PYTHON WEEK 2

- SEQUENCES
- DICTIONARY MAPPING
- SETS
- LOOPS
- CATCH
- FUNCTIONS
- IMPORT & MODULES

References https://www.tutorialspoint.com/python/.

. . . .

PYTHON WEEK 2

Main Concept Of Sequences in Python

Among all sequence types, **Lists** are the most versatile. A list element can be any object. Lists are mutable which means they can be changed. Its elements can be updated, removed, and also elements can be inserted into it.

Tuples are also like lists, but there is one difference that they are immutable meaning they cannot be changed after defined.

Strings are little different than list and tuples, a string can only store characters. Strings have a special notation.

Following are the operations that can be performed on a sequence: –

- + operator combines two sequences in a process. it is also called concatenation. For example, [1,2,3,4,5] + [6,7] will evaluate to [1,2,3,4,5,6,7].
- * operator repeats a sequence a defined number of times. For example, [1,22]*3 will evaluate to [1,22,1,22,1,22].
- x in NewSeq returns True if x is an element of NewSeq, otherwise False. This statement can be negated with either not (x in NewSeq) or x not in NewSeq.
- NewSeq[i] returns the i'th character of NewSeq. Sequences in Python are indexed from zero, so the first element's index is 0, the second's index is 1, and so on.
- NewSeq[-i] returns the i'th element from the end of NewSeq, so NewSeq [-1] will the last element of NewSeq, NewSeq [-2] will be the second -last element.
- All sequences in python can be sliced.

Useful Functions on a sequence: –

- **len(NewSeq):** This returns the number of elements in the sequence NewSeq. Len stands for len
- lists are represented/created with square brackets with each item separated using commas. Example: -[a, b, c, d].
- Tuples are created by the comma operator, but they are not within square brackets. Enclosing parentheses are optional in tuples. However, an empty tuple must use an enclosing parenthesis. Example: – a, b, c or (). A tuple with single item ends with a trailing comma. Ex: – (d,).

SEQUENCES

- **String: –**

Slicing and dicing and indexing a string.

PYTHON WEEK 2

```
>>>"Hello, world!"[0] 'H'
>>>"Hello, world!"[1] 'e'
>>>"Hello, world!"[2] 'l'
>>>"Hello, world!"[3] 'l'
>>>"Hello, world!"[4] 'o'
>>"Hello, world!"[3:9] 'lo, wo'
>>>string = "Hello, world!"
>>>string[:5] 'Hello'
>>>string[-6:-1] 'world'
>>>string[-9:] 'o, world!'
>>>string[:-8] 'Hello'
>>>string[:] 'Hello, world!'
```

- **List: -**

Defining a list and indexing and appending it.

```
>>>spam =['bacon', 'chicken', 42]
>>>spam[0] 'bacon'
>>>spam[1] 'chicken'
>>>spam[2] 42
>>>len(spam)
3
>>>spam.append(10)
>>>spam
    ['bacon', 'chicken', 42, 10]
>>>spam.insert(1, 'and')
>>>spam
    ['bacon', 'and', 'chicken', 42, 10]
>>>spam
    ['bacon', 'and', 'chicken', 42, 10]
>>>del spam[1]
    >>>spam
    ['bacon', 'chicken', 42, 10] >>>spam[0] 'bacon'
    >>>spam[1] 'chicken'
    >>>spam[2] 42
    >>>spam[3] 10
```

Tuples: -

```
Various operations on a tuple.
>>>var = "me", "you", "them", “Their”
>>>var = ("me", "you", "them", “Their”)
>>>print var
('me', 'you', 'them', ‘Their’)
```

Apart from these, there are many other methods and functions are available that can be implemented on strings, lists, and tuple etc. Some such methods for strings are given below: –

- `Capitalize ()`
- `Center(width[, fillchar])`
- `count(sub[, start[, end]])`
- `decode([encoding[, errors]])`
- `encode( [encoding[,errors]])`
- `endswith( suffix[, start[, end]])`
- `expandtabs( [tabsize])`
- `find( sub[, start[, end]])`
- `index( sub[, start[, end]])`
- `isalnum( ) • islower( ) • isupper( )`
- `join( seq)`
- `replace(old, new[, count])`
- `startswith( prefix[, start[, end]])`
- `swapcase( )`

Conclusion

This topic provides a comprehensive understanding of sequences in python. It is expected that students understand the foundations of sequences and must practice given examples on a python IDE or console. From here, students can get ahead with their journey of python programming and if required look for additional practice material on the web or in python practice books. Python language is very much in demand nowadays and having good foundational understanding can benefit students a lot in their future endeavors.

PYTHON WEEK 2

DICTIONARY MAPPING

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.
Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'} print
"dict['Name']: ", dict['Name']
print "dict['Age']: ", dict['Age']
```

When the above code is executed, it produces the following result –

```
dict['Name']:  Zara
dict['Age']:  7
```

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example –

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry dict['School']
= "DPS School"; # Add new entry
 print "dict['Age']: ",
dict['Age']
print "dict['School']: ", dict['School']
```

When the above code is executed, it produces the following result – `dict['Age']:`

```
8
dict['School']:  DPS School
```

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.
To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example –

```
 dict = {'Name': 'Zara', 'Age': 7, 'Class':
'First'} del dict['Name']; # remove entry with key
'Name' dict.clear();     # remove all entries in
dict del dict ;         # delete entire dictionary
```

```
 print "dict['Age']: ",
dict['Age']
print "dict['School']: ", dict['School']
```

This produces the following result. Note that an exception is raised because after **del dict** dictionary does not exist any more − `dict['Age']:`

```
Traceback (most recent call last):
File "test.py", line 8, in <module>
print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

Note − del() method is discussed in subsequent section.

Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys. There are two important points to remember about dictionary keys –

(a) More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins. For example −

```
dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'} print
"dict['Name']: ", dict['Name']
```

When the above code is executed, it produces the following result −

```
dict['Name']:  Manni
```

(b) Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example −

```
dict = {['Name']: 'Zara', 'Age': 7}
print "dict['Name']: ", dict['Name']
```

SETS

Mathematically a set is a collection of items not in any particular order. A Python set is similar to this mathematical definition with below additional conditions.

- The elements in the set cannot be duplicates.
- The elements in the set are immutable(cannot be modified) but the set as a whole is mutable.
- There is no index attached to any element in a python set. So they do not support any indexing or slicing operation.

Set Operations

The sets in python are typically used for mathematical operations like union, intersection, difference and complement etc. We can create a set, access it's elements and carry out these mathematical operations as shown below.

Creating a set

A set is created by using the set() function or placing all the elements within a pair of curly braces.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
Months={"Jan","Feb","Mar"}
Dates={21,22,17}
print(Days) print(Months)
print(Dates)
```

When the above code is executed, it produces the following result. Please note how the order of the elements has changed in the result.

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
set(['Jan', 'Mar', 'Feb']) set([17, 21, 22])
```

Accessing Values in a Set

We cannot access individual values in a set. We can only access all the elements together as shown above. But we can also get a list of individual elements by looping through the set.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
  for d in
Days:
     print(d)
```

When the above code is executed, it produces the following result.

```
Wed
Sun
Fri
Tue
Mon
Thu
Sat
```

Adding Items to a Set

We can add elements to a set by using add() method. Again as discussed there is no specific index attached to the newly added element.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])

Days.add("Sun") print(Days)
```

When the above code is executed, it produces the following result.

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Removing Item from a Set

We can remove elements from a set by using discard() method. Again as discussed there is no specific index attached to the newly added element.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])

Days.discard("Sun") print(Days)
```

When the above code is executed, it produces the following result.

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Union of Sets

The union operation on two sets produces a new set containing all the distinct elements from both the sets. In the below example the element “Wed” is present in both the sets.

```
DaysA = set(["Mon","Tue","Wed"])
```

```
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA|DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Intersection of Sets

The intersection operation on two sets produces a new set containing only the common elements from both the sets. In the below example the element “Wed” is present in both the sets.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA & DaysB print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Wed'])
```

Difference of Sets

The difference operation on two sets produces a new set containing only the elements from the first set and none from the second set. In the below example the element “Wed” is present in both the sets so it will not be found in the result set.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA - DaysB print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Mon', 'Tue'])
```

Compare Sets

We can check if a given set is a subset or superset of another set. The result is True or False depending on the elements present in the sets.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
```

```
SubsetRes = DaysA <= DaysB
SupersetRes = DaysB >= DaysA
print(SubsetRes) print(SupersetRes)
```

When the above code is executed, it produces the following result.

```
True
True
```

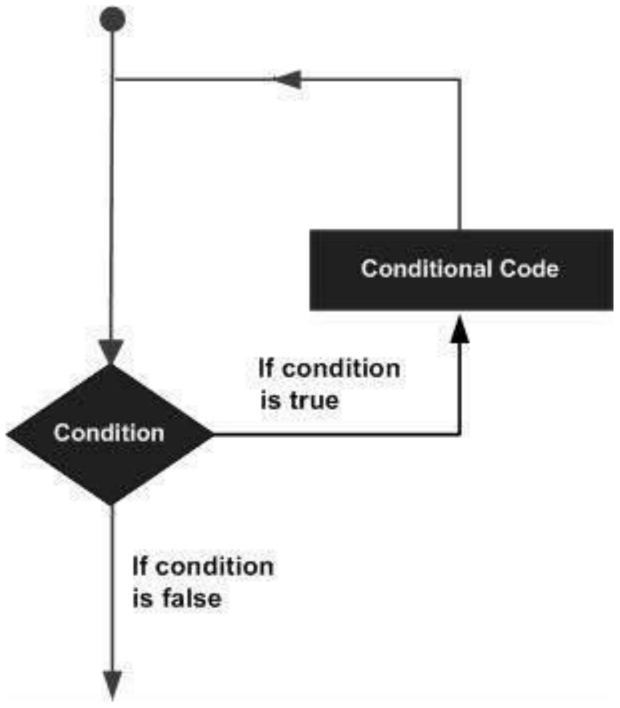
PYTHON WEEK 2

LOOPS

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −



Python programming language provides following types of loops to handle looping requirements.

```
vowels="AEIOU" for iter
in vowels:
print("char:", iter)
```

```
int_list = [1, 2, 3, 4, 5, 6]
sum = 0 for iter in int_list:
sum += iter print("Sum =",
sum)
print("Avg =", sum/len(int_list))
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements. Click the following links to check their detail.

Let us go through the loop control statements briefly

Sr.No.	Control Statement & Description
1	break statement Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	pass statement The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

PYTHON WEEK 2

CATCH

What is an Exception in Python?

An exception is an error which happens at the time of execution of a program. However, while running a program, Python generates an exception that should be handled to avoid your program to crash. In Python language, exceptions trigger automatically on errors, or they can be triggered and intercepted by your code.

The exception indicates that, although the event can occur, this type of event happens infrequently. When the method is not able to handle the exception, it is thrown to its caller function. Eventually, when an exception is thrown out of the main function, the program is terminated abruptly.

Common Examples of Exception:

- Division by Zero
- Accessing a file which does not exist.
- Addition of two incompatible types
- Trying to access a nonexistent index of a sequence
- Removing the table from the disconnected database server.
- ATM withdrawal of more than the available amount

Why should you use Exceptions?

Here are the reasons for using exceptions in Python:

- Exception handling allows you to separate error-handling code from normal code.
- An exception is a Python object which represents an error.
- As with code comments, exceptions helps you to remind yourself of what the program expects.
- It clarifies the code and enhances readability.
- Allows you to stimulate consequences as the error-handling takes place at one place and in one manner.
- An exception is a convenient method for handling error messages.

PYTHON WEEK 2

- In Python, you can raise an exception in the program by using the raise exception method.
- Raising an exception helps you to break the current code execution and returns the exception back to expection until it is handled.
- Processing exceptions for components which can't handle them directly.

Rules of Exceptions

Here are some essential rules of Python exception handling:

- Exceptions must be class objects
- For class exceptions, you can use try statement with an except clause which mentions a particular class.
- Even if a statement or expression is syntactically correct, it may display an error when an attempt is made to execute it.
- Errors found during execution are called exceptions, and they are not unconditionally fatal.

Exceptional Handling Mechanism

Exception handling is managed by the following 5 keywords:

1. try
2. catch
3. finally
4. throw

The Try Statement

A try statement includes keyword try, followed by a colon (:) and a suite of code in which exceptions may occur. It has one or more clauses.

During the execution of the try statement, if no exceptions occurred then, the interpreter ignores the exception handlers for that specific try statement.

In case, if any exception occurs in a try suite, the try suite expires and program control transfers to the matching except handler following the try suite.

Example:

```
Syntax: try:
statement(s)
```

The catch Statement

Catch blocks take one argument at a time, which is the type of exception that it is likely to catch. These arguments may range from a specific type of exception which can be varied to a catch-all category of exceptions.

Rules for catch block:

- You can define a catch block by using the keyword catch
- Catch Exception parameter is always enclosed in parentheses
- It always represents the type of exception that catch block handles.
- An exception handling code is written between two {} curly braces.
- You can place multiple catch block within a single try block.
- You can use a catch block only after the try block.
- All the catch block should be ordered from subclass to superclass exception.

Example:

```
try }
catch (ArrayIndexOutOfBoundsException e)
{
System.err.printin("Caught first " + e.getMessage());
}
catch (IOException e)
{
System.err.printin("Caught second " + e.getMessage());
}
```

Finally Block

Finally block always executes irrespective of an exception being thrown or not. The final keyword allows you to create a block of code that follows a try-catch block.

Finally, clause is optional. It is intended to define clean-up actions which should be that executed in all conditions.

```
try:
    raise KeyboardInterrupt
finally:     print
'welcome, world!'
Output
Welcome, world!
KeyboardInterrupt
```

Finally, clause is executed before try statement.

The Raise Statement

The raise statement specifies an argument which initializes the exception object. Here, a comma follows the exception name, and argument or tuple of the argument that follows the comma.

Syntax:

```
raise [Exception [, args [, traceback]]]
```

In this syntax, the argument is optional, and at the time of execution, the exception argument value is always none.

Example:

A Python exception can be any value like a string, class, number, or an object. Most of these exceptions which are raised by Python core are classes with an argument which is an instance of the class.

Important Python Errors

Error Type	**Description**
ArithmeticError	ArithmeticError act as a base class for all arithmetic exceptions. It is raised for errors in arithmetic operations.
ImportError	ImportError is raised when you are trying to import a module which does not present. This kind of exception occurs if you have made typing mistake in the module name or the module which is not present in the standard path.
IndexError	An IndexErroris raised when you try to refer a sequence which is out of range.
KeyError	When a specific key is not found in a dictionary, a KeyError exception is raised.
NameError	A NameError is raised when a name is referred to in code which never exists in the local or global namespace.

ValueError	Value error is raised when a function or built-in operation receives an argument which may be of correct type but does not have suitable value.
EOFerror	This kind of error raises when one of the built-in functions (input() or raw_input()) reaches an EOF condition without reading any data.
ZeroDivisonError	This type of error raised when division or module by zero takes place for all numeric types.
IOError-	This kind of error raised when an input/output operation fails.
syntaxError	SyntaxErrors raised when there is an error in Python syntax.
IndentationError	This error raised when indentation is not properly defined

Other Important Python Exceptions

Exception	**Description**
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment helps you to the array element of an incompatible type.
ClassCastException	Invalid cast
MlegalMonitorStateException	Illegal monitor operation, like waiting on an unlocked thread.
MlegalStateException	Environment or application is in wrong state.
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object which does not implement the Cloneable interface.
Illegal AccessException	Access to a class is denied.
InstantiationException	Occurs when you attempt to create an object of an interface or abstract class.
CloneNotSupportedException	Attempt to clone an object which does not implement the interface.

Error vs. Exceptions

Error	**Exceptions**
All errors in java are the unchecked type.	Exceptions include both checked and unchecked type.
Errors occur at run time which unknown to the compiler.	Exceptions can be recover by handling them with the help of try-catch blocks.
Errors are mostly caused by the environment in which an is running.	The application itself causes exceptions. application
Examples: Java.lang.StackoverflowError.java.lang.OutofMemoryError	Examples: Checked Exceptions, SQL exception,

NullPointerException,etc.

Summary

- An exception is an error which happened during the execution of a program.
- The exception indicates that, although the event can occur, this type of event happens infrequently.
- Common Examples of Exception are 1) Division by Zero, 2) Accessing a file which is not existent, 3) Addition of two incompatible types.
- An exception is a Python object which represents an error.
- A try statement includes keyword try, followed by a colon (:) and a suite of code in which exceptions may occur. It has one or more clauses.
- Catch blocks take one argument at a time, which is the type of exception that it is likely to catch.
- The raise statement specifies an argument which initializes the exception object.
- Finally, block always executes irrespective of an exception being thrown or not.

PYTHON WEEK 2

FUNCTIONS

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

```
def functionname( parameters ):
"function_docstring"
function_suite    return
[expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
   "This prints a passed string into this function"
print str
   return
```

PYTHON WEEK 2

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function –

```
# Function definition is here def
printme( str ):
   "This prints a passed string into this function"
print str    return;

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

When the above code is executed, it produces the following result –

```
I'm first call to user defined function!
Again second call to the same function
```

Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example –

```
# Function definition is here def
changeme( mylist ):
   "This changes a passed list into this function"
mylist.append([1,2,3,4]);
   print "Values inside the function: ", mylist
return

# Now you can call changeme function
mylist = [10,20,30]; changeme(
mylist );
print "Values outside the function: ", mylist
```

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result –

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
# Function definition is here def
changeme( mylist ):
   "This changes a passed list into this function"    mylist
= [1,2,3,4]; # This would assig new reference in mylist
   print "Values inside the function: ", mylist
return

# Now you can call changeme function
mylist = [10,20,30]; changeme(
mylist );
print "Values outside the function: ", mylist
```

The parameter *mylist* is local to the function changeme. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result −

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

Function Arguments

You can call a function by using the following types of formal arguments − •

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition. To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

```
# Function definition is here def
printme( str ):
   "This prints a passed string into this function"
print str    return;
```

```
# Now you can call printme function
printme()
```

When the above code is executed, it produces the following result −

```
Traceback (most recent call last):
File "test.py", line 11, in <module>
      printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways −

```
# Function definition is here def
printme( str ):
   "This prints a passed string into this function"
print str    return;

# Now you can call printme function printme(
str = "My string")
```

When the above code is executed, it produces the following result −

```
My string
```

The following example gives more clear picture. Note that the order of parameters does not matter.

```
# Function definition is here def
printinfo( name, age ):
   "This prints a passed info into this function"
print "Name: ", name    print "Age ", age
return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

When the above code is executed, it produces the following result −

```
Name:  miki
Age  50
```

PYTHON WEEK 2

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed –

```
# Function definition is here def
printinfo( name, age = 35 ):
   "This prints a passed info into this function"
print "Name: ", name    print "Age ", age
return;

# Now you can call printinfo function printinfo(
age=50, name="miki" )
printinfo( name="miki" )
```

When the above code is executed, it produces the following result –

```
Name:  miki Age
50
Name:  miki
Age  35
```

Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this –

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
function_suite    return
[expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example –

```
# Function definition is here def
printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
print "Output is: "    print arg1    for var
in vartuple:
      print var
return;

# Now you can call printinfo function printinfo(
10 )
printinfo( 70, 60, 50 )
```

When the above code is executed, it produces the following result –

```
Output is: 10
Output is:
70
60
50
```

The *Anonymous* Functions

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons. Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows – `lambda [arg1 [,arg2,.....argn]]:expression`

Following is the example to show how *lambda* form of function works –

```
# Function definition is here sum =
lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function print
"Value of total : ", sum( 10, 20 ) print
"Value of total : ", sum( 20, 20 )
```

When the above code is executed, it produces the following result –

```
Value of total :  30
Value of total :  40
```

The *return* Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.
All the above examples are not returning any value. You can return a value from a function as follows –

```
# Function definition is here def
sum( arg1, arg2 ):
   # Add both the parameters and return them."
total = arg1 + arg2
   print "Inside the function : ", total
return total;

# Now you can call sum function total
= sum( 10, 20 );
print "Outside the function : ", total
```

When the above code is executed, it produces the following result –

```
Inside the function :  30
Outside the function :  30
```

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.
The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python – • Global variables

- Local variables

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.
This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example –

```
 total = 0; # This is global
variable. # Function definition is
here def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2; # Here total is local variable.
print "Inside the function local total : ", total    return
total;

# Now you can call sum function sum(
10, 20 );
print "Outside the function global total : ", total
```

When the above code is executed, it produces the following result −

```
Inside the function local total :  30
Outside the function global total :  0
```

PYTHON WEEK 2

IMPORT / EXPORT

What's an import?

To use any package in your code, you must first make it accessible. You have to import it. You can't use anything in Python before it is defined. Some things are built in, for example the basic types (like `int`, `float`, etc) can be used whenever you want. But most things you will want to do will need a little more than that. Try typing the following into a Python console:

```
oTime = datetime.datetime.now()
```

You'll get a `NameError`. Apparently Python doesn't know what `datetime` means - `datetime` is not defined. You'll get a similar result for doing this:

```
print my_undefined_variable
```

For `datetime` (or anything really) to be considered defined, it has to accessible from the current scope. For this to be true it has to satisfy one of the following conditions:

- it is a part of the defaultPython environment. Like `int`, `list`, `__name__` and `object`. Try typing those in an interpreter and see what happens
- it has been defined in the current program flow (as in you wrote a `def` or a `class` or just a plain 'ol assignment statement to make it mean something. This statement is a bit of a simplification and I'll expand on it pretty soon
- it exists as a separate package and you imported that package by executing a suitable import statement.

Try this out:

```
import datetime
oTime = datetime.datetime.now() print
oTime.isoformat()
```

That worked a whole lot better. First we import `datetime`. Then we use the `now` function to create an object and we assigned it to the `oTime` variable. We then can access functions and attributes attached to `oTime`.

Importing the `datetime` package made it accessible in the current scope.

Ok... but what's a scope?

PYTHON WEEK 2

The word *scope* seems to invoke a lot of fear in new developers so I'll talk a little bit about it here. The basic idea is that if you want to make use of any object at any point in a program, that object needs to be defined before you use it. Some things are always defined no matter where you are in your program (eg: `int`), these are said to exist in the global scope.

Other things need to be defined in an explicit way. This implies actually executing a statement or statements to define the object - to give it a name in the current scope. If you have a bunch of accessible objects then those objects are accessible through use of their names, an object exists in a scope if it has been given a name within that scope. And giving something a name always implies actually executing a statement. Now that I've paraphrased myself a million times, let's move on to some examples.

```
print x   # NameError
x   = 3     # associates the name `x` with 3 print x
# works fine this time
```

Packages, classes and functions work in the same way. Take a look at the `datetime` examples above.

So that's pretty straightforward, right? Wrong! There is still the matter of enclosing scopes.

Try this out...

```
y   = 3                                  # first we
associate the name 'y' with the number 3

def print_stuff():              # then we associate the name
print_stuff with this function
    print "calling print_stuff"          #  (***)
    print y
z = 4
    print z
print "exiting print_stuff"

     print_stuff()                        # we call print_stuff and the
                                        program execution goes to (***)
print y                                 # works fine print
z                                 # NameError!!!
```

So `y` was defined outside `print_stuff` and was accessible both inside and outside `print_stuff`. And z was defined inside `print_stuff` and was only accessible within print_stuff. There are two separate scopes here!

So let's extend our understanding to: an object exists in a scope if it has been given a name within that scope **OR an enclosing scope**.

Now for something a little different... if you are using an interpreter, put this in a fresh one (exit and open a new one).

```
 # no more y here...
 def
print_stuff():
    print "calling print_stuff"
print y                     z =
4           print z
    print "exiting print_stuff"

print_stuff()     # NameError. this shouldn't surprise you since
we haven't yet set y
y = 3             # only now do we associate the name 'y' with
the number 3
print_stuff()     # the rest works fine print
y
```

I like to think of scopes in terms of nested dictionaries. The picture I'm drawing here is not entirely accurate, but it's a nice way of thinking about things...

```
program_scope = {
        'y'  : 3,       # y was defined outside of print_stuff,
#         and is accessible inside and outside
        'print_stuff' : {
            'z' : 4              # z was defined inside of
print_stuff,
                                  #           and is ONLY
accessible inside
        },
}
```

Brace yourself the last example for this section...

```
a    = [1,2,3]               # first we give some names to some
stuff,
b    = [4,5,6]               #     thus making it accessible in
this scope and enclosing scopes c = [8,9,10]
 def
do_things():
a = "new a"
b.append(7)
c = "new c"
print a
print b
print c
 do_things()              #nothing surprising in the
output...
 print a                  # [1,2,3]
print b                  # [4, 5, 6, 7]
print c                  # [8,9,10]
```

When executing a statement that can create a new name in the scope, Python just goes ahead and makes it happen. When adding something to a scope Python does not look at enclosing scopes. But when trying to access an object Python will check the current scope and all enclosing scopes recursively until it either finds it or runs out of scopes (and raises a `NameError`).

Drawing out the scope hierarchy as a dictionary like before, we have something like this:

```
program_scope = {
    'a' : [1,2,3],
    'b' : [4,5,6,7],
'c' : [8,9,10],
do_things: {
       'a' : 'new_a',    #the enclosed scope has its own a and c
but no b!!!
       'c' : 'new c'
    }
}
```

The import mechanism and the scope, and what's a package anyway

A package is just a directory tree with some Python files in it. Nothing magical. If you want to tell Python that a certain directory is a package then create a file called `__init__.py` and just stick it in there. Seriously, that's all it takes. To make a package that is actually useful you would need to do somewhat more. A package can contain other packages but to be useful it must contain modules somewhere in it's hierarchy. A module is just a script that contains a bunch of definitions and declarations (you define classes and functions, and declare variables). The whole point of this system is to be able to organise code in such a way as to make it easy to leverage existing code in new projects without excessive use of copy-paste.

The Python documentation has a wonderful tutorial on the subject that can fill you in on the finer points.

In this section we'll move through the basic motions of creating and using packages and modules.

To start off, make yourself a directory to work in (I'll call it your working directory for the remainder of this text) and follow the instructions below. Now create a file called `my_funcs.py` that looks a little something like:

```
def print_a():
print "a"
     def
print_b():
print "b"
```

Now open up a console and `cd` into your newly created working directory and launch the Python interpreter. Type in the statements below:

```
print my_funcs         # NameError
import my_funcs        # from this point on my_funcs is in the
scope
print my_funcs         # prints information about the module
my_funcs.print_a()     # call the function print_a that is a
member of the my_funcs module my_funcs.print_b()
```

Isn't that exciting? To get to something contained directly within a module you can just import the module and use a dot. But it looks a little verbose, exit the interpreter and launch a fresh one and try this out:

```
from my_funcs import print_a,print_b      # this time we add the
two functions to the scope       print_a()         # it works!
print_b()
print my_funcs    # this raises a NameError since we didn't make
#              my_funcs accessable, only its members
```

PYTHON WEEK 2

Now create a directory within your working directory called my_pkg and move my_funcs.py into my_pkg. Create an empty file called `__init__.py` and stick it in there too. Now you have a directory structure like this:

```
    working_dir\
my_pkg\
__init__.py
my_funcs.py
```

Exit and relaunch your interpreter and type in the following:

```
print my_pkg               # NameError print
my_funcs           # NameError

import my_pkg              # my_pkg is not defined in the scope
 print my_pkg               # isn't that
nice print my_funcs            # NameError
print my_pkg.my_funcs      # module information

my_pkg.my_funcs.print_a() #works fine my_pkg.my_funcs.print_b()
```

Again, all the dots are a little verbose. We can do things a little differently. Play with these:

```
from my_pkg import my_funcs

from my_pkg.my_funcs import print_a, print_b

from my_pkg import my_funcs.print_a, my_funcs.print_b

from my_pkg.my_funcs import print_a as
the_coolest_function_ever, print_b as not_the_brightest_crayon
```

The take home message here is that you can be very specific about what parts of a package or module you want to import, and can even decide what names they will be given in the scope.

Now exit the interpreter and `cd` out of your working directory and launch a new interpreter. Try importing from `my_pkg`. You can't. And that sucks. But we can import `datetime`. So what gives?

sys.path gives

Python is very aware of what is known as the *current working directory*. If you try to import something then the current working directory is where it looks first, where it looks second, third and fourth depend a bit on your python installation. Try this:

```
import sys print
sys.path
```

This prints out a list of directories, when you tell Python to import something then it looks in each of the listed locations in order. The first item in the list is an empty string, that indicates the current working directory. If you were to make a package or module in your current directory and name it `datetime` then Python's standard `datetime` functionality would be out of reach.

Try this out:

```
sys.path.append('/path/to/your/working/directory') #the
directory that contains my_pkg import my_pkg
```

Brilliant! Now we can use `my_pkg` again.

But it would be kind of annoying to have to add new entries to `sys.path` for every package we want to make use of, it would be better to just store Python packages at standard locations and point the path there. Luckily Python's package installation mechanisms handle that sort of thing. It's a bit outside the scope of this text though. The point here is that you can have any number of packages made available to your script through use of `sys.path` and that is terribly convenient. Except when it isn't...

Version conflicts

Let's assume for a moment that version conflicts are horrible things. What if you've gone and started working on two different projects (A and B) that require two different sets of packages, say project A requires you to install C1 and C2; and project B requires you to install packages D1 and D2. And that's it. Nice and tidy. There are no conflicts there.

But what if C1 requires E version 2 and project B requires E version 3. You could just continually install and uninstall different versions of E as needed whenever you need to run the scripts... but that sounds really horrible.

You could package the correct E versions within the projects that need them, ie in their respective working directories... but what if there are dependency issues you aren't yet aware of, they could spring up the next time you install any new dependency.

Considering your shiney new knowledge of the import path, we could put each of the E versions in differnt locations and then alter `sys.path` to include the correrct location. For example in A we would need to do something like:

```
import sys sys.path.append('path/to/E_version_2')
```

Which might seem clever at first but really isn't. That sort of approach would make installing our packages tedious and brittle as everything would need to be put in the right place. And anyway, it does not address the problem of future issues.

Version conflicts are a pretty horrible problem to have to solve, good thing we don't have to. Enter virtual environments.

Virtual Environments

A virtual environment is a group of files and configuration that can be activated (and deactivated). A virtual environment is associated with it's own python executable, and it's own sys.path and thus it's own combination of installed packages.

Here is how I usually kick off a new project (this is bash, not Python):

```
virtualenv venv                 # 1. creates the virtual
environment
source venv/bin/activate        # 2. activates the virtual
environment # whatever you want
deactivate  # 3. deactivates the virtual environment  (you can
also just close the terminal)
```

Line 1 creates a virtual environment called `venv`. It's just a directory structure containing a bunch of Python stuff and some configuration (including a new `sys.path`). This command has a lot of options, you can even pick which Python you are keen on including in the environment (if you have multiple Python versions installed on your system)

Line 2 activates the environment. If the environment is active and you launch Python then you'll be launching the Python interpreter that lives inside the virtual environment. If you install a package while the environment is active then the package will not go to the place where system wide packages go, it will rather get installed inside the environment (directory structure).

Line 3 deactivates the environment and makes things normal again. All the stuff you installed in your environment will be accessible the next time you activate it.

Conclusion

We've covered quite a lot here. We started off with the basics of scope, then proceeded to packages and the import mechanism. We then covered how virtual environments could be used to overcome version conflicts. But the rabbit hole goes a whole lot deeper. If you are interested in taking the import system further then it would be worth checking out the `__import__` built in function, and `import importlib`. Also, there is more to just regular import statements (you can move up a package tree instead of down it, this is occasionally quite useful). You may have noticed the appearance of .pyc files during our experimentations and those are pretty cool on their own. We also touched on the fact that Python has standard package installation mechanisms, these are really worth knowing about if you intend to deploy or give away any significant piece of code.