Python Notes, Tips & Tricks

## 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**.

*Function* An executable statement/block-of-code that can be “called” at another point in the program. See below at **def** and more information [here](http://docs.python.org/2/reference/compound_stmts.html#function-definitions).

*Class* An executable statement/block-of-code. One can think of a class as a “dictionary with functions.” Full information is available [here](http://docs.python.org/2/reference/compound_stmts.html#class-definitions). A class inherits from the (built-in) class named *object* the “qualities” of being a class. This is the reason that the syntax for creating a class must be **class *ClassName(object):***

*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.

*Ordinal numbers* The “counting” or “ordering” numbers (first, second, third, …). They are represented as 1, 2, 3, …

*Cardinal numbers* Numbers that represent the absolute position of an element in a set (this is also known as its *index*). For programming purposes, the first index/position is 0 (zero), the second index/position is 1, the third is 2, … Cardinal numbers are also used to indicate the index/position of a data element in a file.

*Dictionaries* It is best to think of a dictionary as an unordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary:{}. Placing a comma-separated list of *key: value* pairs within the braces adds initial pairs to the dictionary; this is also the way dictionaries are written on output. A full discussion is [here](http://docs.python.org/2/tutorial/datastructures.html#dictionaries).

## 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.

*with…as* A good review of the function’s use is [here](http://effbot.org/zone/python-with-statement.htm). A practical example:  
 **with open(“x.txt”) as f:  
 data = f.read()  
 do something with data**

*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. Further information can be found [here](http://docs.python.org/2/reference/simple_stmts.html#the-return-statement).

See below for the **Function Checklist**

*lambda* A “shorthand” method to create anonymous functions. The expression **lambda arguments: expression** yields a function object. More information available [here](http://docs.python.org/2/reference/expressions.html#lambda).

*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* *For* loops are used to repeat a block of code a certain number of times.

*while* Test the statement that follows, like an **if**-statement, but instead of running the block of code once, jump back to the “top” (where the **while** is) and repeat as long as the condition is True. When the condition is False, the **while**-loop “fails” (*i.e.*, Python stops running that block of code).  
\*\*Caution\*\* Sometimes they do not stop. This is only good if you want to keep looping until the end of the universe.  
*Simple Rules of* ***while****-loops*  
1. Make sure that you use **while**-loops sparingly. Usually a **for**-loop is better.  
2. Review your **while** statements and make sure that the thing you are testing will become False at some point.  
3. When in doubt, print out your test variable at the top and bottom of the **while**-loop to see what it’s doing.

*continue* may only occur syntactically nested in a **for** or **while** loop, but not nested in a function of class definition or **finally** clause within that loop. It continues with the next cycle of the nearest enclosing loop. When **continue** passes control out of a **try** statement with a **finally** clause, that **finally** clause is executed before really starting the next loop cycle.

*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.

*del* When working with lists, **del a[*n*]** deletes the item at index *n*. This differs from the *pop()* method which returns a value.  
  
**del *var*** deletes the entire variable *var*.

*assert* Insert debugging assertions (statements) into programs. Fuller definition and discussion [here](http://docs.python.org/2/reference/simple_stmts.html#the-assert-statement).

*pass* A null operation (when it is executed, nothing happens). Useful as a placeholder when a statement is required syntactically, but no code needs to be executed (*e.g.,* function or class that doesn’t do anything yet).

*yield* Causes the state of a generator function to be frozen, and the value of the expression list is returned to **next()**’s caller. All local state (current bindings of local variables, the instruction pointer, and the internal evaluation stack) is retained. Enough information is retained so that the next time **next()** is invoked, the function can proceed as if the **yield** were just another external call. Further information available [here](http://docs.python.org/2/reference/simple_stmts.html#the-yield-statement).

*break* May only occur syntactically nested in a **for** or **while** loop, but not nested in a function or class definition within that loop. It terminates the nearest enclosing loop, skipping the optional **else** clause if the loop has one. If a **for** loop terminated by **break**, the loop control keeps its current value. When **break** passes control out of a **try** statement with a **finally** clause, that **finally** clause is executed before really leaving the loop.

*try…(except…)][else…][finally…]* The **try** statement specifies exception handlers and/or cleanup code for a group of statements. A full explanation/discussion is [here](http://docs.python.org/2/reference/compound_stmts.html#the-try-statement). The **except** clause(s) specify one or more exception handlers. The optional **else** clause is executed if and when control flows off the end of the **try** clause. If **finally** is present, it specifies a “cleanup” handler.

*exec* Allows dynamic execution of Python code contained in a string, an open file object, or a code object. Full information available [here](http://docs.python.org/release/2.5.2/ref/exec.html).

*raise [expression[“,” expression [“,” expression]]]* Evaluates the expressions to get three objects, using **None** as the value of omitted expressions. The first two objects are used to determine the *type* and *value* of the exception. Fuller discussion [here](http://docs.python.org/2/reference/simple_stmts.html#the-raise-statement).

## 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*.

: Colon

* ends the **def** line of a new function.
* separates sections of a [compound statement](http://docs.python.org/2/reference/compound_stmts.html#compound-statements):

**if test1 : if test2 : print x**

* separates the *key* from its corresponding *value* in a dictionary

; separates sections of a [compound statement](http://docs.python.org/2/reference/compound_stmts.html#compound-statements). The semi-colon binds more tightly than the colon; in the following example, either all or none of the **print** statements is executed:

**if x < y < z: print x; print y; print z**

. Period or dot

* same function as in all mathematics (“decimal point”)
* separates an object from the command/function to be performed on it. [(Examples can be found here)](http://python4astronomers.github.io/python/objects.html)

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

- Minus (subtraction)

/ Slash (division)

// Double-slash (floor division) - divide and round down to the nearest integer

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

\*\* Exponentiation ( 2 \*\* -1 = 0.5 )

% Percent

* In math/formulae, this represents the *remainder* (or modulo) 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**

-= Minus-equals: Same as the above, but for subtraction

\*= Star-equals or asterisk-equals or multiply-equals: Same as the above, but for multiplication

/= Slash-equals: Same as the above, but for division

//= Slash-slash-equals or Double-slash-equals: Same as the above, but for floor-division

%= Percent-equals or modulo-equals: Same as the above, but for modulo (remainder) function

\*\*= Star-star-equals or Double-star-equals: Same as the above, but for exponentiation

( 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).

@ Indicates/introduces a “decorator pattern” for a function. See the full discussion [here](http://wiki.python.org/moin/PythonDecorators).

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.

{ “Open [curly] brace” or “left [curly] brace”: Opens a set or dictionary.

} “Close [curly] brace” or “right [curly] brace”: Closes a set or dictionary.

None A logical return when the result of an argument, expression, or function is a null set.

Style Manual

# Function Style

* Programmers call *functions* that are part of classes *modules*.
* When you work with classes, much of your time is spent talking about making the class “do things.” Instead of naming your functions after what the function does, name it as if it’s a command you are giving to the class. Same as **pop** is saying, “Hey, list **pop** this off.” It isn’t called **remove\_from\_end\_of\_list** because even though that’s what it does, that’s not a *command* to a list.
* Keep your functions small and simple. For some reason when people start learning classes, they forget this.

# Class Style

* Your class should use “camel case,” like **SuperGoldFactory** rather than **super\_gold\_factory**.
* Try not to do too much in your **\_\_init\_\_** functions. It makes them harder to use.
* Your other functions should use “underscore format,” so use **my\_awesome\_hair** and not **myawesomehair** or **MyAwesomeHair**.
* Be consistent in how you organize your function arguments. If your class has to deal with users, dogs, and cats, keep that order throughout, unless it really doesn’t make sense. If you have one function that takes **(dog, cat, user)**, and the other takes **(user, cat, dog)**, it’ll be hard to use.
* Try not to use variables that come from the module of globals. They should be fairly self-contained.
* A foolish consistency is the hobgoblin of little minds. Consistency is good, but foolishly following some idiotic mantra “because everyone else does” is bad style. Think for yourself.
* Always, ***always*** have **class CamelStyleName(object)** format, or else you will be in big trouble.

# Code Style

* There are programmers who will tell you that your code should be readable enough that you do not need comments. They’ll then tell you in their most official-sounding voice that, “Ergo you should never write comments.” Those programmers are either consultants who get paid more if other people can’t use their code, or incompetents who tend never to work with other people. Ignore them and write comments.
* When you write comments, describe *why* you are doing what you are doing. The code already says how, but why you did things the way you did is more important.
* When you write doc comments for your functions, make the comments documentation for someone who will have to use your code. You do not have to go crazy, but a nice little sentence about what someone does with that function helps a lot.
* Finally, while comments are good, too many are bad, and you have to maintain them. Keep your comments relatively short and to the point, and if you change a function, review the comment to make sure it’s still correct.

Using the Skeleton

# Currently located in Jays-MacBook-Pro:~/PythonTheHardWay/projects

1. Make a copy of your skeleton directory. Name it after your new project.
2. Rename (move) the NAME module to be the name of your project or whatever you want to call your root module.
3. Edit your setup.py to have all the information for your project.
4. Rename **tests/NAME\_tests.py** to also have your module name.
5. Double-check it’s all working using **nosetests**.
6. Start coding.

## 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?

## Rules for *If*-Statements

1. Every **if**-statement must have an **else**.
2. If this **else** should never be run because it doesn’t make sense, then you must use a **die** function in the **else** that prints out an error message and dies.
3. Never nest **if**-statements more than 2 deep, and always try to do them 1 deep. This means if you put an **if** in an **if** then you should be looking to move that second **if** into another function.
4. Treat **if**-statements like paragraphs, where each **if**, **elif**, **else** grouping is like a set of sentences. Put blank lines before and after.
5. Your Boolean tests should be simple. If they are complex, move their calculations to variables earlier in your function and use a good name for the variable.

## Rules for Loops

1. Use a **while**-loop only to loop forever, and that probably means never. This applies only to Python; other languages are different.
2. Use a **for**-loop for all other kinds of looping, especially if there is a fixed or limited number of things to loop over.

## Tips for Debugging

1. Do not use a “debugger.” A debugger is like doing a full-body scan on a sick person. You do not get any specific useful information, and you find a whole lot of information that doesn’t help and is just confusing.
2. The best way to debug a program is to use **print** to print out the values of variables at points in the program to see where they are going.
3. Make sure parts of your programs work as you work on them. Do not write massive files of code before you try to run them. Code a little, run a little, fix a little. (Well, fix all the problems that you find, but there shouldn’t be too many of them.)

## Testing Guidelines

1. Tests go in **tests/** and are named **PROJECT\_tests.py**; otherwise *nosetests* won’t run them. This also keeps your tests from clashing with your other code.
2. Write one test file for each module you make.
3. Keep your tests cases (functions) short, but do not worry if they are a bit messy. Test cases are usually kind of messy.
4. Even though test cases are messy, try to keep them clean and remove any repetitive code you can. Create helper functions to get rid of duplicate code. Duplicated code makes changing the tests more difficult.
5. Finally, do not get too attached to your tests. Sometimes the best way to redesign something is to just delete it, the tests, and start over.

A Message for the Procrastinator

Yes, this means YOU!

Every programmer becomes paralyzed by irrational fear when starting a new, large project. They then use procrastination to avoid confronting this fear, and they end up not getting their program working or even started. We all do this. **Everyone** does this.

The best way to avoid this is to make a list of things you should do, and then do one at a time.

Just start doing it. Do a small version. Make it bigger. Keep updating the list of things to do, and do them.

Another help might be a “map” or flowchart of the functionality of the project. Map out all the modules, decision points, functions, outputs, *etc.*, that you need. Once you have the flowchart, start coding. If you find problems with the flowchart, then adjust it, and make the code match. Keep the flowchart up-to-date with new things that you discover that you need to add.

You’ll be amazed how helpful the flowchart will be in keeping the logical flow of the project clear in your mind.

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 |