other Languages

Let's create a simple statement that says: "If a is greater than b, assign 2 to a and 4 to b"

Take a look at these two if statements (we will learn about building out if statements soon).

**Version 1 (Other Languages)**

if (a>b){

a = 2;

b = 4;

}

**Version 2 (Python)**

if a>b:

a = 2

b = 4

You'll notice that Python is less cluttered and much more readable than the first version. How does Python manage this?

Let's walk through the main differences:

Python gets rid of () and {} by incorporating two main factors: a colon and whitespace. The statement is ended with a colon, and whitespace is used (indentation) to describe what takes place in case of the statement.

Another major difference is the lack of semicolons in Python. Semicolons are used to denote statement endings in many other languages, but in Python, the end of a line is the same as the end of a statement.

Lastly, to end this brief overview of differences, let's take a closer look at indentation syntax in Python vs other languages:

## Indentation

Here is some pseudo-code to indicate the use of whitespace and indentation in Python:

**Other Languages**

if (x)

if(y)

code-statement;

else

another-code-statement;

**Python**

if x:

if y:

code-statement

else:

another-code-statement

Note how Python is so heavily driven by code indentation and whitespace. This means that code readability is a core part of the design of the Python language.

# if, elif, else Statements

if Statements in Python allows us to tell the computer to perform alternative actions based on a certain set of results.

Verbally, we can imagine we are telling the computer:

"Hey if this case happens, perform some action"

We can then expand the idea further with elif and else statements, which allow us to tell the computer:

"Hey if this case happens, perform some action. Else, if another case happens, perform some other action. Else, if none of the above cases happened, perform this action."

Let's go ahead and look at the syntax format for if statements to get a better idea of this:

if case1:

perform action1

elif case2:

perform action2

else:

perform action3

## First Example

Let's see a quick example of this:

# 

# Let's add in some else logic:

# 

### Multiple Branches

Let's get a fuller picture of how far if, elif, and else can take us!

We write this out in a nested structure. Take note of how the if, elif, and else line up in the code. This can help you see what if is related to what elif or else statements.

We'll reintroduce a comparison syntax for Python.

# 

Note how the nested if statements are each checked until a True boolean causes the nested code below it to run. You should also note that you can put in as many elif statements as you want before you close off with an else.

Let's create two more simple examples for the if, elif, and else statements:

# 

## Indentation

It is important to keep a good understanding of how indentation works in Python to maintain the structure and order of your code. We will touch on this topic again when we start building out functions!

# 2.6 Functions

## Introduction to Functions

This lecture will consist of explaining what a function is in Python and how to create one. Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.

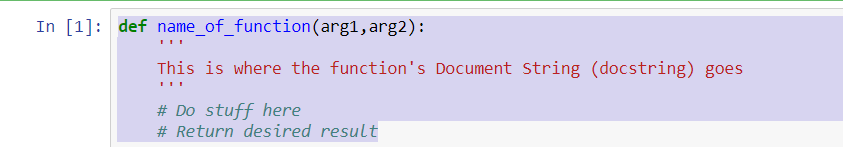
**So what is a function?**

Formally, a function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.

On a more fundamental level, functions allow us to not have to repeatedly write the same code again and again. If you remember back to the lessons on strings and lists, remember that we used a function len() to get the length of a string. Since checking the length of a sequence is a common task you would want to write a function that can do this repeatedly at command.

## def Statements

Let's see how to build out a function's syntax in Python. It has the following form:



We begin with def then a space followed by the name of the function. Try to keep names relevant, for example len() is a good name for a length() function. Also be careful with names, you wouldn't want to call a function the same name as a [built-in function in Python](https://docs.python.org/2/library/functions.html) (such as len).

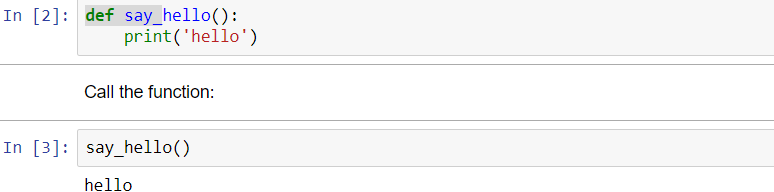
Next come a pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them. After this you put a colon.

Now here is the important step, you must indent to begin the code inside your function correctly. Python makes use of whitespace to organize code. Lots of other programing languages do not do this, so keep that in mind.

Next you'll see the docstring, this is where you write a basic description of the function. Using iPython and iPython Notebooks, you'll be able to read these docstrings by pressing Shift+Tab after a function name. Docstrings are not necessary for simple functions, but it's good practice to put them in so you or other people can easily understand the code you write.

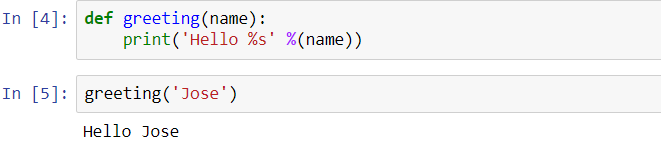
After all this you begin writing the code you wish to execute.

### Example 1: A simple print 'hello' function



### Example 2: A simple greeting function

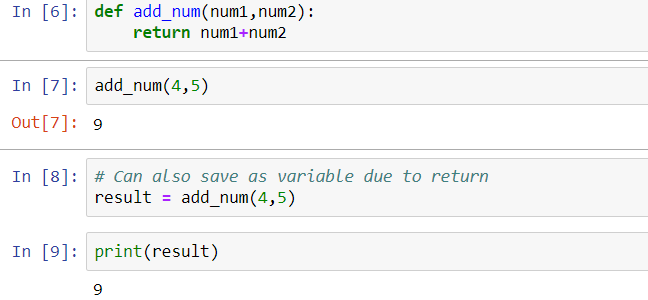
Let's write a function that greets people with their name.



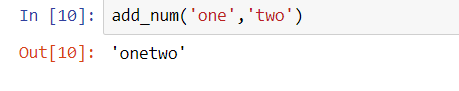
## Using return

Let's see some example that use a return statement. return allows a function to return a result that can then be stored as a variable, or used in whatever manner a user wants.

### Example 3: Addition function



What happens if we input two strings?

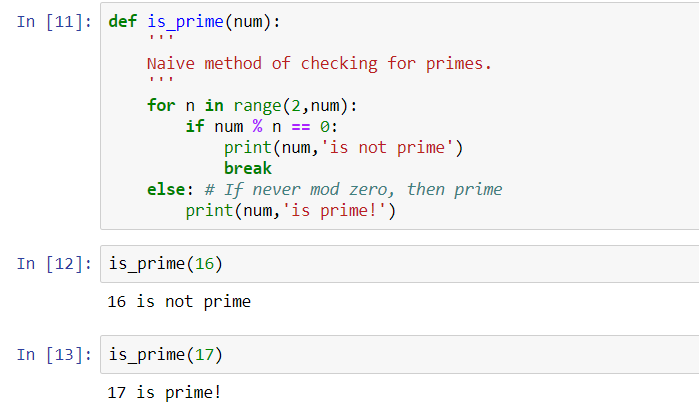


Note that because we don't declare variable types in Python, this function could be used to add numbers or sequences together! We'll later learn about adding in checks to make sure a user puts in the correct arguments into a function.

Let's also start using break, continue, and pass statements in our code. We introduced these during the while lecture.

Finally let's go over a full example of creating a function to check if a number is prime (a common interview exercise).

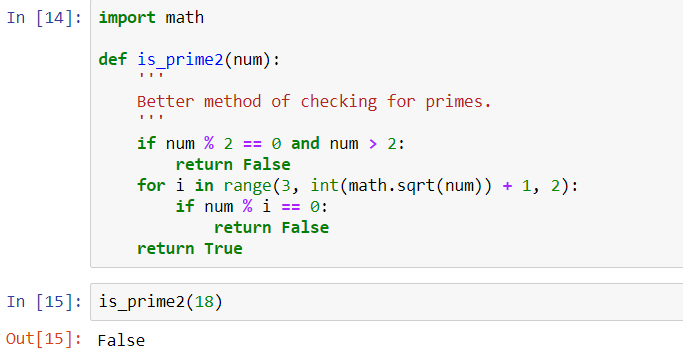
We know a number is prime if that number is only evenly divisible by 1 and itself. Let's write our first version of the function to check all the numbers from 1 to N and perform modulo checks.



Note how the else lines up under for and not if. This is because we want the for loop to exhaust all possibilities in the range before printing our number is prime.

Also note how we break the code after the first print statement. As soon as we determine that a number is not prime we break out of the for loop.

We can actually improve this function by only checking to the square root of the target number, and by disregarding all even numbers after checking for 2. We'll also switch to returning a Boolean value to get an example of using return statements:



# 2.7 Random Numbers in Python

Python defines a set of functions that are used to generate or manipulate random numbers. This particular type of functions are used in a lot of games, lotteries or any application requiring random number generation.

**Randon Number Operations :**

**1. choice()**:- This function is used to **generate 1 random number** from a container.

2. randrange(beg, end, step) :- This function is also used to generate random number but within a range specified in its arguments. This function takes 3 arguments, beginning number (included in generation), last number (excluded in generation) and step ( to skip numbers in range while selecting).

Example1

|  |
| --- |
| # Python code to demonstrate the working of  # choice() and randrange()    # importing "random" for random operations  import random    # using choice() to generate a random number from a  # given list of numbers.  print ("A random number from list is : ",end="")  print (random.choice([1, 4, 8, 10, 3]))    # using randrange() to generate in range from 20  # to 50. The last parameter 3 is step size to skip  # three numbers when selecting.  print ("A random number from range is : ",end="")  print (random.randrange(20, 50, 3)) |

Output:

A random number from list is : 4

A random number from range is : 41

3. random() :- This number is used to generate a float random number less than 1 and greater or equal to 0.

4. seed() :- This function maps a particular random number with the seed argument mentioned. All random numbers called after the seeded value returns the mapped number.

# Python code to demonstrate the working of

# random() and seed()

# importing "random" for random operations

import random

# using random() to generate a random number

# between 0 and 1

print ("A random number between 0 and 1 is : ", end="")

print (random.random())

# using seed() to seed a random number

random.seed(5)

# printing mapped random number

print ("The mapped random number with 5 is : ", end="")

print (random.random())

# using seed() to seed different random number

random.seed(7)

# printing mapped random number

print ("The mapped random number with 7 is : ", end="")

print (random.random())

# using seed() to seed to 5 again

random.seed(5)

# printing mapped random number

print ("The mapped random number with 5 is : ",end="")

print (random.random())

# using seed() to seed to 7 again

random.seed(7)

# printing mapped random number

print ("The mapped random number with 7 is : ",end="")

print (random.random())

Output:

A random number between 0 and 1 is : 0.510721762520941

The mapped random number with 5 is : 0.6229016948897019

The mapped random number with 7 is : 0.32383276483316237

The mapped random number with 5 is : 0.6229016948897019

The mapped random number with 7 is : 0.32383276483316237

1. **shuffle()** :- This function is used to **shuffle the entire list**to randomly arrange them.

# 2.8 Python - Math Module

# Some of the most popular mathematical functions are defined in the math module. These include trigonometric functions, representation functions, logarithmic functions, angle conversion functions, etc. In addition, two mathematical constants are also defined in this module

|  |  |
| --- | --- |
| List of Functions in Python Math Module | |
| **Function** | **Description** |
| ceil(x) | Returns the smallest integer greater than or equal to x. |
| copysign(x, y) | Returns x with the sign of y |
| fabs(x) | Returns the absolute value of x |
| factorial(x) | Returns the factorial of x |
| floor(x) | Returns the largest integer less than or equal to x |
| fmod(x, y) | Returns the remainder when x is divided by y |
| frexp(x) | Returns the mantissa and exponent of x as the pair (m, e) |
| fsum(iterable) | Returns an accurate floating point sum of values in the iterable |
| isfinite(x) | Returns True if x is neither an infinity nor a NaN (Not a Number) |
| isinf(x) | Returns True if x is a positive or negative infinity |
| isnan(x) | Returns True if x is a NaN |
| ldexp(x, i) | Returns x \* (2\*\*i) |
| modf(x) | Returns the fractional and integer parts of x |
| trunc(x) | Returns the truncated integer value of x |
| exp(x) | Returns e\*\*x |
| expm1(x) | Returns e\*\*x - 1 |
| log(x[, base]) | Returns the logarithm of x to the base (defaults to e) |
| log1p(x) | Returns the natural logarithm of 1+x |
| log2(x) | Returns the base-2 logarithm of x |
| log10(x) | Returns the base-10 logarithm of x |
| pow(x, y) | Returns x raised to the power y |
| sqrt(x) | Returns the square root of x |
| acos(x) | Returns the arc cosine of x |
| asin(x) | Returns the arc sine of x |
| atan(x) | Returns the arc tangent of x |
| atan2(y, x) | Returns atan(y / x) |
| cos(x) | Returns the cosine of x |
| hypot(x, y) | Returns the Euclidean norm, sqrt(x\*x + y\*y) |
| sin(x) | Returns the sine of x |
| tan(x) | Returns the tangent of x |
| degrees(x) | Converts angle x from radians to degrees |
| radians(x) | Converts angle x from degrees to radians |
| acosh(x) | Returns the inverse hyperbolic cosine of x |
| asinh(x) | Returns the inverse hyperbolic sine of x |
| atanh(x) | Returns the inverse hyperbolic tangent of x |
| cosh(x) | Returns the hyperbolic cosine of x |
| sinh(x) | Returns the hyperbolic cosine of x |
| tanh(x) | Returns the hyperbolic tangent of x |
| erf(x) | Returns the error function at x |
| erfc(x) | Returns the complementary error function at x |
| gamma(x) | Returns the Gamma function at x |
| lgamma(x) | Returns the natural logarithm of the absolute value of the Gamma function at x |
| pi | Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...) |
| e | mathematical constant e (2.71828...) |

# Examples

# 

# 

# 

# 