Python Dictionary

# *Learning Python the Hard Way*, Exercise 22

## Basics

*python* From the command line, this is the command to start the Python scripting language/program.

*pydoc [function]* From the command line, the command that opens the Python documentation (similar to the UNIX *man* command) for the named *function*.

*Control-D (^D)* Exit Python

*Control-Z (^Z)* Exit Python

*Control-C (^C)* Interrupt the current script.

*Variable* Used like variables in algebra. Names are assigned by the coder. “Programmers use these variable names to make their code read more like English, and because they have lousy memories. If they didn’t use good names for things in their software, they’d get lost when they tried to read their code again.”  
Parts of (words within) the variable are separated with an **underscore (\_)**.  
***DmgCntrl standard:*** Use easy-to-understand variable names, such as **user\_name** rather than “only-I-understand-them” names, such as **uname**.

*Floating point* By default, Python performs integer math. Be sure to specify when floating-point numbers are needed. *For example*, by default 14 / 5 = 2, whereas 14.0 / 5 = 2.8 and 14 / 5.0 = 2.8.

*Nested* Conditional statements placed “within” or “below” one another, so that new *branches* of code are created, based upon the conditions above being met. Allows multiple “levels” of statements/conditions to affect the “direction” that the code flows.

## Commands

*print* Output to screen the string(s) that follow(s)

*str(variable) variable* is/becomes a text string

*float(variable) variable* is/becomes a floating-point number

*int(variable)* *variable* is/becomes an integer

*raw\_input(“prompt”)* Output *“prompt”* to the screen and receive the user’s response from the keyboard.

*[from module\_name] import* Import functionality (especially functions and/or commands and/or operators) from *module\_name*. Examples:

* **from sys import argv** (Import the arguments from the command line. The script name is the first argument after the *python* command and is also the first argument received with this command.) After this command, “unpack” the command-line arguments with **script\_name, *second\_arg, [third\_arg, fourth\_arg, …] = argv***
* **from os.path import exists** [That allows a later call **exists(file\_name)** to test if the file exists. Return is TRUE or FALSE.]
* More available from **pydoc import** or [here](http://docs.python.org/2/reference/simple_stmts.html#the-import-statement).

*open(file\_name)* Open the *file\_name* file

*file\_object\_variable.read()* Read the contents of the file object stored in *file\_object\_variable*

*file\_object\_variable.close()* Close the file object stored in *file\_object\_variable*

*file\_obj\_var.readline()* Read one line of text file *file\_obj\_var*

*file\_obj.truncate()* ***\*\*DANGER\*\**** EMPTIES THE FILE!!

*file\_obj.write(stuff)* Writes string *stuff* to the file.

*File\_obj.seek(offset[, whence])* Set the file’s current position. The *whence* argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file’s end). There is no return value.

*def fn\_name([arg1,arg2,…]):* Define new function *fn\_name* that takes [optional] arguments *arg1, arg2, …* The *def* line must end with a colon **:** . The lines that the function “does” must be indented by four spaces (no more, no less).  
Functions do four things

1. They name pieces of code the way that variables name strings and numbers.
2. They take arguments the way scripts take **argv**.
3. Using #1 and #1, they let you make your own “mini scripts” or “tiny commands.”
4. They can **return** a result when they complete.

See below for the **Function Checklist**

*if [elif, else]* Tests a condition for truthiness. *then* is not explicit; indented lines of code beneath the **if/elif** conditional statement are run as the *then* of the condition. e*lif* is the same as “else-if.” *else* is the code to execute when the conditional(s) above is/are not True.

*for*

*range([start,] stop[, step])* Returns a list containing an arithmetic progression of integers. **range(i, j)** returns [i, i+1, i+2, …, j-1]. s*tart* defaults to zero. When *step* is given, it specifies the increment/decrement.

*while*

## Operators (aka Symbols)

“ (Double Quote) Text string delimiter (It’s OK to use Single Quotes between Double Quotes; they’ll be treated as text.)

‘ (Single Quote) Text string delimiter (It’s OK to use Double Quotes between Single Quotes; they’ll be treated as text.)

“”” Triple-double-quote allows establishing a string with many lines and other elements inside until ended with another trip-double-quote.

‘’’ Triple-single-quote works the same as triple-double-quote, except that double-quotes can be used within the string.

# (Octothorpe) Anything after the character will be ignored by Python. The text after the octothorpe is called a “Comment.” (Symbol also known as “pound”, “hash”, “mesh”, etc.)

, Comma: Standard list separator. Also tells the **print** command not to end the output with a *newline*.

+ Plus (addition) - also performs concatenation on text strings

- Minus (subtraction)

/ Slash (division)

\* Asterisk *or* Star (multiplication) - also indicates ‘repeat this string *n* times’ when used this way: **“.” \* 10** would output ……….

% Percent

* In math/formulae, this represents the *remainder* operation (*e.g.* 10 % 5 = 0, 10 % 4 = 2, 10 % 3 = 1)
* In the midst of strings in **print** commands, this indicates a *placeholder* for a string insertion. The insertion is named after the string is closed, another percent placed, then a string variable named. (*e.g.* **print** “This is a string with a **%s** placeholder.” **% string\_variable**.) The character after the percent indicates what type of string is expected in that place. A list of such string formatting characters is available [here](http://docs.python.org/release/2.5.2/lib/typesseq-strings.html) or [here](http://docs.python.org/2/library/string.html#format-specification-mini-language).

= Equals

< Strictly Less-than

> Strictly Greater-than

<= Less-than-equal (Less than or equal to)

>= Greater-than-equal (Greater than or equal to)

+= Plus-equals: Add the value after the operator to the variable named in front of the operator. Explanation:  
**variable\_a += 5** is the same as  
**variable\_a = variable\_a + 5**

( Open parenthesis

) Close parenthesis

\ Back-slash indicates an “escape sequence” placed in a string. A list of valid escape sequences can be found [here](http://docs.python.org/2/reference/lexical_analysis.html#string-literals).

: Colon: Ends the **def** line of a new function

and Logical operator which functions like the English conjunction.

or Logical operator which functions like the English conjunction.

not Logical operator which functions like the English word.

!= Not equal (Can also be written <> , but this is obsolete usage. New code should always use != .)

== “Double” Equal. Different than single =, which sets a variable to a value/string, == tests a condition/logical statement.

True Logical state of TRUE

False Logical state of FALSE

is Object identity

is not Negated object identity

[ “Open bracket” or “left-bracket”: “Opens” a list.

] “Close bracket” or “right-bracket”: “Closes” a list.

## Function Checklists

*Creating a Function*

1. Did you start your function definition with **def**?
2. Does your function name have only characters and \_ (underscore) characters?
3. Did you put an open parenthesis ( right after the function name (no spaces)?
4. Did you put your argument(s) [if any] after the open parenthesis ( , separated by commas?
5. Did you make each argument unique (*i.e.,* no duplicate names)?
6. Did you put a close parenthesis and a colon ): after the argument(s)?
7. Did you indent all lines of code that you want in the function 4 spaces? No more, no less.
8. Did you use **return** to send “output” or a result from the function back to the part of the script that called it?
9. Did you “end” your function by going back to writing with no indent (aka *dedenting*)?

*Calling (Running/Using) a Function*

1. Did you call (run/use) this function by typing its name?
2. Did you put ( character after the name to run it?
3. Did you put the values for the function’s argument(s) [if any] inside the parentheses, separated by commas?
4. Did you end the function call with a ) character?
5. How are you handling any **return** from the function?

The Truth Tables

|  |  |
| --- | --- |
| **NOT** | **True?** |
| not False | True |
| not True | False |

|  |  |
| --- | --- |
| **OR** | **True?** |
| True or False | True |
| True or True | True |
| False or True | True |
| False or False | False |

|  |  |
| --- | --- |
| **AND** | **True?** |
| True and False | False |
| True and True | True |
| False and True | False |
| False and False | False |

|  |  |
| --- | --- |
| **NOT OR** | **True?** |
| not (True or False) | False |
| not (True or True) | False |
| not (False or True) | False |
| Not (False or False) | True |

|  |  |
| --- | --- |
| **NOT AND** | **True?** |
| not (True and False) | True |
| not (True and True) | False |
| not (False and True) | True |
| not (False and False) | True |

|  |  |
| --- | --- |
| **!=** | **True?** |
| 1 != 0 | True |
| 1 != 1 | False |
| 0 != 1 | True |
| 0 != 0 | False |

|  |  |
| --- | --- |
| **==** | **True?** |
| 1 == 0 | False |
| 1 == 1 | True |
| 0 == 1 | False |
| 0 == 0 | True |