Python for Raspberry Pi Users

Dr. Sarwan Singh
Deputy Director(S)
NIELIT Chandigarh

sarwan singh
@imsarwansingh

Assembly was grandfather, C is father, Python is young,smart nerd of the town. Falling in love is mandatory #Python #CodePython

12:25am - 5 Nov 15 from Chandigarh, India

References

- http://www.linuxjournal.com/article/3882 "Why Python?" by Eric Raymond in the *Linux Journal*, May 1st 2000.

- http://www.tcl.tk/doc/scripting.html "Scripting: Higher Level Programming for the 21st Century", John K. Ousterhout, *IEEE Computer*, March 1998.

- http://www.geeksaresexy.net/2008/07/25/the-ascent-of-scripting-languages/ "The ascent of scripting languages" by Chip. July 25, 2008. From the site Geeks are Sexy ™

- Python web page: http://www.python.org/
  Quick Links (3.3.2) → Documentation → Tutorial

What is a Scripting Language?
Definition 1

- Wikipedia: A **scripting language**, **script language** or **extension language**, is a programming language that controls a software application. "Scripts" are often treated as distinct from “programs", which execute independently from any other application.

What is a Scripting Language?

Definition 2

- "**scripting language**." Encyclopædia Britannica. Encyclopædia Britannica Online. 23 Feb. 2012 http://www.britannica.com/EBchecked/topic/130670/computer-programming-language/248127/Scripting-languages "Scripting languages are sometimes called little languages. They are intended to solve relatively small programming problems that do not require the overhead of data declarations and other features needed to make large programs manageable. Scripting languages are used for writing operating system utilities, for special-purpose file-manipulation programs, and, because they are easy to learn, sometimes for considerably larger programs."

Early Scripting Languages (definition 1 style)

- Controlled the actions of system programs
- Job Control Language (JCL):controlled batch job executions on IBM mainframes
- Shell scripts: A series of instructions to the shell, or command line interpreter, of a UNIX system
- When used like this, scripting languages take pre-existing components, maybe in different languages, and combine them so they work together.

Modern Scripting Languages

- Perl, Python, Ruby, …
- More like a traditional programming language than early scripting languages.
- Higher level in the sense that programmers don't have to worry about many of the details that are a part of C-like languages
  - No explicit memory allocation/deallocation, for example
- These languages are examples of the second definition

Modern Scripting Languages

- It's possible to write full-featured programs in these languages
- Used frequently today for "quick-and-dirty" programming –
  - Quick results
  - Small programs; e.g., system utilities
- Python advocates claim it is more scalable than Perl and other scripting languages – easy to write large programs.

Why Scripts are Usually Interpreted

- Using an interpreter instead of a compiler makes sense when programs change frequently and/or are very interactive.
  - Reason: no extra code generation time
- In the scripting language-as-glue-language mode, performance is dominated by the modules being connected by the scripting commands

Why Python?

- No universal agreement on which scripting language is best
- Perl, Ruby, Python all have advocates
- Perl: good for system administration duties, traditional scripting
- Python – has attracted many former Perl users
- Ruby – syntactically similar to Perl and Python; popularized because it was used to write Ruby on Rails, a framework for web apps.

Python - Intro

- Python is a general purpose scripting language that implements the imperative, object-oriented, and functional paradigms.
- Characteristics include dynamic typing, automatic memory management, exceptions, large standard library, modularity.
    - Extensions can be written in C and C++
    - Other language versions (Jython, IronPython) support extensions written in Java and .Net languages)
- Design philosophy: easy to read, easy to learn

Versions

- Current production versions are 2.x.x and 3.x.x
- Both versions are stable and suitable for use
- Python 2: compatible with much existing software
- Python 3: relatively major redesign
  - Not backward compatible.
  - Most features are the same or similar, but a few will cause older programs to break.
  - Part of the Python philosophy – don’t clutter up the language with outdated features

Interesting Features

- White space <u>has syntactic/semantic significance</u>
  - Instead of curly brackets or begin-end pairs, whitespace is used for block delimiters.
  - Indent statements in a block, un-indent at end of block.
- Statements are terminated by `<Enter>`
- No variable declarations
- Dynamic typing
- Associative arrays (dictionaries)
- Lists and slices, tuples – no array type.

Execution Modes - Calculator

Type in an expression
Python will evaluate it

```
>>> a = 5
>>> b = 10
>>> a + b
15
>>> x = a + b
>>> x
15
```

Dynamic type change

```
>>> a = 'horse'
>>> b = ' cart'
>>> a + b
'horse cart'
>>> a + x
Traceback (most recent call last):
     File "<pyshell#10>",line 1,in <module>
    a + x
 TypeError: Can't convert 'int' object to str implicitly
```

Execution Modes - Program

```
>>> #factorial.py
>>> #compute factorial
>>> def main( ):
  n = int(input("enter an int "))
  fact = 1
  for factor in range (n, 1, -1):
      fact = fact * factor
  print("the answer is: ", fact)


>>> main( )
enter an int 5
the answer is:  120
>>>
```

Execution Modes - Program

- You can also create the program in a text file, adding the statement

  ```
  main()
  ```

  as the last statement, and then run it later.
- Instructions for creating, testing, and executing Python programs will be posted on K drive with the notes. There will also be sample programs you can experiment with.

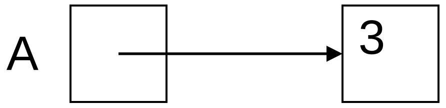
Program Elements

- Identifiers:
  - Must begin with letter or underscore, followed by any number of letters, digits, underscores
- Variables:
  - Are not declared (although a comment/declaration is good practice)
  - A variable is created when a value is assigned to it:
    Examples: `num = 3`
  - Don’t use a variable name in a statement until it has been assigned a value
    - Error message: *Name Error* – means no value is associated with this name

Variables

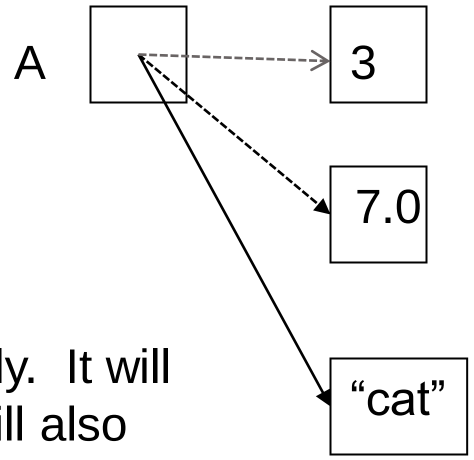
- Variable names don’t have static types – values (or objects) do
  - A variable name has the type of the value it currently references
- Variables actually contain references to values (similar to pointers).
  - This makes it possible to assign different object types to the same variable

Variables contain references to data values



A = A * 2.5

A = "cat"



Python handles memory management automatically. It will create new objects and store them in memory; it will also execute garbage collection algorithms to reclaim any inaccessible memory locations.

Python does not implement reference semantics for simple variables; if A = 10 and B = A, A = A + 1 does not change the value of B

Basic Data Types

- Numeric types: integer, floats, complex
- A literal with a decimal point is a float; otherwise an integer
- Complex numbers use "j" or "J" to designate the imaginary part: `x = 5 + 2j`
- `type()` returns the type of any data value:

Data Types

```
>>> type(15)
<class 'int'>
>>> type (3.)
<class 'float'>
>>> x = 34.8
>>> type(x)
<class 'float'>
```

```
>>> a = "What's for
 dinner?"
>>> type(a)
<class 'str'>
>>>
```

Expressions

- An expression calculates a value
- Arithmetic operators: +, -, *, /, **, %
  (** = exponentiation)
- Add, subtract, multiply, divide work just as they do in other C-style languages
- Spaces in an expression are not significant
- Parentheses override normal precedence

Expressions

- Mixed type (integer and float) expressions are converted to floats:

```
>>> 4 * 2.0 /6
1.3333333333333333
```

- Mixed type (real and imaginary) conversions:

```
>>> x = 5 + 13j
>>> y = 3.2
>>> z = x + y
>>> z
(8.2 + 13J )
```

- Explicit casts are also supported:

```
>>> y = 4.999              >>> x = 8
>>> int(y)                      >>> float(x)
4                              8.0
```

Output Statements
3.x version

- Statement syntax: (EBNF notation)

```
        print ({expression,} )
```

where {*expression,*} is a list of print descriptors & the final comma in the list is optional.

- Examples :

```
print() # prints a blank line
print(3, y, z + 10)  # go to new line
print('result: ',z,end=" ")#no newline
```

- Output items are automatically separated by a single space.

Assignment Statements

- Syntax: *Assignment* $\longrightarrow$ *variable* = *expression*
- A variable's type is determined by the type of the value assigned to it.
- Multiple_assign $\longrightarrow$*var* {, *var*} = *expr* {, *expr)*

```
>>> x, y = 4, 7
>>> x
4
>>> y
7
>>> x, y = y, x
>>> x
7
>>> y
4
>>>
```

Interactive Input

- Syntax: *input* $\longrightarrow$ *variable = input(string)*
  The string is used as a prompt.
  Inputs a string

```
>>> y = input("enter a name --> ")
enter a name --> max
>>> y
'max'
>>> number = input("Enter an integer ")
Enter an integer 32
>>> number
'32'
```

- `input()` reads input from the keyboard as a string;

Input

- To get numeric data use a cast :

```
>>> number = int(input("enter an integer:"))
enter an integer:87
>>> number
87
```

- If types don’t match (e.g., if you enter 4.5 and try to cast it as an integer) you will get an error:

ValueError: invalid literal for int() with base 10: ‘4.5’

Input

Multiple inputs:

```
>>> x, y = int(input("enter an integer: ")),
              float(input("enter a float: "))
enter an integer: 3
enter a float: 4.5
>>> print("x is", x, " y is ", y)
x is 3  y is  4.5
```

Instead of the cast you can use the *eval( )* function and Python chooses the correct types:

```
>>> x, y = eval(input("Enter two numbers: 
  "))
Enter two numbers: 3.7, 98
>>> x, y
(3.7, 98)
```

Python Control Structures

- Python loop types:
  - while
  - for
- Decision statements:
  - if
- Related features:
  - range( )   # a function used as a loop control
  - break   # statements similar to those in C/C++
  - continue

General Information

- All control structure statements must end with a colon (:)
- The first statement in the body of a loop must be indented.
- All other statements must be indented by the same amount
- To terminate a loop body, enter a blank line or “unindent”.

While Loop

```
>>> x = 0
>>> while (x < 10):   # remember the colon!
        print(x, end = " ")      #no new line
        x = x + 1

0 1 2 3 4 5 6 7 8 9
```

Conditions are evaluated just as in C/C++: 0 is false, non-zero is true

The conditional operators are also the same

Note indentation – provided by the IDLE GUI

For Loops

- Syntax:

```
for <var> in <sequence>:
    <body>
```

- `<sequence>:` can be a list of values or it can be defined by the `range( )` function
  - range(n) produces a list of values: 0, 1, …,n-1
  - range(start, n): begins at start instead of 0
  - range(start, n, step): uses step as the increment
-

```
>>> for i in range(3):  # C++ equivalent: for (i=0; i<3; i++)
               print(i,end = " ")
0 1 2
>>> for i in range(5,10):
                print(i,end = " ")
5 6 7 8 9
>>> for i in range(6,12,2):
                print(i)
6
8
10
>>> for i in (1, 'a', 3.8, [1, 2]): #using a tuple as LCV
       print(i)
1
a
3.8
[1, 2]
>>> for i in (1, 'a', 3.8, [1, 2]):
       print ('*', end = " ")

* * * *
>>>
```

Ranges can also be specified using expressions:

```
>>> n = 5
>>> for i in range(2*n + 3):
        print(i,end = " ")


0 1 2 3 4 5 6 7 8 9 10 11 12
```

Using a List in a `for` loop

Lists are enclosed in square brackets

```
>>> for i in [3, 2, “cat”]:
            print(i,end = " ")


3 2 cat
>>> list1 = [1, 'a']
>>> for i in list1:
     print(i, end = " ")


1 a
>>>
```

If Statement

- Python `if` statement has three versions:
  - The `if` (only one option)
  - The `if-else` (two options)
  - The `if-elif-else` (three or more options)
- The `if-elif-else` substitutes for the `switch` statement in other languages.
- Each part must be followed with a colon and whitespace is used as the delimiter.

If-elif-else Statement

```
>>>x = int(input("enter an integer: “))
if x < 0:
        print (‘Negative’)
elif x == 0:
     print ('Zero‘)
elif x == 1:
     print ('Single')
else:
     print ('More' )
```

Right Way

```
>>> if x < 0:
            y = x
elif x == 0:
            y = 1
elif x < 10:
            y = 100
else:
            print("none”)
```

(you may have to override the IDLE indentation)

Wrong Way

```
>>> if x < 0:
            y = x
    elif x == 0:
            y = 1
    elif x < 10:
            y = 100
    else:
            print("none ")
```

SyntaxError: unindent does not match any outer indentation level (<pyshell#86>, line 3)

Python Data Types/Data Structures

- Many built-in simple types: ints, floats, infinite precision integers, complex, etc.
- Built-in data structures:
  - Lists or dynamic arrays
  - Tuples
    - Similar to lists, but immutable (cannot be modified)
    - Sometimes used like C/C++ structs, but indexed by position instead of field name
  - Strings (like tuples, indexed and immutable)
  - Dictionaries (also known as associative arrays or hash tables)
    - Each entry is a key with an associated value
    - Access is by the key in the key-value pair

String Data Type

- A sequence of characters enclosed in quotes (single or double; just be consistent)
- `import string`

  includes the string library in your program

String Operations

- Elements can be accessed via an index, but you can’t change the individual characters directly (strings are immutable, like tuples)
- Indexing: 0-based indexing
- General format: `<string>[<expr>]`

```
>>> name = 'bob jones'
>>> name[0]
'b'
>>> name[0] = 'r`  #attempted assignment
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    name[0] = 'r'
TypeError: 'str' object does not support item assignment
```

- Negative indexes work from right to left:

```
>>> myName = "Joe Smith"
>>> myName[-3]
`i'
```

How to Get Around Restrictions

```
>>> text
'Now is the time for Every   Good man   to come InsiDE .'
>>> text = text.lower()
>>> text
'now is the time for every   good man   to come inside .‘
```

You can modify strings in controlled ways by using one of the string functions, which returns a string modified according to the function. To keep the new version, assign it to a variable.

String Data Type Examples

```
>>> str1 = 'happy'
 >>> str2 = "Monday‟
 >>> str2[0]
 'M'                  # indexing the string
 >>> x = str1, str2
 >>> x
 ('happy', 'Monday')   # x is a tuple
 >>> x[1]              # indexing the tuple
 'Monday'
 >>> x[1][2]     # indexing tuple & element
 'n'
```

- Tuples are immutable (can't be changed), like strings

String Data Type - continued

```
>>> print(str1, str2)
Happy Monday
>>> "I don't like hotdogs"
"I don't like hotdogs"
>>> 'I don't like hotdogs'
>>>
```

Syntax Error: invalid syntax

Use double quotes when you want to include single quotes (and vice versa)

String Data Type

- More examples:

```
>>> '"why not?" said Jim'
'"why not?" said Jim'
>>> 'can\'t'
"can't"
```

You can get the same effect by using the escape sequence \’ or \”

String Operations

- Slicing: selects a 'slice' or segment of a string *<string>[<start>:<end>]* where *<start>* is the beginning index and *<end>* is one past final index

```
>>> myName = "Joe Smith"
>>> myName[2:7]
'e Smi`
```

- If either *<start>* or *<end>* is omitted, the start and end of the string are assumed

```
>>> myName[:5]
'Joe S'
>>> myName[4:]
'Smith'
>>> myName[:]
'Joe Smith'
```

String Operations

Concatenation (+)

```
 >>> "happy" + "birthday"
'happybirthday'
 >>> 'happy' + ' birthday'
 'happy birthday'
```

Repetition (*)

```
>>> word = 'ha'
>>> 3 * word
'hahaha'
>>> word + 3 * '!'
'ha!!!'
```

String Operations

Other examples:

```
>>> word = 'ha'
>>> len(word)   # length function
2
>>> for ch in myName:
          print(ch, end = " ")

J o e   S m i t h
```

Lists

- A list is a comma-separated sequence of items, enclosed in square brackets
- Lists can be heterogeneous – items don’t have to be from the same data type
- Like strings, lists can be sliced, indexed, concatenated, and repeated.
- The `len()` function will return the number of elements in the list

Lists

- Unlike strings & tuples, lists are mutable (can be modified by element assignment)
  - Make an assignment, using the index

```
>>> myList = ['milk','eggs','bread']
>>> myList[1] = 'butter'
>>> myList
['milk', 'butter', 'bread']
```

- You can also assign to slices, and even change the length of the list by inserting or deleting elements.

List Slice Operations

```
#create a list and assign it to a variable
>>> data = ['bob', 32, 'sue', 44]
>>> data
['bob', 32, 'sue', 44]
#assign to a list slice
>>> data[1:3] = ['dave', 14]
>>> data
['bob', 'dave', 14, 44]
#insert an element (or several)
>>> data[1:1] = [19]
>>> data
['bob', 19, 'dave', 14, 44]
#delete an element
>>> data[3:4] = []
>>> data
['bob', 19, 'dave', 44]
```

Python Lists

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[1] = 6
>>> b
[1, 6, 3]
>>> a = [6, 7, 8]
>>> b
[1, 6, 3]
>>>
```

Python Lists

Like C/C++, multi-dimensioned arrays are represented as lists of lists:

```
>>> twoDList = [[1, 2, 3], [4, 5, 6]]
>>> twoDList[1][2]
6
>>> twoDList[1] # access by rows (sublist)
[4, 5, 6]
```

Adding to a List

- You can grow the list dynamically with the concatenation operator:

```
>>> x = [2, 4, 6, 8]
>>> x
[2, 4, 6, 8]
>>> x = x + [10]
>>> x
[2, 3, 6, 8, 10]
```

List Insertion by Slicing

```
>>> x = [2,3,6,8.10]
# slice/concatenate to get a new list
>>> x = x[:2]+[100,200] + x[2:]
>>> x
[2, 3, 100, 200, 6, 8.1]
>>> x = x[:4] + []
>>> x
[2, 3, 100, 200]
```

Nested Lists

```
>>> grades = [100, 97, 85]
>>> stRec = ['A000','jack',grades]
>>> stRec
['A000', 'jack', [100, 97, 85]]
>>> len(stRec)
3
```

Additional String Operations

- `split`: divides a string into a list of substrings

```
>>> myStr = 'The fat black cat'
>>> myStr.split()
['The', 'fat', 'black', 'cat']
```

- `split` defaults to blank as the delimiter, but you can specify a different character:

```
>>> myStr = '12/10/2008'
>>> myStr.split('/')
['12', '10', '2008']
```

Strings

- After you have split a string into a list of substrings, you may want to convert some of the substrings to specific data types.
  - specific casts: *int( ), float( ), long( ),* and str( )
  - If you don’t know the data types, you can use the generic cast *eval( )*

Example

```
>>> mysplitStr
['12', '10', '2008']
>>> first = eval(mysplitStr[0])
>>> first
12
>>> #etc. – you can use the type function to
  determine if the list elements are the
  type you expected:
>>> x = 3.4
>>> if type(x) == float:
  print('float')

float
```

More About Lists

- See Section 5.1 in the Python tutorial to get a whole set of list functions:
  - append(x)
  - insert(i, x)
  - etc.
- Since lists are objects, dot notation is used to call the functions

List Functions

```
>>> stRec = ['A000', 'jack', grades]
>>> stRec.remove(grades)
>>> stRec
['A000', 'jack']
>>> stRec.append([100, 97, 85])
>>> stRec
['A000', 'jack', [100, 97, 85]]
>>> stRec.pop(2)#removes item at index
[100, 97, 85]
>>> stRec
['A000', 'jack']
```

Python Tuples

- A tuple is a sequence of comma-separated values:

```
>>> t =('A000','jack',3.56,'CS')
```

- Tuples are similar to strings, except they are immutable (cannot assign to individual elements)
- Like lists and strings, tuples can be sliced and concatenated

Tuple Example

```
>>> t = ('A000', 'jack', 3.56, 'CS')
>>> t1 = ('A001', 'jill', 2.78, 'MA')
>>> t2 = ('A0222', 'rachel', 3.78, 'CS')
>>> students = [t, t1, t2]
>>> for (ID, name, GPA, major) in students:
          print(ID, name, GPA, major)

A000 jack 3.56 CS
A001 jill 2.78 MA
A0222 rachel 3.78 CS
>>>
```

For each iteration, one tuple in the list of tuples is unpacked into the individual variables

Sequences

- Strings, lists, and tuples are all examples of the *sequence* data type.
- Operations that can be performed on sequences:

<seq> + <seq>	concatenation
<seq> * <int-exp>	repetition
<seq>[ ]	indexing
<seq>[:]	slicing
len(<seq>)	length
for <var> in <seq>	iteration

- Only the list sequence type is directly modifiable

Dictionaries

- A dictionary is an example of a *mapping* object (currently, the only Python example).
- Some languages call them associative arrays or hashes.
- Unlike sequences (lists, strings, tuples) which are indexed by an ordered range of values, dictionaries are indexed by *keys*
- Items are retrieved according to their keys.

Dictionary Entries

- Dictionaries consist of <key, value> pairs
- The key is a unique identifier (student #, account #, SSN, etc.) and the value is whatever data might be associated with the key; e.g., name, address, age, etc.
  - The value can be a single value, or it can be a list, or string, or tuple, or …
- The primary purpose of a dictionary is to provide a data structure with direct add/retrieve operations
  - Compare to an array or list which must be searched linearly (if not sorted) or sorted (to use binary search)

Dictionary Characteristics

- A key must be unique within a given dictionary object.
- Keys must be hashable; i.e., they cannot contain lists, dictionaries, or other mutable objects. Strings and simple values are often used as keys. Tuples can also be used as keys as long as they don't contain lists, dictionaries, …

More about Dictionaries

- List: an ordered collection of elements
- Dictionary: an <u>unordered</u> collection of elements
  - items aren't stored sequentially or in order of entry; storage order is determined some other way, probably a hash function
  - Other characteristics:
    - Mutable (add, delete elements or change them)
    - Variable length
    - Cannot be sliced or concatenated, because these operations depend on having an <u>ordered</u> set of elements.

Creating Dictionaries

Create an empty dictionary:

```
>>> roll = { }
```

Create and initialize a dictionary:

```
>>> roll = {1023: 'max', 404: 'sue'}
```

Retrieve an item by its key:

```
>>> roll[1023]
'max'
```

Add an item:

```
>>> roll[9450] = 'alice'
>>> roll
{9450:'alice', 404:'sue', 1023:'max'}
```

Alternate Dictionary Creation

Using the dict() function:

```
>>> tel = dict(max = (94, 'x'), alice =
  (5, 'y'))
>>> tel
{‘max': (94, 'x'), ‘alice': (5, 'y')}
```

To get a list of the keys:

```
>>> list(roll.keys())
[9450, 404, 1023]
```

To get the list in sorted form:

```
>>> x = sorted(roll.keys())
>>> x
[404, 1023, 9450]
```

To find out if a key is in the dictionary:

```
>>> 2600 in roll  # for lookup or insertion
False
```

To remove a key

```
>>> del roll[404]
>>> roll
{9450: 'alice', 1023: 'max'}
```

To change an element’s value:

```
>>> roll[9450] = ('alice', 3.45, 'CS')
>>> roll
{9450:('alice', 3.45, 'CS'), 1023:'max'}
```

Sort on the value field if sortable:

```
>>> numd = {34: 56, 45:101, 906: 25, 100: 3}
>>> numd
{34: 56, 100: 3, 45: 101, 906: 25}
>>> sorted(numd.values())
[3, 25, 56, 100]
>>> sorted(numd.keys())
[34, 45, 101, 906]
```

Looping Through Dictionaries

You can retrieve the key and the value at the same time:

```
>>> d = dict(‘max’ = 37, ‘joe’ = 409,
             ‘cal’ = 100)
>>> for name, code in d.items():
          print(name, code)

max 37
cal 100
joe 409
```

Notice the items are processed in stored order, not the order in which they were entered

If the dictionary value is a list:

- Index like a two-dimensional array to get one list item.

```
>>> box = {}
>>> box[36] = [2, 'cat', 3.5]
>>> box
{36: [2, 'cat', 3.5]}
>>> box[36][1]
'cat'
```

Python Functions

- A Python function can be either void or value returning, although the definition isn't identified as such.
  - Technically, they are all value returning, since a void function returns the value **`none`**
- Each function defines a scope. Parameters and local variables are accessible in the scope; values in other functions are available only as parameters.

Function Syntax

Function definition

```
def <name> (formal-parameters>):
      <body>
```

Function call syntax:

```
<name>(<arguments>)
```

```
>>> def square(x):
          return x * x

>>> square(10)
100
>>> z = 23
>>> square(z * 4)
8464
>>> x = square(z)
>>> x
529
```

You can enter functions at the command line and use in calculator mode, or include them in programs, or in modules (similar to libraries).

An example of a program file

```
#File: chaos.py
#A simple program illustrating chaotic behavior
#Insert other function definitions here
def main( ):
    print("a chaotic function“)
    x=eval(input("enter a number betw 0 and 1: "))
    for i in range (10):
        x = 3.9 * (1 - x)
        print(x)
#Alternate location for function definitions

main()  #call main function first
```

Functions That Return Values

```
>>> def sum(x, y):
      sum = x + y
      return sum

>>> num1,num2 =
  3,79
>>> sum(num1, num2)
82
```

```
>>> def SumDif(x,y):
      sum = x + y
      dif = x - y
      return sum, dif

>>> x, y = 7, 10
>>> a,b = SumDif(x,y)
>>> a
17
>>> print(a, b)
17 -3
>>>
```

Parameters

- Parameters are passed by value, so in general the actual parameters can’t be changed by the function – only their local copies
- Mutable objects (e.g., lists) act like reference parameters
  - the value passed is a pointer to a list of pointers, and these pointers remain the same; only the things pointed to change!
  - (don’t worry about the subtleties here)

```
>>> def incre(list):
     n = len(list)
     for i in range(n):
          print(i)
          list[i] = list[i] + 1

>>> myList = [2, 4, 6]
>>> incre(myList)
0
1
2
>>> print(myList)
[3, 5, 7]
>>>
```

File I/O

- To open a file:

  `<filevar> = open(<filename>, <mode>)`

  where **`filename`** is the name of the file on disk, and **`mode`** is usually **`'r'`** or **`'w'`**
  - **`infile = open("data.in", "r")`**
  - **`outfile = open("data.out", "w")`**
- Files must also be closed:
  - **`infile.close( )`**
  - **`outfile.close( )`**
- Put data file in same folder as program file or use a path name.

Input Operations

- **`<filevar>.read`**: returns the remaining contents of the file as a multi-line string (lines in the file separated by \n)
- **`<filevar>.readline( ):`** returns next line as a string; includes the newline character (you can slice it off, if needed)
  This is the function you'll probably use the most of the three here.
- **`<filevar>.readlines( ):`** returns the remaining lines in the file, as a list of lines (each list item includes the newline character)

Detecting End of File (from the documentation)

- f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string, and is omitted on the last line of the file only if the file doesn’t end in a newline. This makes the return value unambiguous;
- if f.readline() returns an empty string, the end of the file has been reached, while a blank line is represented by '\n', a string containing only a single newline.

Detecting End of File (from the documentation)

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
` `
```

Reading a File Line-by-Line

```
>>> infile=open("C:\Temp\Test.txt",'r')
>>> for line in infile:
        myString = line
        print(myString)
This is the first line of the file.
Second line of the file
```

Stops when there are no more lines in the file – easiest way to process an entire file.

File Output

`f.write(string)` writes the contents of **`string`** to the file, returning the number of characters written.

```
>>> f.write('This is a test\n') 15
```

To write something other than a string, convert it to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> s
 "('the answer:', 42)"
```

Formatting Output

- “Fancier Output Formatting”
- New features in versions 3.x
- *repr():* converts something to a string for easier printing
- Other functions: *rjust()* – right justify
  - *ljust()*
  - *center()*

File Output

```
>>>ostring = str(part).ljust(10) +
             str(price).ljust(20)+ "\n"
>>>f.write(ostring)
```

Output with *repr()*

```
>>> outfile=open("C:\Temp\exmp.out", "w")
>>> outfile.write("this is the first line\n")
23
>>> s =("the value of x is " + repr(x) + ",
  and y is " + repr(y) + '...')
>>> outfile.write(s)
39   #39 characters written
>>> outfile.close()
```

```
File contents:
this is the first line
the value of x is 14, and y is 3.149...
```