**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **24/July/2020** | **Name:** | **Prashantha naik** |
| **Course:** | **Python workshop** | **USN:** | **4al17ec074** |
| **Topic:** | **Case studies – ‘How to build Pythonic code in List, Tuple and**  **Dictionary?’** | **Semester & Section:** | **6th b** |
| **GitHub Repository:** | **prashanth\_course** |  |  |

|  |
| --- |
| **SESSION DETAILS** |
| **Image of session** |
| **Report – Report can be typed or hand written for up to two pages.**  **Example #1**  x=[1, 2, 3, 4, 5, 6]  result = []  for idx in range(len(x)):  result.append(x[idx] \* 2)  print(result)  **Output: [2, 4, 6, 8, 10, 12]**  Consider the above code, where you’re trying to multiply some elements, “x” by 2.  So, what we did here was, we created an empty list to store the results. We would then append the solution of the computation into the result. The result now contains a function which is 2 multiplied by each of the elements.  Now, if you were to write the same code in a Pythonic way, you might want to simply use **list comprehensions.**  Here’s how:  x=[1, 2, 3, 4, 5, 6]  print([(element \* 2) for element in x])  **only 2 line! it is pythonic**  **Example #2**  x=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  result = []  for idx in range(len(x)):  if x[idx] % 2 == 0:  result.append(x[idx] \* 2)  else:  result.append(x[idx])  print(result|)  We’ve actually created an if else statement to solve this problem, but there is a simpler way of doing things t**he Pythonic way. The Pythonic way is to combine for and if using list comprehension**  x=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  [(element \* 2 if element % 2 == 0 else element) for element in x]  **Output: [1, 4, 3, 8, 5, 12, 7, 16, 9, 20]**  **Example #3 filtering only even numbers**  x=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  result = []  for idx in range(len(x)):  if x[idx] % 2 == 0  print(result|)  We’ve actually created an if else statement to solve this problem, but there is a simpler way of doing things t**he Pythonic way.**  x=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  [element \* 2 for element in x if element % 2 == 0] How to Write Beautiful Python Code With PEP 8 PEP stands for Python Enhancement Proposal, and there are several of them. A PEP is a document that describes new features proposed for Python and documents aspects of Python, like design and style, for the community. Naming Conventions When you write Python code, you have to name a lot of things: variables, functions, classes, packages, and so on. Choosing sensible names will save you time and energy later. You’ll be able to figure out, from the name, what a certain variable, function, or class represents. You’ll also avoid using inappropriate names that might result in errors that are difficult to debug.  **Never use l, O, or I single letter names as these can be mistaken for 1 and 0, depending on typeface:**  O = 2 # This may look like you're trying to reassign 2 to zero Naming Styles  How to Choose Names? Choosing names for your variables, functions, classes, and so forth can be challenging. You should put a fair amount of thought into your naming choices when writing code as it will make your code more readable. The best way to name your objects in Python is to use descriptive names to make it clear what the object represents.  When naming variables, you may be tempted to choose simple, single-letter lowercase names, like x. But, unless you’re using x as the argument of a mathematical function, it’s not clear what x represents. Imagine you are storing a person’s name as a string, and you want to use string slicing to format their name differently. You could end up with something like this:  # Not recommended  x = 'John Smith'  y, z = x.split()  print(z, y, sep=', ')  'Smith, John'  This will work, but you’ll have to keep track of what x, y, and z represent. It may also be confusing for collaborators. A much clearer choice of names would be something like this:  name = 'John Smith'  first\_name, last\_name = name.split()  print(last\_name, first\_name, sep=', ')  'Smith, John'  Similarly, to reduce the amount of typing you do, it can be tempting to use abbreviations when choosing names. In the example below, I have defined a function db() that takes a single argument x and doubles it:  # Not recommended  def db(x):  return x \* 2  At first glance, this could seem like a sensible choice. db() could easily be an abbreviation for double. But imagine coming back to this code in a few days. You may have forgotten what you were trying to achieve with this function, and that would make guessing how you abbreviated it difficult.  The following example is much clearer. If you come back to this code a couple of days after writing it, you’ll still be able to read and understand the purpose of this function:  # Recommended  def multiply\_by\_two(x):  return x \* 2  The same philosophy applies to all other data types and objects in Python. Always try to use the most concise but descriptive names possible. Code LayoutBlank Lines Vertical whitespace, or blank lines, can greatly improve the readability of your code. Code that’s bunched up together can be overwhelming and hard to read. Similarly, too many blank lines in your code makes it look very sparse, and the reader might need to scroll more than necessary. Below are three key guidelines on how to use vertical whitespace.  Surround top-level functions and classes with two blank lines. Top-level functions and classes should be fairly self-contained and handle separate functionality. It makes sense to put extra vertical space around them, so that it’s clear they are separate:  class MyFirstClass:  pass  class MySecondClass:  pass  def top\_level\_function():  return None  Surround method definitions inside classes with a single blank line. Inside a class, functions are all related to one another. It’s good practice to leave only a single line between them:  class MyClass:  def first\_method(self):  return None  def second\_method(self):  return None  Use blank lines sparingly inside functions to show clear steps. Sometimes, a complicated function has to complete several steps before the return statement. To help the reader understand the logic inside the function, it can be helpful to leave a blank line between each step.  In the example below, there is a function to calculate the variance of a list. This is two-step problem, so I have indicated each step by leaving a blank line between them. There is also a blank line before the return statement. This helps the reader clearly see what’s returned:  def calculate\_variance(number\_list):  sum\_list = 0  for number in number\_list:  sum\_list = sum\_list + number  mean = sum\_list / len(number\_list)  sum\_squares = 0  for number in number\_list:  sum\_squares = sum\_squares + number\*\*2  mean\_squares = sum\_squares / len(number\_list)  return mean\_squares - mean\*\*2 Maximum Line Length and Line Breaking PEP 8 suggests lines should be limited to 79 characters. This is because it allows you to have multiple files open next to one another, while also avoiding line wrapping.  Of course, keeping statements to 79 characters or less is not always possible. PEP 8 outlines ways to allow statements to run over several lines.  Python will assume line continuation if code is contained within parentheses, brackets, or braces:  def function(arg\_one, arg\_two,  arg\_three, arg\_four):  return arg\_one  If it is impossible to use implied continuation, then you can use backslashes to break lines instead:  from mypkg import example1, \  example2, example3  However, if you can use implied continuation, then you should do so.  If line breaking needs to occur around binary operators, like + and \*, it should occur before the operator. This rule stems from mathematics. Mathematicians agree that breaking before binary operators improves readability. Compare the following two examples.  Below is an example of breaking before a binary operator:  # Recommended  total = (first\_variable  + second\_variable  - third\_variable)  # Not Recommended  total = (first\_variable +  second\_variable -  third\_variable) Indentation Indentation, or leading whitespace, is extremely important in Python. The indentation level of lines of code in Python determines how statements are grouped together.  Consider the following example:  if x > 5:  print('x is larger than 5')  The indented print statement lets Python know that it should only be executed if the if statement returns True. The same indentation applies to tell Python what code to execute when a function is called or what code belongs to a given class.  The key indentation rules laid out by PEP 8 are the following:   * Use 4 consecutive spaces to indicate indentation. * Prefer spaces over tabs.   # Not Recommended  var = function(arg\_one, arg\_two,  arg\_three, arg\_four)  # Not Recommended  def function(  arg\_one, arg\_two,  arg\_three, arg\_four):  return arg\_one  # Recommended  def function(  arg\_one, arg\_two,  arg\_three, arg\_four):  return arg\_one Where to Put the Closing Brace? Line up the closing brace with the first character of the line that starts the construct:  list\_of\_numbers = [  1, 2, 3,  4, 5, 6,  7, 8, 9  ] Whitespace in Expressions and Statements Whitespace can be very helpful in expressions and statements when used properly. If there is not enough whitespace, then code can be difficult to read, as it’s all bunched together. If there’s too much whitespace, then it can be difficult to visually combine related terms in a statement. Whitespace Around Binary Operators Surround the following binary operators with a single space on either side:   * Assignment operators (=, +=,-=, and so forth) * Comparisons (==, !=, >, <. >=, <=) and (is, is not, in, not in) * Booleans (and, not, or)   Note: When = is used to assign a default value to a function argument, do not surround it with spaces.  # Recommended  def function(default\_parameter=5):  # ...  # Not recommended  def function(default\_parameter = 5):  # ...  When there’s more than one operator in a statement, adding a single space before and after each operator can look confusing. Instead, it is better to only add whitespace around the operators with the lowest priority, especially when performing mathematical manipulation. Here are a couple examples:  # Recommended  y = x\*\*2 + 5  z = (x+y) \* (x-y)  # Not Recommended  y = x \*\* 2 + 5  z = (x + y) \* (x - y)  # Not recommended  if x > 5 and x % 2 == 0:  print('x is larger than 5 and divisible by 2!')  # Recommended  if x>5 and x%2==0:  print('x is larger than 5 and divisible by 2!')  # Definitely do not do this!  if x >5 and x% 2== 0:  print('x is larger than 5 and divisible by 2!')  Don’t use if x: when you mean if x is not None: Sometimes, you may have a function with arguments that are None by default. A common mistake when checking if such an argument,arg, has been given a different value is to use the following:  # Not Recommended  if arg:  # Do something with arg...  # Recommended  if arg is not None:  # Do something with arg...  The mistake being made here is assuming that not None and truthy are equivalent. You could have set arg = []. As we saw above, empty lists are evaluated as falsy in Python. So, even though the argument arghas been assigned, the condition is not met, and so the code in the body of the if statement will not be executed.  Use .startswith() and.endswith()instead of slicing.If you were trying to check if a string word was prefixed, or suffixed, with the word cat, it might seem sensible to use [**list slicing**](https://realpython.com/python-strings/#string-slicing). However, list slicing is prone to error, and you have to hardcode the number of characters in the prefix or suffix. It is also not clear to someone less familiar with Python list slicing what you are trying to achieve:  # Not recommended  if word[:3] == 'cat':  print('The word starts with "cat"')  # Recommended  if word.startswith('cat'):  print('The word starts with "cat"')  # Not recommended  if file\_name[-3:] == 'jpg':  print('The file is a JPEG')  # Recommended  if file\_name.endswith('jpg'):  print('The file is a JPEG') |