pragmatic_introduction_to_python_language_2

February 28, 2021

1 Writing Structured Programs

By now you will have a sense of the capabilities of the Python programming language for processing natural language. However, if you're new to Python or to programming, you may still be wrestling with Python and not feel like you are in full control yet. In this chapter we'll address the following questions:

- How can you write well-structured, readable programs that you and others will be able to re-use easily?
- How do the fundamental building blocks work, such as loops, functions and assignment?
- What are some of the pitfalls with Python programming and how can you avoid them?

Along the way, you will consolidate your knowledge of fundamental programming constructs, learn more about using features of the Python language in a natural and concise way, and learn some useful techniques in visualizing natural language data.

1.1 Back to the Basics

1.1.1 Assignment

Assignment would seem to be the most elementary programming concept, not deserving a separate discussion. However, there are some surprising subtleties here. Consider the following code fragment:

[51]:
```
a = 'Monty'
b = a
a = 'Python'
print(b)
```

```
Monty
```

This behaves exactly as expected. When we write `b = a` in the above code, the value of `a` (the string `'Monty'`) is assigned to `b`. That is, **`b` is a copy of `a`**, so when we overwrite a with a new string `'Python'`, the value of `b` is not affected.

However, assignment statements do not always involve making copies in this way. Assignment always copies the value of an expression, but a value is not always what you might expect it to be. In particular, the "value" of a structured object such as a list is actually just a **reference** to the object. In the following example, assigns the reference of `a` to the new variable `b`. Now when we modify something inside `a` on line, we can see that the contents of bar have also been changed.

[52]:
```
a = ['Monty', 'Python']
b = a
print("Now we print b list...")
print(b)
a[1] = 'Jupyter'
print("Now we print again b list...")
print(b)
```

```
Now we print b list…
['Monty', 'Python']
Now we print again b list…
['Monty', 'Jupyter']
```

[53]:
```
empty=[]
nested = [empty, empty, empty]
nested
```

[53]:
```
[[], [], []]
```

[54]:
```
nested[0].append('Python')
nested
```

[54]:
```
[['Python'], ['Python'], ['Python']]
```

[55]:
```
nested[1] = 'Monty'
nested
```

[55]:
```
[['Python'], 'Monty', ['Python']]
```

[56]:
```
nested[0].append('Jupyter')
nested
```

[56]:
```
[['Python', 'Jupyter'], 'Monty', ['Python', 'Jupyter']]
```

1.1.2 Equality

Python provides two ways to check that a pair of items are the same. The `is` operator tests for object identity. We can use it to verify our earlier observations about objects. First we create a list containing several copies of the same object, and demonstrate that they are not only identical according to `==`, but also that they are one and the same object: “

[57]:
```
size     = 5
x        = ['Python']
myList   = [x] * 5
print(myList)
# check if values are the same...
print(myList[0] == myList[1] == myList[2] == myList[3] == myList[4])
# check if objects are the same...
```

```
print(myList[0] is myList[1] is myList[2] is myList[3] is myList[4])
```

```
[['Python'], ['Python'], ['Python'], ['Python'], ['Python']]
True
True
```

[58]:
```
myList[1] = ['Python']
print(myList)
# check if values are the same...
print(myList[0] == myList[1] == myList[2] == myList[3] == myList[4])
# check if objects are the same...
print(myList[0] is myList[1] is myList[2] is myList[3] is myList[4])
```

```
[['Python'], ['Python'], ['Python'], ['Python'], ['Python']]
True
False
```

[59]:
```
id_list = [id(x) for x in myList]
print(id_list)
```

```
[2357511477320, 2357511477576, 2357511477320, 2357511477320, 2357511477320]
```

1.2 Functions: The Foundation of Structured Programming

Functions provide an effective way to package and re-use program code, as already explained in ?. They also help make it reliable. When we re-use code that has already been developed and tested, we can be more confident that it handles a variety of cases correctly. We also remove the risk that we forget some important step, or introduce a bug. The program that calls our function also has increased reliability. The author of that program is dealing with a shorter program, and its components behave transparently.

To summarize, as its name suggests, a function captures functionality. It is a segment of code that can be given a meaningful name and which performs a well-defined task. Functions allow us to abstract away from the details, to see a bigger picture, and to program more effectively.

1.2.1 Function Inputs and Outputs

We pass information to functions using a function's parameters, the parenthesized list of variables and constants following the function's name in the function definition. Here's a complete example:

[60]:
```
def repeat(msg, num):
    return ' '.join([msg] * num)

monty = 'Monty Python'
repeat(monty, 3)
```

[60]:
```
'Monty Python Monty Python Monty Python'
```

- We first define the function to take two parameters, `msg` and `num`;

- Then we call the function and pass it two arguments, `monty` and `3`;
- these arguments fill the "placeholders" provided by the parameters and provide values for the occurrences of `msg` and `num` in the function body.

It is not necessary to have any parameters.A function usually communicates its results back to the calling program via the `return` statement, as we have just seen.

1.2.2 Parameter Passing

As we have previously seen assignment works on values, but that the value of a structured object is a reference to that object. The same is true for functions. Python interprets function parameters as values (this is known as call-by-value). In the following code, set_up() has two parameters, both of which are modified inside the function. We begin by assigning the string 'PUT' to w and an empty list to p. After calling the function, w is unchanged, while p is changed:

[61]:
```
def set_up(option_type, properties):
    option_type = 'CALL'
    properties.append('strike')
    properties = 5

w = 'PUT'
p = ['spot', 'volatility']

set_up(w, p)

print(w)
print(p)
```

```
PUT
['spot', 'volatility', 'strike']
```

Notice that w was not changed by the function. When we called set_up(w, p), the value of w (an empty string) was assigned to a new variable called 'option_type'. Inside the function, the value of 'option_type' was modified. However, that change did not propagate to w.

Let's look at what happened with the list p. When we called set_up(w, p), the value of p (a reference to an empty list) was assigned to a new local variable properties, so both variables now reference the same memory location. The function modifies properties, and this change is also reflected in the value of p as we saw. The function also assigned a new value to properties (the number 5); this did not modify the contents at that memory location, but created a new local variable.

Thus, to understand Python's call-by-value parameter passing, it is enough to understand how assignment works. Remember that you can use the id() function and is operator to check your understanding of object identity after each statement.

1.2.3 Variable Scope

Function definitions create a new, local scope for variables. When you assign to a new variable inside the body of a function, the name is only defined within that function. The name is not visible outside the function, or in other functions. This behavior means you can choose variable names without being concerned about collisions with names used in your other function definitions.

When you refer to an existing name from within the body of a function, the Python interpreter first tries to resolve the name with respect to the names that are local to the function. If nothing is found, the interpreter checks if it is a global name within the module. Finally, if that does not succeed, the interpreter checks if the name is a Python built-in. This is the so-called LGB rule of name resolution: local, then global, then built-in.

1.2.4 Checking Parameter Types

Python does not allow us to declare the type of a variable when we write a program, and this permits us to define functions that are flexible about the type of their arguments. For example, a tagger might expect a sequence of words, but it wouldn't care whether this sequence is expressed as a list or a tuple (or an iterator, another sequence type that is outside the scope of the current discussion).

However, often we want to write programs for later use by others, and want to program in a defensive style, providing useful warnings when functions have not been invoked correctly. The author of the following tag() function assumed that its argument would always be a string.

[62]:
```
def tag(word):
    if word in ['a', 'the', 'all']:
        return 'det'
    else:
        return 'noun'

tag('the')
'det'
tag('knight')
'noun'
tag(["nothing", 'but', 'a', 'scratch'])
'noun'
```

[62]:
```
'noun'
```

The function returns sensible values for the arguments 'the' and 'knight', but look what happens when it is passed a list — it fails to complain, even though the result which it returns is clearly incorrect. The author of this function could take some extra steps to ensure that the word parameter of the `tag()` function is a string. A naive approach would be to check the type of the argument using `if not type(word) is str`, and if word is not a string, to simply return Python's special empty value, `None`. This is a slight improvement, because the function is checking the type of the argument, and trying to return a "special", diagnostic value for the wrong input. However, it is also dangerous because the calling program may not detect that `None` is intended as a "special" value, and this diagnostic return value may then be propagated to other parts of the program with unpredictable consequences. Here's a better solution, using an `assert` statement.

[63]:
```
def tag(word):
     assert isinstance(word, str), "argument to tag() must be a string"
```

```
    if word in ['a', 'the', 'all']:
        return 'det'
    else:
        return 'noun'
```

[64]:
```
tag('the')
```

[64]:
```
'det'
```

[65]:
```
tag(['a','the','all'])
```

```
        ␣
↪---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call␣
↪last)

        <ipython-input-65-b01a07bc51d3> in <module>
    ----> 1 tag(['a','the','all'])


        <ipython-input-63-8056b1d249d8> in tag(word)
          1 def tag(word):
    ----> 2     assert isinstance(word, str), "argument to tag() must be a␣
↪string"
          3     if word in ['a', 'the', 'all']:
          4         return 'det'
          5     else:


        AssertionError: argument to tag() must be a string
```

If the assert statement fails, it will produce an error that cannot be ignored, since it halts program execution. Additionally, the error message is easy to interpret. Adding assertions to a program helps you find logical errors, and is a kind of **defensive programming**.

A more fundamental approach is to document the parameters to each function using docstrings as described later in this section.

1.2.5 Functional Decomposition

Well-structured programs usually make extensive use of functions. When a block of program code grows longer than 10-20 lines, it is a great help to readability if the code is broken up into one or more functions, each one having a clear purpose. This is analogous to the way a good essay is divided into paragraphs, each expressing one main idea.

Functions provide an important kind of abstraction. They allow us to group multiple actions into a single, complex action, and associate a name with it.

Appropriate use of functions makes programs more readable and maintainable. Additionally, it becomes possible to reimplement a function — replacing the function's body with more efficient code — without having to be concerned with the rest of the program.

[66]:
```
import nltk

from urllib import request
from bs4 import BeautifulSoup

def freq_words(url, freqdist, n):
    html = request.urlopen(url).read().decode('utf8')
    raw = BeautifulSoup(html, 'html.parser').get_text()
    for word in nltk.word_tokenize(raw):
        freqdist[word.lower()] += 1
    result = []
    for word, count in freqdist.most_common(n):
        result = result + [word]
    print(result)
```

[67]:
```
book = "https://www.gutenberg.org/files/84/84-h/84-h.htm"
fd = nltk.FreqDist()
freq_words(book, fd, 30)
```

```
[',', 'the', 'and', 'i', '.', 'of', 'to', 'my', 'a', 'in', 'that', 'was', ';',
'me', 'with', 'but', 'had', 'you', 'he', 'not', 'which', 'it', 'as', 'his',
'for', 'by', '"', 'on', 'this', 'from']
```

This function has a number of problems. The function has two side-effects: it modifies the contents of its second parameter, and it prints a selection of the results it has computed. The function would be easier to understand and to reuse elsewhere if we initialize the `FreqDist()` object inside the function (in the same place it is populated), and if we moved the selection and display of results to the calling program. Given that its task is to identify frequent words, it should probably just return a list, not the whole frequency distribution.

[68]:
```
def freq_words(url, n):
    html = request.urlopen(url).read().decode('utf8')
    text = BeautifulSoup(html, 'html.parser').get_text()
    freqdist = nltk.FreqDist(word.lower() for word in nltk.word_tokenize(text))
    return [word for (word, _) in fd.most_common(n)]
```

[69]:
```
freq_words(book, 5)
```

[69]:
```
[',', 'the', 'and', 'i', '.']
```

1.2.6 Documenting Functions

[70]:
```
def accuracy(reference, test):
    """
    Calculate the fraction of test items that equal the corresponding reference␣
 ↪items.

    Given a list of reference values and a corresponding list of test values,
    return the fraction of corresponding values that are equal.
    In particular, return the fraction of indexes
    {0<i<=len(test)} such that C{test[i] == reference[i]}.

        >>> accuracy(['ADJ', 'N', 'V', 'N'], ['N', 'N', 'V', 'ADJ'])
        0.5

    :param reference: An ordered list of reference values
    :type reference: list
    :param test: A list of values to compare against the corresponding
        reference values
    :type test: list
    :return: the accuracy score
    :rtype: float
    :raises ValueError: If reference and length do not have the same length
    """

    if len(reference) != len(test):
        raise ValueError("Lists must have the same length.")
    num_correct = 0
    for x, y in zip(reference, test):
        if x == y:
            num_correct += 1
    return float(num_correct) / len(reference)
```

1.2.7 Functions as Arguments

So far the arguments we have passed into functions have been simple objects like strings, or structured objects like lists. Python also lets us pass a function as an argument to another function. Now we can abstract out the operation, and apply a different operation on the same data. As the following examples show, we can pass the built-in function len() or a user-defined function last_letter() as arguments to another function:

[71]:
```
sent = ['Take', 'care', 'of', 'the', 'sense', ',', 'and', 'the',
         'sounds', 'will', 'take', 'care', 'of', 'themselves', '.']
```

[72]:
```
def extract_property(prop):
     return [prop(word) for word in sent]
```

```
[73]: extract_property(len)
```

```
[73]: [4, 4, 2, 3, 5, 1, 3, 3, 6, 4, 4, 4, 2, 10, 1]
```

```
[74]: def last_letter(word):
          return word[-1]
```

```
[75]: extract_property(last_letter)
```

```
[75]: ['e', 'e', 'f', 'e', 'e', ',', 'd', 'e', 's', 'l', 'e', 'e', 'f', 's', '.']
```

```
[ ]:
```

1.3 Error Handling

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

1.4 OOP: Object Oriented Programming

Object Oriented Programming (OOP for short) is a particular way of programming that focuses on where responsibility rests with various tasks. The idea behind object-oriented programming is that a computer program is composed of a collection of individual units, or objects, as opposed to a traditional view in which a program is a list of instructions to the computer. Each object is capable of receiving messages, processing data, and sending messages to other objects and should be responsible only for a particular task. Just to have a clue on what an object is, you can think of it as data and functionality packaged together in some way to form a single unit of well identified code (examples will soon follow).

What is then peculiar in this approach is that special attention is given to creating the appropriate objects as opposed to focusing solely on solving the problem. For this reason OOP is often called a paradigm rather than a style or type of programming, to emphasize that OOP can change the way software is developed, by changing the way that programmer think about it. A programming paradigm provides (and determines) the view that the programmer has of the execution of the program. On one hand, for instance, in functional programming a program can be thought of as a simple sequence of function evaluations. On the other hand, as we have already pointed out, in object-oriented programming programmers can think of a program as a collection of interacting objects. Therefore the Paradigm of OOP is essentially a paradigm of design. The challenge in OOP therefore is of designing a well defined object system.

The best way to explain the methodological approach we are talking about is to give a practical example….

[ ]:

[ ]:

[ ]:

[ ]:

1.5 Working with Numbers

1.5.1 Armstrong Number, a Question by Amazon

The k-digit number N is an **Armstrong number** if and only if the k-th power of each digit sums to N. - Given a positive integer N, return true if and only if it is an Armstrong number.

the original problem

A naive approach would be to turn an integer into a string and iterate over the entire string

[76]:
```
def Armstrong(num):

    k = len(str(num))

    digit_sum =0

    for i in str(num):
        digit_sum += int(i)**k

    return digit_sum == num

# test case
Armstrong(num = 153)
```

[76]:
```
True
```

[ ]:

1.5.2 Matplotlib

1.5.3 NumPy

Even though numpy arrays (often written as ndarrays, for n-dimensional arrays) are not part of the core Python libraries, they are so useful in scientific Python that we'll include them here in the core lesson. Numpy arrays are collections of things, all of which must be the same type, that work similarly to lists (as we've described them so far). The most important are:

1. You can easily perform elementwise operations (and matrix algebra) on arrays
2. Arrays can be n-dimensional
3. There is no equivalent to append, although arrays can be concatenated

Arrays can be created from existing collections such as lists, or instantiated “from scratch” in a few useful ways.

When getting started with scientific Python, you will probably want to try to use ndarrays whenever possible, saving the other types of collections for those cases when you have a specific reason to use them.

[77]:
```
# We need to import the numpy library to have access to it
# We can also create an alias for a library, this is something you will␣
↪commonly see with numpy
import numpy as np
```

[78]:
```
# Make an array from a list
alist = [2, 3, 4]
blist = [5, 6, 7]
a = np.array(alist)
b = np.array(blist)
print(a, type(a))
print(b, type(b))
```

```
[2 3 4] <class 'numpy.ndarray'>
[5 6 7] <class 'numpy.ndarray'>
```

[79]:
```
# Do arithmetic on arrays
print(a**2)
print(np.sin(a))
print(a * b)
print(a.dot(b), np.dot(a, b))
```

```
[ 4  9 16]
[ 0.90929743  0.14112001 -0.7568025 ]
[10 18 28]
56 56
```

[80]:
```
# Boolean operators work on arrays too, and they return boolean arrays
print(a > 2)
print(b == 6)

c = a > 2
print(c)
print(type(c))
print(c.dtype)
```

```
[False  True  True]
[False  True False]
[False  True  True]
<class 'numpy.ndarray'>
bool
```

[81]:
```
# Indexing arrays
print(a[0:2])

c = np.random.rand(3,3)
print(c)
print('\n')
print(c[1:3,0:2])

c[0,:] = a
print('\n')
print(c)
```

```
[2 3]
[[0.45359614 0.54600725 0.64543144]
 [0.88365808 0.32978443 0.39890602]
 [0.53247524 0.8112649  0.33295592]]


[[0.88365808 0.32978443]
 [0.53247524 0.8112649 ]]


[[2.         3.         4.        ]
 [0.88365808 0.32978443 0.39890602]
 [0.53247524 0.8112649  0.33295592]]
```

[82]:
```
# Arrays can also be indexed with other boolean arrays
print(a)
print(b)
print(a > 2)
print(a[a > 2])
print(b[a > 2])

b[a == 3] = 77
print(b)
```

```
[2 3 4]
[5 6 7]
[False  True  True]
[3 4]
[6 7]
[ 5 77  7]
```

[83]:
```
# ndarrays have attributes in addition to methods
#c.
print(c.shape)
print(c.prod())
```

```
c.prod?
```

```
(3, 3)
0.401277757213792
```

[84]:
```
# There are handy ways to make arrays full of ones and zeros
print(np.zeros(5), '\n')
print(np.ones(5), '\n')
print(np.identity(5), '\n')
```

```
[0. 0. 0. 0. 0.]

[1. 1. 1. 1. 1.]

[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

[85]:
```
# You can also easily make arrays of number sequences
print(np.arange(0, 10, 2))
```

```
[0 2 4 6 8]
```

[ ]:

[ ]:

[ ]:

2 Credits

Steven Bird, Ewan Klein and Edward Loper, **Natural Language Processing with Python**, O'REILLY

Andrea Gigli, **Python Course for Data Science** here the link to github

[ ]: