Lesson 5 Intro Python

June 2, 2020

1 An informal introduction to Python

Python is a popular programming language that is reliable, flexible, easy to learn, free to use on all operating systems, and supported by both a strong developer community and many free libraries. Python supports all manners of development, including web applications, web services, desktop apps, scripting, data science, scientific computing, and Jupyter notebooks. Python is a language used by many universities, scientists, casual developers, and professional developers alike.

You can learn more about the language on python.org and Python for Beginners.

This walkthrough is a version of the python.org introductory tutorial, with some edits and amendments. This introduction is written for Python 3.6.7 but is generally applicable to other Python 3.x versions.

2 Comments

Many of the examples in this notebook include comments. Comments in Python start with the hash character, #, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

[1]:
```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
print(text)
```

```
# This is not a comment because it's inside quotes.
```

3 Explore basic data types

Let's try some simple Python commands to explore numbers, strings, and lists.

3.1 Numbers

The Python interpreter can act as a simple calculator: type an expression at it outputs the value.

Expression syntax is straightforward: the operators +, -, * and / work just like in most other programming languages (such as Pascal or C); parentheses (()) can be used for grouping and order or precedence. For example:

[2]:
```
2 + 2
```

[2]:
```
4
```

[4]:
```
50 - 5*6
```

[4]:
```
20
```

[5]:
```
(50 - 5*6) / 4
```

[5]:
```
5.0
```

[6]:
```
8 / 5  # Division always returns a floating point number.
```

[6]:
```
1.6
```

The integer numbers (e.g. 2, 4, 20) have type int, the ones with a fractional part (e.g. 5.0, 1.6) have type float. We'll see more about numeric types later in the tutorial.

Division (/) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use %:

[7]:
```
17 / 3  # Classic division returns a float.
```

[7]:
```
5.666666666666667
```

[8]:
```
17 // 3  # Floor division discards the fractional part.
```

[8]:
```
5
```

[9]:
```
17 % 3  # The % operator returns the remainder of the division.
```

[9]:
```
2
```

[10]:
```
5 * 3 + 2  # result * divisor + remainder
```

[10]:
```
17
```

Use the ** operator to calculate powers:

[11]:
```
5 ** 2  # 5 squared
```

[11]: 25

[12]:
```
2 ** 7  # 2 to the power of 7
```

[12]: 128

** has higher precedence than -; if you want a negative base, use parentheses:

[14]:
```
-3**2  # Same as -(3**2)
```

[14]: -9

[15]:
```
(-3)**2
```

[15]: 9

The equal sign (=) assigns a value to a variable:

[16]:
```
width = 20
height = 5 * 90
width * height
```

[16]: 9000

If a variable is not "defined" (assigned a value), using it produces an error:

[18]:
```
n  # Try to access an undefined variable.
```

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

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-18-2d383632fd8e> in <module>
   ----> 1 n  # Try to access an undefined variable.


       NameError: name 'n' is not defined
```

Python provides full support for floating point numbers; operators with mixed type operands convert the integer operand to floating point:

[19]:
```
3 * 3.75 / 1.5
```

[19]: 7.5

[20]:
```
7.0 / 2
```

[20]:
```
3.5
```

In addition to int and float, Python supports other types of numbers, such as Decimal and Fraction. Python also has built-in support for complex numbers, and uses the j or J suffix to indicate the imaginary part (e.g. 3+5j).

3.2 Strings

Besides numbers, Python can also manipulate strings. Strings can enclosed in single quotes ('...') or double quotes ("...") with the same result. Use to escape quotes, that is, to use a quote within the string itself:

[21]:
```
'spam eggs'  # Single quotes.
```

[21]:
```
'spam eggs'
```

[22]:
```
'doesn\'t'  # Use \' to escape the single quote...
```

[22]:
```
"doesn't"
```

[23]:
```
"doesn't"  # ...or use double quotes instead.
```

[23]:
```
"doesn't"
```

[25]:
```
'"Yes," he said.'
```

[25]:
```
'"Yes," he said.'
```

[26]:
```
"\"Yes,\" he said."
```

[26]:
```
'"Yes," he said.'
```

[27]:
```
'"Isn\'t," she said.'
```

[27]:
```
'"Isn\'t," she said.'
```

In the interactive interpreter and Jupyter notebooks, the output string is enclosed in quotes and special characters are escaped with backslashes. Although this output sometimes looks different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise its enclosed in single quotes. The print() function produces a more readable output by omitting the enclosing quotes and by printing escaped and special characters:

[28]:
```
print('"Isn\'t," she said.')
```

```
"Isn't," she said.
```

[29]:
```
s = 'First line.\nSecond line.'  # \n means newline.
s  # Without print(), \n is included in the output.
```

[29]:
```
'First line.\nSecond line.'
```

[30]:
```
print(s)  # With print(), \n produces a new line.
```

```
First line.
Second line.
```

If you don't want escaped characters (prefaced by) to be interpreted as special characters, use raw strings by adding an r before the first quote:

[31]:
```
print('C:\some\name')  # Here \n means newline!
```

```
C:\some
ame
```

[32]:
```
print(r'C:\some\name')  # Note the r before the quote.
```

```
C:\some\name
```

String literals can span multiple lines and are delineated by triple-quotes: """..."""" or ''"..."'. End of lines are automatically included in the string, but it's possible to prevent this by adding a at the end of the line. For example, without a , the following example includes an extra line at the beginning of the output:

[33]:
```
print("""
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Because Python doesn't provide a means for creating multi-line comments, developers often just use triple quotes for this purpose.

[36]:
```
"""
Everything between the first three quotes, including new lines,
is part of the multi-line comment. Technically, the Python interpreter
simply sees the comment as a string, and because it's not otherwise
used in code, the string is ignored. Convenient, eh?
"""
```

[36]:
```
"\nEverything between the first three quotes, including new lines,\nis part of
the multi-line comment. Technically, the Python interpreter\nsimply sees the
comment as a string, and because it's not otherwise\nused in code, the string is
ignored. Convenient, eh?\n"
```

Strings can be concatenated (glued together) with the + operator, and repeated with *:

[37]:
```
# 3 times 'un', followed by 'ium'
3 * 'un' + 'ium'
```

[37]:
```
'unununium'
```

Two or more string literals (that is, the values enclosed in quotes) placed next to each other are automatically concatenated:

[38]:
```
'Py' 'thon'
```

[38]:
```
'Python'
```

Automatic concatenation works only with two literals; it does not work with variables or expressions, so the following code produces an error:

[41]:
```
prefix = 'Py'
prefix 'thon'  # Can't concatenate a variable and a string literal.
```

```
  File "<ipython-input-41-00ad70cd97bc>", line 2
    prefix 'thon'  # Can't concatenate a variable and a string literal.
                ^
SyntaxError: invalid syntax
```

To concatenate variables, or a variable and a literal, use +:

[43]:
```
prefix = 'Py'
prefix + 'thon'
```

[43]:
```
'Python'
```

Automatic concatenation is particularly useful when you want to break up long strings:

[44]:
```
text = ('Put several strings within parentheses '
        'to have them joined together.')
text
```

[44]:
```
'Put several strings within parentheses to have them joined together.'
```

Strings can be indexed (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

[45]:
```
word = 'Python'
word[0]  # Character in position 0.
```

[45]:
```
'P'
```

[46]:
```
word[5]  # Character in position 5.
```

[46]:
```
'n'
```

Indices may also be negative numbers, which means to start counting from the end of the string. Note that because -0 is the same as 0, negative indices start from -1:

[48]:
```
word[-1]  # Last character.
```

[48]:
```
'n'
```

In addition to indexing, which extracts individual characters, Python also supports slicing, which extracts a substring. To slide, you indicate a range in the format start:end, where the start position is included but the end position is excluded:

[49]:
```
word[0:2]  # Characters from position 0 (included) to 2 (excluded).
```

[49]:
```
'Py'
```

If you omit either position, the default start position is 0 and the default end is the length of the string:

[50]:
```
word[:2]   # Character from the beginning to position 2 (excluded).
```

[50]:
```
'Py'
```

[51]:
```
word[4:]  # Characters from position 4 (included) to the end.
```

[51]:
```
'on'
```

[52]:
```
word[-2:] # Characters from the second-last (included) to the end.
```

[52]:
```
'on'
```

This characteristic means that s[:i] + s[i:] is always equal to s:

One way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of n characters has index n. For example:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
```

0 1 2 3 4 5 6

-6 -5 -4 -3 -2 -1

The first row of numbers gives the position of the indices 0…6 in the string; the second row gives the corresponding negative indices. The slice from i to j consists of all characters between the edges labeled i and j, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large results in an error:

[55]:
```
word[42]  # The word only has 6 characters.
```

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

IndexError                                Traceback (most recent call␣
↪last)

<ipython-input-55-e894f93573ea> in <module>
----> 1 word[42]  # The word only has 6 characters.


IndexError: string index out of range
```

However, when used in a range, an index that's too large defaults to the size of the string and does not give an error. This characteristic is useful when you always want to slice at a particular index regardless of the length of a string:

[56]:
```
word[4:42]
```

[56]:
```
'on'
```

[57]:
```
word[42:]
```

[57]:
```
''
```

3.3 Lists

Python knows a number of compound data types, which are used to group together other values. The most versatile is the list, which can be written as a sequence of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

[58]:
```
squares = [1, 4, 9, 16, 25]
squares
```

[58]:
```
[1, 4, 9, 16, 25]
```

[60]:
```
squares[0]  # Indexing returns the item.
```

[60]:
```
1
```

Like strings (and all other built-in sequence types), lists can be indexed and sliced:

[62]:
```
squares[-1]
```

[62]:
```
25
```

[63]:
```
squares[-3:]  # Slicing returns a new list.
```

[63]:
```
[9, 16, 25]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a new (shallow) copy of the list:

[64]:
```
squares[:]
```

[64]:
```
[1, 4, 9, 16, 25]
```

Lists also support concatenation with the + operator:

[65]:
```
squares + [36, 49, 64, 81, 100]
```

[65]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, which means you can change any value in the list:

[66]:
```
cubes = [1, 8, 27, 65, 125]  # Something's wrong here ...
4 ** 3  # the cube of 4 is 64, not 65!
```

[66]:
```
64
```

[67]:
```
cubes[3] = 64  # Replace the wrong value.
cubes
```

[67]:
```
[1, 8, 27, 64, 125]
```

Use the list's append() method to add new items to the end of the list:

[68]:
```
cubes.append(216)  # Add the cube of 6 ...
cubes.append(7 ** 3)  # and the cube of 7.
```

```
cubes
```

[68]:
```
[1, 8, 27, 64, 125, 216, 343]
```

You can even assign to slices, which can change the size of the list or clear it entirely:

[70]:
```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
letters
```

[70]:
```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

[71]:
```
# Replace some values.
letters[2:5] = ['C', 'D', 'E']
letters
```

[71]:
```
['a', 'b', 'C', 'D', 'E', 'f', 'g']
```

[72]:
```
# Now remove them.
letters[2:5] = []
letters
```

[72]:
```
['a', 'b', 'f', 'g']
```

The built-in len() function also applies to lists:

[73]:
```
letters = ['a', 'b', 'c', 'd']
len(letters)
```

[73]:
```
4
```

You can nest lists, which means to create lists that contain other lists. For example:

[75]:
```
a = ['a', 'b', 'c']
n = [1, 2, 3]
x = [a, n]
x
```

[75]:
```
[['a', 'b', 'c'], [1, 2, 3]]
```

[76]:
```
x[0]
```

[76]:
```
['a', 'b', 'c']
```

4 First Steps Towards Programming¶

Python, being used by professional programmers and scientists, among others, is capable of far more complicated tasks than adding numbers, playing with strings, and manipulating lists. For

instance, we can write an initial sub-sequence of the Fibonacci series as follows:

[77]:
```
# Fibonacci series:
# the sum of two elements defines the next.
a, b = 0, 1

while b < 10:
    print(b)
    a, b = b, a+b
```

```
1
1
2
3
5
8
```

This example introduces several new features of the Python language:

- The first line contains a multiple assignment: the variables a and b simultaneously get the new values 0 and 1. On the last line this assignment is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.
- The while loop executes as long as the condition (here: b < 10) remains true. In Python, as in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: < (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to) and != (not equal to).
- Every line in the body of the loop is indented: indentation is Python's way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line. In practice, you typically write Python code in an editor, including a Jupyter notebook, that provides automatic indentation.
- The print() function writes the value of the argument(s) its given. print differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can easily format your output nicely. For example:

[78]:
```
i = 256*256
print('The value of i is', i)
```

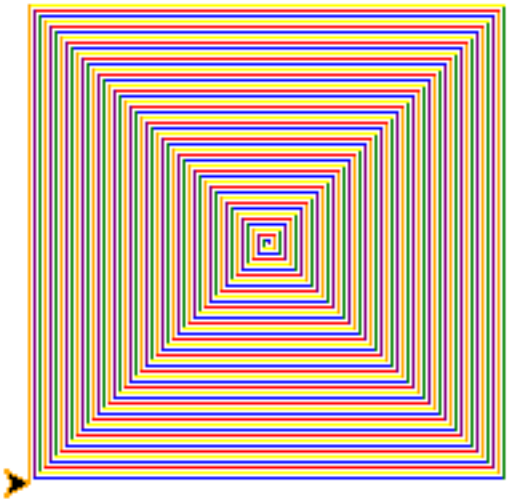
```
The value of i is 65536
```

5 Something Visual

```
import turtle


t = turtle.Pen()
colors = ['red', 'purple', 'blue', 'green', 'yellow', 'orange']

for x in range(180):
    t.pencolor(colors[x%6])
    t.forward(x)
    t.left(90)
```



6 Home Assignment

Continue the tutorial on python.org about More Control Flow Tools