PRE-LEARNING FOR PYTHON (PFP)

drboboland@hotmail.com and Team

Version 10 draft - September 2013

30 MINUTES

CONTENTS

1. Introduction

Python provides: input, processing and output. This PFP is designed to be 30 minutes of simple pre-learning of the Python basic vocabulary, grammar and procedures, for people with no previous computer or language programming experience.

Don't try to understand all PFP. Just absorb what you can and become more familiar with Python, before starting your Python training program.

PFP uses ideas from four excellent resources: Python Tutorial, Instant Python (Hetland), Tutorial Python in 10 Minutes (Stavros), and udacity lesson notes (David Owen), all freely available on Google.

Install and play with Python from Google.com search: python idle download, and then choose 27.5. The Python Package Index, provides extensive help with: language guide, tutorials, syntax (grammar), help etc.

TEST OF PFP

NOW PLEASE READ **EXHIBIT A** FOR 10 MINUTES …

… THEN COMPLETE THE 30 MINUTES OF PFP…

… THEN READ **EXHIBIT A** AGAIN …

… AND THEN BRIEFLY EMAIL US YOUR IDEAS AND REACTIONS.

2. Vocabulary (Syntax)

The Python syntax (language grammar) uses both normal and exotic language. # – indicates a non-program explanation
Need to gradually become familiar with the following:

Python Word	English Meaning
<Name>	file name
<Input>	here is new data
<Output>	here is the output (result)
String	""words_with_spaces"" with quotes
s	string_of_ words
Num	numbers
Mutable	can be changed
Immutable	cannot be changed, only replaced
Concatenation	combination
Parameter	variable element in numbers or text
Block	action taken
Item	data in numbers or text
Slicing	selecting with an index (:)
Colon:	indent the next instruction
Indent	insert big space from left margin
Def	define a new relationship
Expression	new or revised data
Return	take action now
Block	take action to get result
[ ] brackets	part of a process
del	delete variables
Index	(:)
x = 5	x assigned to new value 5
==	exactly equal
!=	not equal
- +, -, *, /	normal arithmetic
1 += 2	add 1 to 2
1 < a < 3	less than 3 and greater than 1.

<u>Example</u>: # assignments with spaces

```
x, y,z = 1, 2, 3                      # assign variables to x, y & z
first,  second  =  second,  first    # make a change
a  = 450                             # assign value to a
```

<u>Example</u>:   # Concatenate (combine) two strings.

```
>>> mystring = "Hello"       # File mystring with word input assigned
>>> mystring += "   world." # Combine with  added string to make a
                                combination, with a space
                                provided before -  "     world""
>>> print mystring           # instruction to print out the new string
Hello world.                 # printing
```

<u>Example</u>:   # get user interaction

```
X =   input (""Please enter number:    "") # request for a  number
Print ("The number is,"" x * x                 # printing instruction
```

3. Inputs and Outputs

3.1 Python provides: input, processing and output.
Data may be: float (decimal number), or integer (number) or string (letters with spaces).

Assign with code =, and confirm equality with code ==-

Example:

```
Variable assigned by the code =       x = 6
Equality confirmed by the code ==   y == b (called Bodean equality)
```

3.2 The instructions for Print and Block are indented after a colon:

Example: # print after a testing argument

```
if x > 10 and x < 20):                # If x is greater than 10 but less than 20
    print "Value is OK"               # indent print instruction  after the colon
    Value is OK                       # printing
```

3.3 The loop variable, iterates (works) through the elements of a list using the function range( ).

Example: # print out a range

```
for value in range(100):             # set the values from 0 to 99
                                   inclusive. Note the colon.
    print value                       # print instruction for values to  1 … 99
    1,2,3 …99                         # printing
```

4. Indexing & Slicing

4.1 Python provides: input, processing and output. Use indexing to get access into specific elements of a list either separately or in groups.

Indexing can append (add) items to a list. Pot can remove them. The first list item ALWAYS has the index of code (0) not (1).

<u>Example</u>:   # indexing & printing

```
name = ["Smith", "John"]:    # assign names to list [0 & 1] brackets.
      print name[1], name[0]  # print name 1 & name 0 (¨¨Smith¨¨)
     John Smith                # printing
```

4.2 Slicing (selecting) is like indexing, with start and stop instructions, indices, brackets etc.

```
X ="spam0","spam1","eggs2”,"and3","spam4"]
                                           # Names the 5 elements
                                           # Listed with ¨¨  ¨¨ coded 0 to 4
      print x[2:4]                  # prints code 2 to 3 ONLY not 4
                                    # Prints from the list from 5 to 6 (excluding 7)
       eggs and                     # printing code 2 and 3
```

The end code 4 is not-included. But, if an end index is omitted (1:), then print everything up to and including the last item.

Example for a list of 8 items (0 to 7):

List [:] coded as 0 to 7

List [:8] coded as 0 to 7

List (1:3) coded as 2 to 4.

List [3:] coded as 4 to 7

5. Functions

5.1 Python provides: input, processing and output. A function is a procedure using variable inputs to give an output. Each function and variable (parameter) has a name. A function can be defined with the code. A local function applies to only part of a program, a global function applies to all.

Example: # define a function

```
def square(x):          # define function square and parameter x  with :
     return x*x         # start procedure – square of x
     print square(2)    # print out the square of x = 2  (4)
     4                  # printing
```

5.2 Bind (pass) a parameter to a function, with a new reference (keeping original on file if needed).

Example: # working with the file somelist

```
def change(some_list):          # define the change in a specified
                                   list. Colon with indent.
     some_list[1] = 4           # possible change item 1 to 4 but
                                   not yet activated with return.
x = [1,2,3]                     # identify the list items 1, 2, 3
                                # (coded as 0, 1, 2)
change(x)                       # change item 1 (the second item after 0)
                                to 4) now activated
print x                         # print instruction  [1,4,3]
1,4,3                           # printing
```

5.3 Functions are values in Python. A function like square, can be used directly.

Example: # define queeble as function

```
queeble = square                # queeble now a square function
print queeble(2)                # print instruction qeeble
                                  number 2 to be squared
4                               # printing
```

6. Objects

6.1 Python provides: input, processing and output. Define objects in classes with a keyword class. Use the common initiation instruction __init__ to help definition.

```
Example:                                # set up a class

class Basket:                           # name of class
def __init__(self,contents=None):       #  use init (initiation) of file self, contents
                                            class, with
                                        value none
      self.contents = contents or [ ]   #  set values of contents
                                        idebtified with file self.
```

6.2 The *self* argument is used extensively.

```
Example:                                #  set up self file

def add(self,element):                  # define self elements colon
      self.contents.append(element)     # append (add) contents file
                                               with element

def print_me(self):                     # order print of self file
      result = " "                      # specify results
      for element in self.contents:     # specify content elements
           result = result + " " + `element   # specify results, text and
                                          the element
      print "Contains:"+result          # order printing with text
                                          and results
```

7. Methods

7.1 Python provides: input, processing and output All methods (functions) receive an additional argument at the start of the argument list, often called self (usual Python term)

Methods may be defined with the code - __init__ (two underscores) and a name, to mean constructor of a class i.e. a function that creates an instance class.

```
Example:                                  #  init method used
object.method(arg1,arg2)                  # sets the arguments
```

7.2 Some arguments can be optional and given a default value.

```
Example:                                  # display default value

def spam(age=32): ...                     # define spam with default value
                                          32 if no other value given
```

7.3 In short-circuit logic, expressions may be evaluated in two ways: a and b, or alternatively a or b.

Example:

a and b
First, check if a is true.
If it is not true, then simply return it. a.
If it is true, then return b (which will represent the true value of the expression.)

a or b
First, check if a is true.
If a is true, then return it. a.
If it is not true, then return b.

8. Baskets

8.1 Python provides: input, processing and output. Make a Basket of new data to be processed.

```
Example:                                # set up basket

  b = Basket(['apple','orange'])        # defines basket contents
  b.add("lemon")                        # add new element lemon
  b.print_me( )                         # print result
```

```
Example:                                # set up basket

 def __str__(self):                     # defines a string and self
      self = sels(contents)             # sets up contents
      result = " "                      # action taken
      for element in self.contents:     # locate self file
      result = result + " " + `element  # defines contents
      return "Contains:"+result         # action taken
      print                             # print contents
```

8.2 In basket use the standard __init__. The argument content has a default value of None. Need to check if it has a value.

```
Example:                                # check out a self file

if contents:                            # locate file
     self.contents = contents           # check values,  if none go to else
     else:                              # alternative orders
         self.contents = [ ]            # sets values
```

```
Example:                                # check out a self file

def __init__(self, contents=[ ]):       # define location
        self.contents = contents[:]     # provide for input
```

9. Other

9.1 Useful functions and classes are put into modules, which are text-files in available as an import for all programs.

For a program to be both an importable module and runnable program, add this at the end of it: if __name__ == "__main__": go()

9.2 Exceptions (like dividing something by zero) produce an error condition or exception and your program ends and prints out an error message. Avoid this with a try/except-statement.

Example: # avoid an exception by testing

```
def safe_division(a,b):          # define procedure
    try:                                 # test
        return a/b                # check result
    except ZeroDivisionError:    # if error - standard error notice
        return None                  # result of error - none
```

10. Conclusions

This simple pre-learning of basic Python vocabulary, grammar and procedures, is designed to encourage people with no previous computer programming experience.

Resolve misunderstandings with an experienced programmer and be motivated to complete basic Python training.

A good summary of Udacity Python learning is freely available on https://www.udacity.com/wiki/cs101 with the four excellent sources of this PFP.

TEST OF PFP - AFTER COMPLETING 30 MINUTES OF PFP, PLEASE READ AGAIN EXHIBIT A FOR 10 MINUTES … AND SEND US YOUR REACTIONS.

Sources of PFP (all free on Google):

Udacity Computer Science (David Owen)
  https://www.udacity.com/
Python in 10 Minutes (Stavros)
  www.amazon.it/Learn-Python-Ten-Minutes-ebook/dp/B0052T2V06
  (14 pages)
Python Web File
  http://www.01-telecharger.com/python
Instant Python by Magnus Lie Hetland
  magnus@hetland.org (9 pages)

EXHIBIT A - PYTHON GUIDE

(Introduction to Computer Science (Udacity - David Owen))

Contents

Arithmetic Expressions

addition: <Number> + <Number> = <Number>

- outputs the sum of the two input numbers

multiplication: <Number> * <Number> = <Number>

- outputs the product of the two input numbers

subtraction: <Number> - <Number> = <Number>

- outputs the difference between the two input numbers

division: <Number> / <Number> = <Number>

- outputs the result of dividing the first number by the second Note: if both numbers are whole numbers, the result is truncated (reduced) to the whole number part.

modulo: <Number> % <Number> = <Number>

- outputs the remainder of dividing the first number by the second

exponentiation: <Base> ** <Power> = <Number>

outputs the result of raising to the power (multiplying by itself number of times).

Comparisons equality: <Value> == <Value> = <Boolean>

- outputs True if the two input values are equal, False otherwise

inequality: <Value> != <Value> = <Boolean>

- outputs True if the two input values are not equal, False otherwise

greater than: <Number1> > <Number2> = <Boolean>

- outputs True if Number~1~ is greater than Number~2~

less than:<number~1~> <="" <number~2~>="<Boolean"></number~1~>>

- outputs True if Number~1~ is less than than Number~2~

greater or equal to: <Number~1~> >= <Number~2~> = <Boolean>

- outputs True if Number~1~, is not less than Number~2~

less than or equal to: <Number~1~> <= <Number~2~> = <Boolean>

- outputs True if Number~1~ is not greater than Number~2~

Variables and Assignment

Names

A variable name in Python can be any sequence of letters, numbers, and underscores (_) that do not start with a number. We usually use all lowercase letters for variable names, but capitalization must match exactly. Here are some valid examples of names in Python (but most of these would not be good choices to actually use in your programs):

- my_name
- one2one
- Dorina
- this_is_a_very_long_variable_name

Assignment Statement

An assignment statement assigns a value to a variable:

- =

After the assignment statement, the variable refers to the value of the on the right side of the assignment. An assignment is any Python construct that has a value.

Multiple Assignment

We can put more than one name on the left side of an assignment statement, and a corresponding number of expressions on the right side:

- <name~1~>, <name~2~>,="" ...="<Expression1">, , ...</name~1~>,>

All of the expressions on the right side are evaluated first. Then, each name on the left side is assigned to reference the value of the corresponding expression on the right side. This is handy for swapping variable values. For example,

- s, t = t, s

would swap the values of s and t so after the assignment statement s now refers to the previous value of t, and t refers to the previous value of s.

Procedures
A procedure takes inputs and produces outputs. It is an abstraction that provides a way to use the same code to operate on different data by passing in that data as its inputs.

Defining a procedure:

```
def <Name>(<Parameters>):
    <Block>
```

The <Parameters> are the inputs to the procedure. There is one <Name> for each input in order, separated by commas. There can be any number of parameters (including none).

To produce outputs:

```
return <Expression>, <Expression>, ...
```

There can be any number of expressions following the return (including none, in which case the output of the procedure is the special value None).

Using a procedure:

```
<"Procedure">(<"Input">, <"Input">, ...)
```

The number of inputs must match the number of parameters. The value of each input is assigned to the value of each parameter name in order, and then the block is evaluated.

If Statements
The if statement provides a way to control what code executes based on the result of a test expression.

```
if <TestExpression>:
   <Block>
```

The code in <Block> only executes if the <TestExpression> has a True value.

Alternate clauses. We can use an else clause in an if statement to provide code that will run when the <TestExpression> has a False value.

```
if <TestExpression>:
   <BlockTrue>
else:
   <BlockFalse>
```

Logical Operators

The and and or **operators behave similarly to logical conjunction (and) and disjunction (or). The important property they have which is different from other operators is that the second operand expression is evaluated only when necessary.

- <Expression~1~> and <Expression~2~>

IfExpression~1~ has a False value, the result is False and Expression~2~ is not evaluated (so even if it would produce an error it does not matter).

If Expression~1~ has a True value, the result of the and is the value of Expression~2~.

- <*Expression~1~>*' or* <Expression~2~>'*

If Expression~1~ has a True value, the result is True and Expression~2~ is not evaluated (so even if it would produce an error it does not matter).

If Expression~1~ has a False value, the result of the or is the value of Expression~2~'.

Loops
Loops provide a way to evaluate the same block of code an arbitrary number of times.

While Loops
A while loop provides a way to keep executing a block of code as long as a test expression is True.

```
while <TestExpression>:
    <Block>
```

If the <TestExpression> evaluates to False, the while loop is done and execution continues with the following statement.

If the <TestExpression> evaluates to True, the <Block> is executed. Then, the loop repeats, returning to the <TestExpression> and continuing to evaluate the <Block> as long as the <TestExpression> is True.

Break Statement
A break statement in the <Block> of a while loop, jumps out of the containing while loop, continuing execution at the following statement.

```
break
```

For Loops
A for loop provides a way to execute a block once for each element of a collection:

```
for <Name> in <Collection>:
  <Block>
```

The loop goes through each element of the collection, assigning that element to the <Name> and evaluating the <Block>. The collection could be a String, in which case the elements are the characters of a string; a List, in which case the elements are the elements of the list; a Dictionary, in which case the elements are the keys in the dictionary; or many other types in Python that represent collections of objects.

Strings
A string is sequence of characters surrounded by quotes. The quotes can be either single or double quotes, but the quotes at both ends of the string must be the same type. Here are some examples of strings in Python:

- "silly"
- 'string'
- "I'm a valid string, even with a single quote in the middle!"

String Operations

length: len(<String>) = <Number>

- Outputs the number of characters in <String>

string concatenation: <String> + <String> = <String>

- Outputs the concatenation of the two input strings (pasting the strings together with no space between them)

string multiplication: <String> * <Number> = <String>

- Outputs a string that is <Number> copies of the input pasted together

INDEXING STRINGS

string indexing: <String>[<Number>] = <String>

- The indexing operator provides a way to extract subsequences of characters from a string.
-
- Outputs a single-character string containing the character at position of the input . Positions in the string are counted starting from 0, so s[1] would output the second character in s. If the <Number> is negative, positions are counted from the end of the string: s[-1] is the last character in s.

string extraction: <String>[<Start Number>:<Stop Number>] = <String>

- Outputs a string that is the subsequence of the input string starting from position <Start Number> and ending just before position <Stop Number>. If <Start Number> is missing, starts from the beginning of the input string; if <Stop Number> is missing, goes to the end of the input string.

find: <Search String>.find(<Target String>) = <Number>

- The find method provides a way to find sub-sequences of characters in strings.
- Outputs a number giving the position in <Search String> where <Target String> first appears. If there is no occurrence of <Target String> in <Search String>, outputs -1.

To find later occurrences, we can also pass in a number to find:

find after: <Search String>.find(<Target String>, <Start Number>) = <Number>

- Outputs a number giving the position in <Search String> where <Target String> first appears that is at or after the position give by <Start Number>. If there is no occurrence of <Target String> in <Search String> at or after <Start Number>, outputs -1.

CONVERTING BETWEEN NUMBERS AND STRINGS

str: str(<Number>) =

- Outputs a string that represents the input number. For example, str(23) outputs the string '23'.

ord: ord(<One-Character String>) = <Number>

- Outputs the number corresponding to the input string.

chr: chr(<Number>) = <One-Character String>

- Outputs the one-character string corresponding to the number input. This function is the inverse of ord: chr(ord(a)) = a for any one-character string a.

SPLITTING STRINGS

split: <String>.split() = [<String>, <String>, ...]

- outputs a list of strings that are (roughly) the words contained in the input string. The words are determined by whitespace (either spaces, tabs, or newlines) in the string.

LOOPING THROUGH STRINGS

A for loop provides a way to execute a block once for each character in a string (just like looping through the elements of a list):

```
for <"Name"> in <"String">:
   <"Block">
```

The loop goes through each character of the string in turn, assigning that element to the <Name> and evaluating the <Block>.

LISTS

A list is a mutable (changeable) collection of objects. The elements in a list can be of any type, including other lists.

Constructing a list. A list is a sequence of zero or more elements, surrounded by square brackets:

- [<Element>, <Element>, ...]

Constructing a list of integers (numbers). The range function can help us in this task.

This is a function to create lists containing arithmetic progressions. It is most often used in for loops.

The arguments must be plain integers. If the step argument is omitted, it defaults to 1. If the start argument is omitted, it defaults to 0.

- range([start], stop[, step]).

Example:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

Selecting elements: <List>[<Number>] = <Element>

- Outputs the value of the element in at position . Elements are indexed starting from 0.

Selecting sub-sequences: <List>[<Start> : <Stop>] = <List>

- Outputs a sub-sequence of starting from position , up to (but not including) position .

Update: <List>[<Number>] = <Value>

- Modifies the value of the element in <List> at position <Number> to be <Value>.

Length: len(<List>) = <Number>

- Outputs the number of (top-level) elements in.

Append: <List>.append(<Element>)
- Mutates <List> by adding <Element> to the end of the list.

Concatenation: <List~1~> + <List~2~> = <Element>
- Outputs a new list that is the elements of <List~1~> followed by the elements of <List~2~>.

Popping: <List>.pop() = <Element>
- Mutates <List> by removing its last element. Outputs the value of that element. If there are no elements in <List>, [].pop() produces an error.

Finding: <List>.index(<Value>) = <Number>
- Outputs the position of the first occurrence of an element matching <Value> in <List>. If <Value> is not found in <List>, produces an error.

Membership: <Value> in <List> = <Boolean>
- Outputs True if <Value> occurs in <List>. Otherwise, outputs False.

Non-membership: <Value> not in <List> = <Boolean>
- Outputs False if <Value> occurs in <List>. Otherwise, outputs True.

LOOPS ON LISTS

A for loop provides a way to execute a block once for each element of a List:

```
for <"Name"> in <"List">:
   <"Block">
```

DICTIONARIES

A Dictionary provides a mapping between keys, which can be values of any immutable type, and values, which can be any value.

Because a Dictionary is implemented using a hash table, the time to lookup a value does not increase (significantly) even when the number of keys increases.

Constructing a Dictionary. A Dictionary is a set of zero or more key-value pairs, surrounded by squiggly braces:

- { : , : , ... }

Looking up elements: <Dictionary>[<Key>] = <Value>
- outputs the value associated with <Key> in the <Dictionary>. Produces an error if the <Key> is not a key in the <Dictionary>.
-

Updating elements: <Dictionary>[<Key>] = <Value>
- updates the value associated with <Key> in the <Dictionary> to be <Value>. If <Key> is already a key in <Dictionary>, replaces the value associated with <Key>; if not, adds a new <Key>: <Value> pair to the <Dictionary>.
-

Membership: <Key> in <Dictionary> = <Boolean>
- outputs True if <Key> is a key in <Dictionary>, False otherwise.

LOOPS ON DICTIONARIES

A for loop provides a way to execute a block once for each key in a Dictionary:

```
for <Name> in <Dictionary>:
          <Block>
```

Eval The eval function provides a way to evaluate an input string as a Python expression:
- eval() =

The input is a string that is a Python expression; the value is the value that string would evaluate to in Python.

Time The time library provides functions for obtaining and measuring time.
- import time

System time: time.clock()= <Number>
- outputs the processor time in seconds (a real number, including fractional seconds with limited accuracy) (Note that what clock means depends on your platform, and it may not provide accurate timings on all platforms.)

EXCEPTIONS

```
try:
   <"Try Block">
except:
   <"Handler Block">
```

Execute the code in the <Try Block>. If it completes normally, skip the and continue with the following statement. If code in the <Try Block> raises an error, jump to the code in the <Handler Block>.

The except can be followed by an exception type and a variable name:

- except <Exception Type>, <Name>:
    - <Handler Block>

If the exception raised in the <Try Block> matches the type <Exception Type>, evaluate the <Handler Block> with the variable <Name> referring to the exception object.

LIBRARIES

- import <Library>

Imports the <Library> into the Python environment, enabling the following code to use the definitions provided in the <Library>.

Python provides many libraries such as urllib that we use for downloading web pages in get_page, and you can also import your own code. As programs get