Python Summary

Table of Contents

[Python Summary 3](#_Toc88735068)

[Basic Python commands 3](#_Toc88735069)

[Arithmetic, input, and output 3](#_Toc88735070)

[Casting 3](#_Toc88735071)

[Printing 5](#_Toc88735072)

[See Python Printing 5](#_Toc88735073)

[Basic Print 5](#_Toc88735074)

[Conditional commands 6](#_Toc88735075)

[While loop 7](#_Toc88735076)

[Sequences (strings and tuples) and the for-command 7](#_Toc88735077)

[Lists 11](#_Toc88735078)

[Lists can be nested inside lists. 13](#_Toc88735079)

[Dictionaries 14](#_Toc88735080)

[Functions 15](#_Toc88735081)

[Syntax and semantics of functions 15](#_Toc88735082)

[Modules 17](#_Toc88735083)

[There are two main uses of modules: 17](#_Toc88735084)

[Files 18](#_Toc88735085)

[Classes and objects 19](#_Toc88735086)

# Python Summary

# Basic Python commands

|  |  |  |
| --- | --- | --- |
| COMMAND | DESCRIPTION | EXAMPLES |
| print DATA | displays the DATA in the window | print "Game Over"  print (3 \* 2) - 1 |
| raw\_input (STRING) | displays the STRING on the display and waits for the human (``user'') to type something on the keyboard and press Enter | raw\_input("Please type something") |
| # COMMENT (DOCUMENTATION) STATEMENT | gives one line of explanation about the program for a human to read | # This Game is Over |
| arithmetic expressions using integers, addition, subtraction, multiplication, and division | tells Python to compute the answer of the expression | 3 + 2 - 1  3 \* (2 - 4) |
| STRING + STRING | concatenates two strings into one | "abc" + "d" |

# Arithmetic, input, and output

## Casting

|  |  |
| --- | --- |
| Casting means **changing the data type of a piece of data from one type to another**. The data may be stored inside a variable. ... You can cast a piece of data by specifying the data you have, and the data type you want to cast it to. This code asks the user to type in a number, then doubles that number. | In programming, the conversion of one data type into another; for example, from an integer to a string or vice versa. The casting statement in the source code of a program causes the compiler to generate the machine code that performs the actual conversion. |
| Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types. | Casting in python is therefore done using constructor functions:  **int()** - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)  **float()** - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)  **str()** - constructs a string from a wide variety of data types, including strings, integer literals and float literals |

Here is a summary of what we have learned so far:

A Python PROGRAM is a sequence of COMMANDs, one per line. The COMMANDs are executed one after the other.

A COMMAND can be

* an ASSIGNMENT, which has the syntax,  
  1. VARIABLE = EXPRESSION

EXAMPLES:

minutes = hours \* 60 and

hours = int(raw\_input("Type an int: "))

The semantics of an assignment goes in two steps:

* 1. The namespace is checked to see if there is a cell for VARIABLE. If there is not, one is constructed (but there is no value in the cell).
  2. The value of the EXPRESSION is computed.
  3. The value is assigned to VARIABLE's cell in the namespace.
* a print command,  
  1. print EXPRESSION\_COMMAS

where EXPRESSION\_COMMAS is a sequence of expressions separated by commas.   
  
EXAMPLE: print "minutes are", (hours \* 60), "."

The semantics is that

* 1. the expressions are evaluated one by one, from left to right
  2. one string is assembled from the answers of the expressions
  3. the string is displayed in the command window

There are many possible forms of EXPRESSION:

* strings, built from letters enclosed in quotes, from variables that have string values, and from + (EXAMPLE: "abc" + "3" + phrase, where phrase = "hi")
* int expressions, built from integers, variables that have int values, arithmetic operators, +, -, \*, /, %, //, and parentheses (EXAMPLE: (hours \* 60) + 2)
* float expressions, built from floats, variables that have float values, arithmetic operators, +, -, \*, /, and parentheses (EXAMPLE: ((9.0/5.0) \* fahrenheit) + 32.0)
* answers computed by these Python *functions*:
  + raw\_input(STRING): prints STRING and receives a string from the user in reply
  + int(STRING): computes the int that is represented by STRING
  + float(STRING): computes the float that is represented by STRING

and these two *methods*:

* + STRING.upper(): makes a string that looks just like STRING, but in all upper case
  + STRING.lower(): makes a string that looks just like STRING, but in all lower case

Remember that a method, like phrase.upper(), can be understood as the function, upper(phrase).

A Python program has a meaning (*semantics*): When a program is started, it is copied into computer storage, along with the Python interpreter and an empty *namespace*. The Python interpreter reads the program's commands, one by one, and tells the CPU what to do to compute the commands. The program's variables are constructed in the namespace.

# Printing

## See Python Printing

## Basic Print

x = 5

y = "John"

z = 4

print (f'x=' , x, (type(x)))

print (f'y=', y, (type(y)))

print (f'z=', z, (type(z)))

print (f'x+z=', x+z)

# Conditional commands

Here is a summary of the new Python constructions:

The new COMMAND is the CONDIIONAL, which can have these forms of syntax:

if CONDITION :

COMMANDs

if CONDITION :

COMMANDs1

else :

COMMANDs2

if CONDITION1 :

COMMANDs1

elif CONDITION2 :

COMMANDs2

elif CONDITION3:

COMMANDs3

...

else :

COMMANDsLAST

That is, a conditional consists of one or more CONDITION--COMMANDs pairings, optionally finished by an else : COMMANDs. The semantics of the conditional goes like this:

1. The CONDITIONs are computed one by one until a condition is located that computes to True.
2. The COMMANDs that follow the condition's colon are executed.
3. If all the conditions compute to False, then the COMMANDs that follow the else : are executed. (If there is no else, then no commands at all are executed.)

*It is important that all the COMMANDs are indented exactly the same amount, e.g., 4 spaces or one tab.*

A CONDITION is an expression that computes to a boolean (True-False) value. CONDITIONs can be built in these ways:

* EXPRESSION OP EXPRESSION, where the EXPRESSIONs are numerical values, and OP is from >, <, >=, <=, ==, !=
* CONDITION LOGICAL\_OP CONDITION, where the LOGICAL\_OP can be and, or, not. Parentheses are used as needed.
* a VARIABLE, where the variable holds a boolean (True-False) value. An example was if ok : ....

The semantics of a CONDITION is that it is evaluated from left to right until its result is definitely True or False.

# While loop

The syntax is:

while CONDITION :

COMMANDs

The semantics goes as follows:

1. The CONDITION is computed.
2. If CONDITION computes to True, then COMMANDS execute and *the process repeats, restarting at Step 1.*
3. If CONDITION computes to False, then the COMMANDs are ignored, and the loop terminates.

To print an integer a fixed number of places, use this trick:

num = . . .

import string

print string.zfill(num, PLACES)

where PLACES is a nonnegative int. For example,

cents = 4

print string.zfill(cents, 3)

prints 004 --- leading zeroes are added. (When the number is larger than the number of places, the number prints correctly, anyway.)

## Sequences (strings and tuples) and the for-command

There is a new COMMAND, the for-loop; its syntax is

for VARIABLE in SEQUENCE :

COMMANDs

where VARIABLE is a variable name and SEQUENCE is an expression that computes to a sequence. (See below.) The semantics of the for-loop goes like this:

1. The SEQUENCE is computed into a sequence, call it S.
   * VARIABLE is assigned S[0], and the COMMANDs are executed.
   * VARIABLE is assigned S[1], and the COMMANDs are executed.  
     . . .
   * VARIABLE is assigned S[len(S)-1] (the last element in S), and the COMMANDs are executed.

We also learned about sequences. At the moment, we know of two forms of them:

A SEQUENCE is either

* a STRING, or
* a TUPLE, where a tuple has the form,
* ()
* or
* (EXPRESSION,)
* or
* (EXPRESSION1, EXPRESSION2, ..., EXPRESSIONn)
* where n > 1

The elements of a sequence are numbered (indexed) by 0, 1, 2, ....

Here are the operations that can be applied to sequences:

* length:
* len(SEQUENCE)

computes to the integer length of SEQUENCE.

* indexing:
* SEQUENCE[EXPRESSION]

where EXPRESSION computes to a nonegative integer, m, extracts the element numbered by m in the SEQUENCE.

There are also these general forms of indexing, which can extract multiple elements from a sequence:

* + up to (and not including) element number n:
  + SEQUENCE[:EXPRESSION]

where EXPRESSION computes to nonnegative integer, n.

* + element number m onwards:
  + SEQUENCE[EXPRESSION:]

where EXPRESSION computes to nonnegative integer, m.

* + all elements from n up to (and not including) m:
  + SEQUENCE[EXPRESSION1:EXPRESSION2]

where EXPRESSION1 computes to nonnegative integer, n and EXPRESSION2 computes to nonnegative integer, m.

(Note: Python also lets you use negative integers for indexing. Negative integers count from the *right* instead of the left of the sequence. We won't use these.)

* concatenation:
* SEQUENCE1 + SEQUENCE2

builds a new sequence whose elements are exactly the ones of SEQUENCE1 followed by SEQUENCE2.

* There is a CONDITION operation,
* EXPRESSION in SEQUENCE

which computes to True when the value named by EXPRESSSION is an element within SEQUENCE. (Otherwise, it computes to False.)

* Strings can be compared with the usual arithmetic comparisons, ==, !==, <=, <, >=, >. The answers are computed using dictionary (*lexicographic*) ordering.
* str(EXPRESSION) is a function that converts a number into a string.

Finally, here are many more useful operations on strings:

* S1.lower() builds a string that looks just like S1, but all letters in S1 are changed to to lower case. (E.g., for name = "JanE12", the method name.lower() builds the string, "jane12".)

Similarly, S1.upper() makes a string whose letters are upper case, e.g., for name = "JanE12", name.upper() builds the string, "JANE12".

* S1.capitalize() capitalizes the first letter of string S1 (e.g., for name = "jane", name.capitalize() builds "Jane").

Similarly, S1.title() capitalizes the first letter of each ``word'' in S1, case, e.g., for book = "The 4 corners of Earth", book.title() builds , "The 4 Corners Of Earth".

* S1.strip() builds a new string that removes leading and trailing blanks, tabs, and newline characters from S1. For example, for name = " jane \n", the method name.strip() builds the string, "jane".
* We can test if a string is all letters, numbers, both, or spaces:
  + S1.isalpha() computes to True if string S1 is all letters (alphabetic characters)
  + S1.isdigit() computes to True if string S1 is all numbers. This is especially useful for checking user-supplied input:
  + data = raw\_input("Please type an integer: ")
  + if data.isdigit() :
  + num = int(data)
  + . . .
  + else :
  + print "error --- nonnumeric input"
  + S1.isalnum() computes to True if string S1 is a mix of only letters and numeric digits.
  + S1.isspace() computes to True if string S1 is one or more space characters

We can convert between single letters and integers:

* + If S1 is a string that is a single character, ord(S1) computes to the internal, ASCII, integer code for the integer. (E.g., ord("a") computes to 97 and ord(" ") computes to 32.)
  + If n is an integer, chr(n) computes to the ASCII character that n encodes, e.g., chr(97) computes to 'a'.

To print an integer a fixed number of places, use this trick:

num = . . .

import string

print string.zfill(num, PLACES)

where PLACES is a nonnegative int. For example,

cents = 4

print string.zfill(cents, 3)

prints 004 --- leading zeroes are added. (When the number is larger than the number of places, the number prints correctly, anyway.)

* S1.find(S2) searches for the string, S2, within the string S1 and returns the index number where the string was first found. (If S2 is not found in S1, then -1 is returned.) For example,
* line = "abcdecd"
* pattern = "cd"
* print line.find(pattern)

prints 2, since "cd" first begins at index 2 in line. We can use find to rewrite the FindChar program in Figure 5 like this:

# FindChar

# locates the leftmost occurrence of a character in a string.

# assumed inputs: s - the string to be searched

# c - the character to be found

# guaranteed output: if c is in s, then c's position is printed

# if c is not in s, then -1 is printed

s = raw\_input("Type a string: ")

c = raw\_input("Type a single char to search for: ")

c = c[0] # in case the user typed some extra blanks, extract the first char

print s.find(c)

All the hard work is done by find!

* S1.replace(PATTERN, REPLACEMENT) builds a new string, which looks like string S1, except that all occurrences of string PATTERN are replaced by a copy of string REPLACEMENT. For example,
* line = "abcabdabe"
* new\_line = line.replace("ab", "!")

assigns the string, "!c!d!e" to new\_line.

# Lists

Lists are similar to arrays (but lists can grow as needed).

A list is written with this syntax:

[ ELEMENTS ]

where ELEMENTS are zero of more EXPRESSIONs, separated by commas. Each expression can compute to any value at all --- number, boolean, string, tuple, list, etc. The elements in a list are saved in the list's cells which are indexed (numbered) by 0, 1, 2, ....

A list can be assigned to a variable, as usual:

**gameboard = [ "", "", "", "", "", "", "", "" ]**

We can use a shortcut to make a list whose items start with the same value:

**gameboard = [ "" ] \* 8**

Given a list, LIS, we can compute its length (number of cells) like this:

**len( LIS )**

The primary operation on lists is indexing, and we can write this indexing expression:

LIS [ INT\_EXPRESSION ]

where LIS is a list, and INT\_EXPRESSION is an expression that computes to an integer that falls between 0 and len( LIS ) -1. The expression returns the element saved at cell number INT\_EXPRESSION.

We update a list's cell with this assignment command:

**LIS [ INT\_EXPRESSION ] = EXPRESSION**

The assignment destroys the value that was formerly held at cell INT\_EXPRESSION and replaces it with the value of EXPRESSION.

Since a list is a SEQUENCE, we can use the for-loop to systematically process a list's elements:

**for ITEM in LIS :**

**... ITEM ...**

where ITEM is a variable name that stands for each element from list LIS.

There is a special operation, range, that constructs a list of numbers: range(NUM) builds the list, [0, 1, 2, ..., NUM-1]. (E.g., range(3) computes [0, 1, 2].) We can use range to print a list's index numbers and contents:

**for index in range(len(LIS)) :**

**print index, LIS[index]**

Here are two methods that alter lists:

* **LIS.append( EXPRESSION )** adds the value of EXPRESSION to the end of list LIS, increasing the list's length by one.
* **LIS.sort()** reorders the elements in LIS into ascending (alphabetical/numeric) order

And here is a useful method for breaking an input string into a list of the words it holds:

* **STR.split( SEPARATORS )** builds a new list which is string STR split into a list of its words, where the blanks within STR separate the words. For example,
* s1 = " this is a sentence."
* words = s1.split()
* print words

prints ['this', 'is', 'a', 'sentence.']

* **STR.split( SEPARATOR )** builds a new list which is string STR broken into those words separated by string SEPARATOR. For example,
* time = "12:35:49"
* words = time.split(":")
* print words

prints ['12', '35', '49'].

## Lists can be nested inside lists.

Here is how to build an 8-by-8 chessboard, a *matrix*:

**board = [ ]**

**for row in range(8) :**

**board.append([ "" ] \* 8)**

We assign to individual cells of the board with two indexes:

**board[5][5] = ("white", "pawn")**

and we use nested for-loops to print the board:

**for row in board :**

**for square in row :**

**print square, "|",**

**print**

# Dictionaries (key:value pairs)

Dictionaries are similar to record structures (but dictionaries can grow as needed and the key set is not fixed in advance).

A dictionary is written with this syntax:

**{ KEY\_ELEMENT\_PAIRS }**

where KEY\_ELEMENT\_PAIRS are zero or more **KEY : ELEMENTs,** separated by commas. Each KEY must be an immutable value (number, boolean, string, or tuple). Each ELEMENT can compute to any value at all --- number, boolean, string, list, dictionary, etc. The elements in a list are saved in a *hash table* structure.

A dictionary can be assigned to a variable, as usual.

Given a dictionary, DICT, we can find an element by using its key:

**DICT [ KEY ]**

But if KEY is not found in DICT, it is an error, so it is better to ask first if the KEY is present:

**if KEY in DICT :**

**... DICT[KEY] ...**

We update a dictionary with this assignment command:

**DICT [ KEY ] = EXPRESSION**

If the KEY is new to DICT, then a new key, element pair is added. If the KEY is already in use, the assignment destroys the value that was formerly associated with KEY and replaces it with the value of EXPRESSION.

We can use the for-loop to systematically process a dictionary:

**for K in DICT :**

**... K ... DICT[K] ...**

where K is a variable name that stands for each key saved in dictionary DICT.

The operation, del DICT[K] deletes key K and its element from DICT. (If K is not in DICT, it is an error.)

Here are two useful methods for dictionaries:

* **DICT.keys(),** which returns a list that holds all the keys saved in the dictionary.
* **DICT.items(),** which returns a list that holds all the elements saved in the dictionary.

Finally, here is a simple way to print the contents of a dictionary ordered by the keys:

**keylist = my\_dictionary.keys()**

**keylist.sort()**

**for k in keylist:**

**print k, ":", my\_dictionary[k]**

# Functions

## Syntax and semantics of functions

A function definition has the form,

**def NAME(PARAMETERS) :**

**DOC\_STRING**

**COMMANDs**

where

* NAME is a name, like a variable name.
* PARAMETERS are zero or more variable names, separated by commas
* DOC\_STRING (this is optional) is an indented (multi-line) string, bracketed by triple double-quotes, describing the purpose of the function and its parameters.
* COMMANDs is an indented sequence of one or more commands, which can include the new command form,
* return EXPRESSION

The semantics of a function definition is that the function's name, its parameters, and its commands are saved in the current namesapce.

A function call (invocation) has the format,

NAME(ARGUMENTS)

where

* NAME is the name of a previously defined function
* ARGUMENTS are zero or more EXPRESSIONs.

The semantics of a function call operates as follows:

1. The arguments are computed to answers, one by one, from left to right.
2. The function's name and its body are located in the namespace.
3. Execution at the position of the function call is paused.
4. The arguments are assigned, one by one, to the parameters listed in the function's definition.
5. The commands within the function are executed.
6. When the commands finish, execution resumes at the position where the function was called.

When a return EXPRESSION command is executed within the function's body, the expression part is computed to an answer, the function immediately terminates, and the answer is inserted exactly in the position where the function was called.

A function can also be called with keyword arguments.

A function, F's documentation string can be printed with the command, print F.\_\_doc\_\_

Functions can be written and testing separately, using interactive testing:

1. Place the function, fred, in a file by itself, say, Test.py.
2. Open a command window and start the Python interpreter.
3. Within the interpreter, type these two commands:
4. >>> import Test
5. >>> from Test import \*
6. Interactively test the function in Test.py --- type
7. >>> fred(..arguments...)

This executes the function, just as if it was called from within a program.

1. If you change the coding of fred, you can retest it by reloading the file that contains it:
2. >>> reload(Test)
3. >>> from Test import \*
4. >>> fred(..arguments...)

There are three important uses of functions:

1. doing one small thing that must be done over and over from different places in a program. This is called *bottom-up programming*.
2. doing one important thing that must be designed and tested by itself before it is inserted into the rest of the program. This is called *top-down programming*.
3. doing one thing to a global variable (data structure) to help keep the global variable correctly updated. This is called *modular programming*.

# Modules

Every program is a *module*, and one program can execute another by importing it. The command forms are

* import MODULE: This executes the code in MODULE, constructs its namespace in computer storage, and defines the name, MODULE, so it can be later referenced.

A variable defined in MODULE is referenced as MODULE.NAME, and a function defined in MODULE is referenced as MODULE.NAME(ARGUMENTS).

* from MODULE import NAMES\_SEQUENCE  
  where NAMES\_SEQUENCE is a sequence of function names separated by commas: This executes the code in MODULE, constructs its namespace in computer storage, and defines all names in NAMES\_SEQUENCE so that they can be referenced, just as if they were defined locally.
* from MODULE import \*: This executes the code in MODULE, constructs its namespace in computer storage, and defines all names in MODULE so that they can be referenced, just as if they were defined locally.
* reload(MODULE): This reloads MODULE, replacing its existing namespace by a new one.

## There are two main uses of modules:

* A program that does its computation in a sequence of stages (modules).

The benefit is that each module is reasonably sized and can be read and understood in one sitting.

* A program that divides its computation into a module that holds a data structure with its maintenance operations and a module that holds the algorithm that controls how the user computes on the data structure.

The benefit is that the data-structure module is self contained and can be readily improved or replaced. In particular, *all detailed maintenance of the data structure is contained in this one module and nowhere else.* The module can be reused in another program if needed.

# Files

The example program below shows the key ideas.

FIGURE================================================

**# This program makes a copy of one sequential file into another.**

**# Ask the user for the names of the files to read from and to write to:**

**inputfilename = raw\_input("Type input file to copy: ")**

**outputfilename = raw\_input("Type output file: ") # this can be a name of a**

**# file that does not yet exist**

**# A file must be opened --- this requires a new variable to hold the**

**# address in heap storage of the file's data-structure representation:**

**input = open(inputfilename, "r") # "r" means we will read from inputfilename**

**output = open(outputfilename, "w") # "w" means we will write to outputfilename**

**## You can read the lines one by one with a while loop:**

**#**

**# line = input.readline() # read the next line of input --- this includes**

**# # the ending \r and \n characters!**

**# # The line is of course a string.**

**#**

**# while line != "" : # When the input is all used up, line == ""**

**# print line**

**# output.write(line) # Write the string to the output file**

**# line = input.readline()**

**# But here is the simplest way to read all the lines one by one:**

**for line in input :**

**print line**

**output.write(line)**

**# when finished, you must close the files:**

**input.close()**

**output.close()**

# Classes and objects

**A Python class is a ``mini-module'' that can be ``imported'' (*constructed*) multiple times.**

**Each time the class is ``imported,'' a new, permanent namespace is constructed. Each namespace is called an *object*.**

A class looks like a module --- it has code for building a data structure and the functions for maintaining the data structure. The code for building the data structure must be inserted inside a special function, a *constructor* function, named \_\_init\_\_. There are additional functions, such as the event handling function, handle. The class has this form:

============================

**class NAME ( BASE\_CLASS ) :**

**def \_\_init\_\_(self, ... ) :**

**"""constructs the object (namespace) in heap storage. The**

**address of the new object is assigned to the parameter, self.**

**Parameter self is used to define and save the object's variables**

**in its namespace.**

**"""**

**. . .**

**self.mydata = ... # make a variable, mydata, in the object**

**. . .**

**# the function automatically returns the value of self as its answer.**

**def handle(self, ... ) :**

**"""event handling function for the object whose address is self"""**

**. . . self.mydata . . .**

====================================

When the class is used to construct an object, we write this:

**x = NAME( ... )**

The occurrence of NAME on the right-hand side of the assignment is a disguised call of the \_\_init\_\_ function. Indeed, the Python interpreter reformats the above assignment into a ``module'' call in dot notation:

**x = NAME.\_\_init\_\_( getNewAddressInHeapForTheNewObject, ... )**

Notice that an extra argument is supplied for free --- the address in the heap where the new object will be constructed.

Once the object is constructed, the new address is assigned to variable x. A method within the object is called like this:

**x.handle(...)**

Again, the Python intepreter reformats this call, to look like this:

**C.handle(x, ...)**

and now it is clear that the address held by variable x is assigned to parameter self in handle, correctly connecting the object to its event handling function.