413-513 Notes- Week 1 COMMON PRACTICES **Chapters 1 & 2**

-**Concentration**-

(Week 1-2)

On syntax (operators, variables, assignments), IO & simple functions –

(Weeks 2-14)

Algorithm development

**[ Coding layout logic ]**

Tabs or spaces?

Spaces are the preferred indentation method.

Coding layout preferred: Use 4 spaces per indentation level.

Tabs should be used solely to remain consistent with code that is already indented with tabs.

**Python 3 disallows mixing the “use” of tabs and spaces for indentation.**

Python 2 code *indented* with a mixture of tabs and spaces should be converted to using spaces exclusively

…Note errors on indentation

**str = "Python Superstars!!";**

**print (str.rjust(10));**

**-**

**def func1():**

**"func docstring"**

**return "hi"**

**x = func1();**

**print(x)**

Maximum Line Length

**\*Limit all lines to a maximum of 79 characters\***

For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters.

[The Python standard library is conservative and requires limiting lines to 79 characters (and docstrings/comments to 72)]

-Comments (good for documentation/debugging)

'''

This is a multiline

comment.

'''

Use ( **'''** ) and ( **"""** ) to *comment out* the block.

**IDLE** IDE has options like **Comment out region** and **Uncomment regions** (shortcut: Alt+3 for Win (ctrl + 3 mac) and Alt+4 respectively) under **Format** menu. Now it is more easier than ever..

Documentation Strings - **P**ython **E**nhancement **P**roposals (PEP)

Conventions for writing good documentation strings (a.k.a. "docstrings") are immortalized in [PEP 257](http://legacy.python.org/dev/peps/pep-0257).

* Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the def line.

For one liner docstrings, please keep the closing **"""** on the same line.

**[ Python Basic** [**Arithmetic**](https://www.tutorialspoint.com/python3/python_basic_operators.htm) **Operators ]**

Assume below variable **a** holds the value 10 and variable **b** holds the value 20.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + Addition | Adds values on either side of the operator. | a + b = 31 |
| - Subtraction | Subtracts right hand operand from left hand operand. | a – b = -11 |
| \* Multiplication | Multiplies values on either side of the operator | a \* b = 210 |
| / Division | Divides left hand operand by right hand operand | b / a = 2.1 |
| % Modulus | Divides left hand operand by right hand operand and returns remainder | b % a = 1 |
| \*\* Exponent | Performs exponential (power) calculation on operators | a\*\*b =10 to the power of 20 |
| // | Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity): | 9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0 |

**[ Multiple assignment statements ]**

The basic rule is that Python evaluates the entire right-hand side of the **=** statement. Then it **matches** values with destinations on the left-hand side. If the lists are different lengths, an exception is raised and the program stops.

Because of the complete evaluation of the right-hand side, the following construct works nicely to swap to variables.

**#ex. swap vars**

**a,b = 1,4**

**b,a = a,b**

**print (a,b)**

Example**line.py**

**#!/usr/bin/env python**

**# Compute line between two points.**

**x1,y1 = 2,3 # point one**

**x2,y2 = 6,8 # point two**

**m,b = float(y1-y2)/(x1-x2), y1-float(y1-y2)/(x1-x2)\*x1**

**print ("y=",m,"\*x +",b)**

**./line.py**

y = 1.25 \*x + 0.5

Assignment chaining allowed

**num1 = num2 = 1;**

**[ Variables ]**

Keyword avoidance!

Declarations (LOCAL VS. GLOBAL VS. MODULE)

Conventions

**Note – no constants in Python!!!**

What to do then…help!!!

**-Use of strings**

- methods (Study chapter / slides)

- indexing thru

<http://www.tutorialspoint.com/python/python_strings.htm>

Multi-Line Statements

var1 = 'Hello World!'

var2 = "Python Programming"

print ("var1[0]: ", var1[0])

print ("var2[1:5]: ", var2[1:5])

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue.

For example −

total = item\_one + \

item\_two + \

item\_three

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example −

days = ['Monday', 'Tuesday', 'Wednesday',

'Thursday', 'Friday']

Quotations in Python

Python accepts single ('), double (") and triple (''' or """) quotes to denote string ‘literals’, as long as the same type of quote starts and ends the string. Triple quotes are used to span the string across multiple lines. For example, all the following are legal −

word = 'word'

sentence = "This is a sentence."

paragraph = """This is a paragraph. It is

made up of multiple lines and sentences."""

…or use concat operator +

**refs:**

Py software foundation- Python Enhancement Proposals (PEPs)

<https://www.python.org/dev/peps/pep-3101/>

<http://www.tutorialspoint.com/python/string_ljust.htm>

**[ Operators ]**

- Shortcut assignments avoid pitfall!

**>>> c = 2**

**>>> a = 1**

**>>> b = 2**

**>>> c\*=a+b #s/b 4???**

**>>> print (c)**

**6**

**>>>**

-Bitwise bit shifts

**Onto the Demos!!!!** (Check ‘**Scripts**’ link in BB)

0. IDLE sys config

1. keywords

**import keyword**

**print(keyword.kwlist)**

2. adding date/time

3. conversion funcs from input()

4. help()

5. print statement formatting

currency style/rounding (check **intellisense** on functions)

<https://docs.python.org/3/library/string.html>

Format String Syntax

\t\n --easiest!

currecy style

**value = 12233.335**

**print ("${:,}".format(round(value,2)))**

Format strings contain “replacement fields” surrounded by curly braces {}. Anything that is not contained in braces is considered ‘literal’ text, which is copied unchanged to the output - If you need to include a brace character in the literal text, it can be escaped by doubling: {{ and }}.

**PADDING!**

Use of negation!

print ('%-10s is todays date' % str); #includes padding of string if nec.

justifications/paddings

string.**ljust**(*s*, *width*[, *fillchar*])

string.**rjust**(*s*, *width*[, *fillchar*])

ex.

The method **ljust()** returns the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is a space). The original string is returned if width is less than len(s).

Syntax

Following is the syntax for **ljust()** method −

str.ljust(width[, fillchar])

Parameters

* **width** -- This is string length in total after padding.
* **fillchar** -- This is filler character, default is a space.

**str = "this is string example....wow!!!";**

**print (str.ljust(50, '0'));**

**this is string example....wow!!!000000000000000000**

format specifiers (d vs. f), aka utilizing specifiers within print method

**s1 = "cats"; s2 = "dogs"**

**s3 = " %s and %s living together" % (s1, s2)**

**NOT**

**s3 = " %s and %s living together" , (s1, s2)**

**TAKES AS LITERAL INFO!**

**print (s3)**

end,sep args

**end – helps to avoid newline**

**sep help to avoid spacing between display values**

**import time, keyword**

**print("hello",end =' ')**

**print("hey"," man", "cow", sep=" ")**

**print (keyword.kwlist)**

Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

Imports should be grouped in the following order:

* standard library imports
* related third party imports
* local application/library specific imports

You should put a blank line between each group of imports.