PyLearn_PyTutorial_7_IO

May 2, 2020

#

Learning Python

1 The Python Tutorial -> Input & Output

Link: https://docs.python.org/3/tutorial/inputoutput.html

1.1 Fancier Output Formatting

formatted string literals
- begin a string with `f` or `F` before the opening quotation mark or triple quotation mark.
- can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

[4]:
```
year = 2016
event = 'Referendum'
print(f'Results of the {year} {event}')
```

```
Results of the 2016 Referendum
```

`str.format()`

[5]:
```
yes_votes = 42_572_654
no_votes = 43_132_495
percentage = yes_votes / (yes_votes + no_votes)
print( '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage) )
```

```
 42572654 YES votes  49.67%
```

`str()`, `repr()`

- The `str()` function is meant to return representations of values which are fairly human-readable,
- while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax).
- For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`.

- Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function.
- Strings, in particular, have two distinct representations.

[14]:
```
print(1/7)
print(str(1/7))
print(repr(1/7))
```

```
0.14285714285714285
0.14285714285714285
0.14285714285714285
```

[16]:
```
x = 10 * 3.25
y = 200 * 200
s1 = 'The value of x is ' + str(x) + ', and y is ' + str(y) + '...'
s2 = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
print(s1)
print(s2)
```

```
The value of x is 32.5, and y is 40000…
The value of x is 32.5, and y is 40000…
```

[18]:
```
# The repr() of a string adds string quotes and backslashes:
hello = 'hello, world\n'
hellos = repr(hello)
print(hello)
print(hellos)
```

```
hello, world

'hello, world\n'
```

[21]:
```
x = 10 * 3.25
y = 200 * 200
# The argument to repr() may be any Python object:
print(repr((x, y, ('spam', 'eggs'))))
```

```
(32.5, 40000, ('spam', 'eggs'))
```

1.1.1 Formatted String Literals

Also called **f-strings** for short
An optional format specifier can follow the expression.

[22]:
```
import math
print(f'The value of pi is approximately {math.pi:.3f}.')
```

```
The value of pi is approximately 3.142.
```

[23]:
```python
table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
for name, phone in table.items():
    print(f'{name:10} ==> {phone:10d}')
```

```
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

Other modifiers can be used to convert the value before it is formatted.
`!a` applies **`ascii()`**
`!s` applies **`str()`**
`!r` applies **`repr()`**

For a reference on these format specifications, see the reference guide for the Format Specification Mini-Language: https://docs.python.org/3/library/string.html#formatspec

[25]:
```python
animals = 'eels'
print(f'My hovercraft is full of {animals}.')
print(f'My hovercraft is full of {animals!r}.')
```

```
My hovercraft is full of eels.
My hovercraft is full of 'eels'.
```

1.1.2 The String format() Method

`str.format()` The brackets and characters within them (called **format fields**) are replaced with the objects passed into the `str.format()` method.
A number in the brackets can be used to refer to the position of the object passed into the `str.format()` method.

References:
`str.format()` : https://docs.python.org/3/library/stdtypes.html#str.format
Format String Syntax : https://docs.python.org/3/library/string.html#formatstrings

[30]:
```python
print('We are the {} who say "{}!"'.format('knights', 'Ni'))
```

```
We are the knights who say "Ni!"
```

[31]:
```python
print('{0} and {1}'.format('spam', 'eggs'))
print('{1} and {0}'.format('spam', 'eggs'))
```

```
spam and eggs
eggs and spam
```

If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the argument.

[33]:
```python
print('This {food} is {adjective}.'.format(
      food='spam', adjective='absolutely horrible'))
```

```
print('This {food} is {adjective}.'.format(
     adjective='absolutely horrible', food='spam'))
```

```
This spam is absolutely horrible.
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

[34]:
```
print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
                                                   other='Georg'))
```

```
The story of Bill, Manfred, and Georg.
```

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets `[]` to access the keys

[35]:
```
table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
      'Dcab: {0[Dcab]:d}'.format(table))
```

```
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the table as keyword arguments with the `**` notation.
This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables.

[36]:
```
table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
```

```
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

As an example, the following lines produce a tidily-aligned set of columns giving integers and their squares and cubes

[38]:
```
for x in range(1, 11):
     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
```

```
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

1.1.3 Manual String Formatting

[39]:
```python
for x in range(1, 11):
    print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
    # Note use of 'end' on previous line
    print(repr(x*x*x).rjust(4))
```

```
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

> Note that the one space between each column was added by the way `print()` works: it always adds spaces between its arguments.
>
> The `str.rjust()`, `str.ljust()` and `str.center()` methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged. If weu really want truncation we can always add a slice operation, as in `x.ljust(n)[:n]`.

There is another method, **`str.zfill()`**, which pads a numeric string on the left with zeros. It understands about plus and minus signs:

[40]:
```python
print('12'.zfill(5))

print('-3.14'.zfill(7))

print('3.14159265359'.zfill(5))
```

```
00012
-003.14
3.14159265359
```

1.1.4 Old string formatting

printf-style String Formatting
https://docs.python.org/3/library/stdtypes.html#old-string-formatting

[41]:
```python
import math
print('The value of pi is approximately %5.3f.' % math.pi)
```

```
The value of pi is approximately 3.142.
```

1.2 Reading and Writing Files

The second argument is a string containing the mode.
- **r** : read only - **w** : write only (existing file with the same name will be erased) - **a** : append - **r+** : for read and write. - **b** : binary mode - **t** : text mode

The mode argument is optional; **r** will be assumed if it's omitted.

Open file for **writing** using: `fout = open('testfile.txt', 'w')`
and write to file using **`fout.write(...)`**
`fout.write(...)` returns the number of characters written.

[61]:
```
def testWriteFile1():
    print("ENTER testWriteFile1()...")
    fout = open('testfile.txt', 'w')

    numCharacters = 0
    numCharacters += fout.write("This is first line of text file\n")
    numCharacters += fout.write("This is second line of text file\n")
    print("Number of Characters written = " + str(numCharacters))

    fout.close()
    print(".EXIT testWriteFile1().")

testWriteFile1()
```

```
ENTER testWriteFile1()…
Number of Characters written = 65
.EXIT testWriteFile1().
```

Open file for **reading** using: `fin = open('testfile.txt', 'r')`
and read the **whole file** into string using **`fin.read()`**

[62]:
```
def testReadFile1():
    print("ENTER testReadFile1()...")

    fin = open('testfile.txt', 'r')

    str = fin.read()    # read() without argument, it will read the whole file

    print(str)

    fin.close();
    print(".EXIT testReadFile1().")

testReadFile1()
```

```
ENTER testReadFile1()…
This is first line of text file
This is second line of text file
```

```
.EXIT testReadFile1().
```

Open file for **appending** using: `fout = open('testfile.txt', 'a')`
and write to file using **`fout.write(...)`**

[63]:
```python
def testWriteFile2():
    print("ENTER testWriteFile2()...")

    fout = open('testfile.txt', 'a')

    for i in range(5):
        for j in range(10):
            fout.write( str( (i+1)*(j+1) ) + ' ' )
        fout.write('\n') # there is no such function called writeline()

    fout.close();
    print(".EXIT testWriteFile2().")

testWriteFile2()
```

```
ENTER testWriteFile2()…
.EXIT testWriteFile2().
```

It is good practice to use the **`with`** keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using with is also much shorter than writing equivalent **`try-finally`** blocks.
If you're not using the with keyword, then you should call **`f.close()`** to close the file and immediately free up any system resources used by it

[84]:
```python
def testReadFile2():
    print("ENTER testReadFile2()...")

    with open('testfile.txt') as fin:
        print("reading...")
        read_data = fin.read()
        print("data read => \n" + read_data)

    print("File is closed? => " + str(fin.closed))
    print(".EXIT testReadFile2().")

testReadFile2()
```

```
ENTER testReadFile2()…
reading…
data read =>
This is first line of text file
This is second line of text file
1 2 3 4 5 6 7 8 9 10
```

```
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50

File is closed? => True
.EXIT testReadFile2().
```

While there are still data to be read, read a **chunk of 25 characters** into string using **`fin.read(25)`**

[85]:
```
def testReadFile3():
    print("ENTER testReadFile3()...")

    with open('testfile.txt') as fin:
        while True:
            print("reading...")
            read_data = fin.read(25)
            if not read_data:
                break
            print("data read => \n" + read_data)

    print("File is closed? => " + str(fin.closed))
    print(".EXIT testReadFile3().")

testReadFile3()
```

```
ENTER testReadFile3()…
reading…
data read =>
This is first line of tex
reading…
data read =>
t file
This is second lin
reading…
data read =>
e of text file
1 2 3 4 5
reading…
data read =>
6 7 8 9 10
2 4 6 8 10 12
reading…
data read =>
 14 16 18 20
3 6 9 12 15
reading…
```

```
data read =>
 18 21 24 27 30
4 8 12 1
reading…
data read =>
6 20 24 28 32 36 40
5 10
reading…
data read =>
 15 20 25 30 35 40 45 50
reading…
data read =>


reading…
File is closed? => True
.EXIT testReadFile3().
```

While there are still data to be read, read a **whole line** into string using **`fin.readline()`**
if While there are still data to be read, read a **whole line** into string using **`fin.readline()`** returns an **empty string**, the **end of the file** has been reached,
while **a blank line** is represented by **`\n`**, a string containing only a single newline.

[91]:
```python
def testReadFile4():
    print("ENTER testReadFile4()...")

    with open('testfile.txt') as fin:
        while True:
            print("reading...")
            read_data = fin.readline()
            if not read_data:
                break
            print("data read => \n" + read_data)

    print("File is closed? => " + str(fin.closed))
    print(".EXIT testReadFile4().")

testReadFile4()
```

```
ENTER testReadFile4()…
reading…
data read =>
This is first line of text file

reading…
data read =>
This is second line of text file
```

```
reading…
data read =>
1 2 3 4 5 6 7 8 9 10

reading…
data read =>
2 4 6 8 10 12 14 16 18 20

reading…
data read =>
3 6 9 12 15 18 21 24 27 30

reading…
data read =>
4 8 12 16 20 24 28 32 36 40

reading…
data read =>
5 10 15 20 25 30 35 40 45 50

reading…
File is closed? => True
.EXIT testReadFile4().
```

For **reading line by line** from a file, we can loop over the file object using:

`for read_data in fin:`

This is memory efficient, fast, and leads to simple code.

[93]:
```python
def testReadFile5():
    print("ENTER testReadFile5()...")

    with open('testfile.txt') as fin:
        for read_data in fin:
            print("reading...")
            print("data read => \n" + read_data)

    print("File is closed? => " + str(fin.closed))
    print(".EXIT testReadFile5().")

testReadFile5()
```

```
ENTER testReadFile5()…
reading…
data read =>
This is first line of text file

reading…
```

```
data read =>
This is second line of text file

reading…
data read =>
1 2 3 4 5 6 7 8 9 10

reading…
data read =>
2 4 6 8 10 12 14 16 18 20

reading…
data read =>
3 6 9 12 15 18 21 24 27 30

reading…
data read =>
4 8 12 16 20 24 28 32 36 40

reading…
data read =>
5 10 15 20 25 30 35 40 45 50

File is closed? => True
.EXIT testReadFile5().
```

Read the **all the lines in the whole file** into a **list** using **fin.readlines()**

[99]:
```python
def testReadFile6():
    print("ENTER testReadFile6()...")

    read_data = []
    with open('testfile.txt') as fin:
        print("reading...")
        read_data = fin.readlines()
        print("data read => \n" + str(read_data))

    print("File is closed? => " + str(fin.closed))
    print(".EXIT testReadFile6().")

testReadFile6()
```

```
ENTER testReadFile6()…
reading…
data read =>
['This is first line of text file\n', 'This is second line of text file\n', '1 2
3 4 5 6 7 8 9 10 \n', '2 4 6 8 10 12 14 16 18 20 \n', '3 6 9 12 15 18 21 24 27
30 \n', '4 8 12 16 20 24 28 32 36 40 \n', '5 10 15 20 25 30 35 40 45 50 \n']
```

```
File is closed? => True
.EXIT testReadFile6().
```

Read the **all the lines in the whole file** into a `list` using `list(fin)`

[100]:
```python
def testReadFile7():
    print("ENTER testReadFile7()...")

    read_data = []
    with open('testfile.txt') as fin:
        print("reading...")
        read_data = list(fin)
        print("data read => \n" + str(read_data))

    print("File is closed? => " + str(fin.closed))
    print(".EXIT testReadFile7().")

testReadFile7()
```

```
ENTER testReadFile7()…
reading…
data read =>
['This is first line of text file\n', 'This is second line of text file\n', '1 2
3 4 5 6 7 8 9 10 \n', '2 4 6 8 10 12 14 16 18 20 \n', '3 6 9 12 15 18 21 24 27
30 \n', '4 8 12 16 20 24 28 32 36 40 \n', '5 10 15 20 25 30 35 40 45 50 \n']
File is closed? => True
.EXIT testReadFile7().
```

1.2.1 `f.seek()`, `f.tell()`

- `f.tell()` returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

- To change the file object's position, use `f.seek(offset, whence)`. The position is computed from adding offset to a reference point; the reference point is selected by the whence argument.
  - A whence value of 0 measures from the beginning of the file,
  - 1 uses the current file position, and
  - 2 uses the end of the file as the reference point.

- whence can be omitted and defaults to 0, using the beginning of the file as the reference point.
- In text files (those opened without a `b` in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with `seek(0, 2)`) and the only valid offset values are those returned from the `f.tell()`, or zero. Any other offset value produces undefined behaviour.
- File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

```
[148]: def testReadFile8():
           print("ENTER testReadFile8()...")

           with open('testfile.txt','r') as fin:   # TEXT mode

                               # initially, position is 0
               print("== File Object's position = " + str(fin.tell()) + " ==")
               read_data = fin.read(5)
               print(read_data)

               fin.seek(40)    # seek to offset 40 from beginning of file
               print("== File Object's position = " + str(fin.tell()) + " ==")
               read_data = fin.read(5)
               print(read_data)

               fin.seek(8)     # seek to offset 8 from beginning of file
                               # JUMP BACKWARD IS OKAY
               print("== File Object's position = " + str(fin.tell()) + " ==")
               read_data = fin.read(5)
               print(read_data)

               fin.seek(65, 0) # seek to offset 65 from beginning of file (whence␣
        ↪equals 0)
                               # whence equals 0 is the default
               print("== File Object's position = " + str(fin.tell()) + " ==")
               read_data = fin.read(5)
               print(read_data)

               fin.seek(0,1)   # seek to offset 0 from current position (whence equals␣
        ↪1)
                               # i.e. NOT DOING ANYTHING
               print("== File Object's position = " + str(fin.tell()) + " ==")
               read_data = fin.read(5)
               print(read_data)

               fin.seek(0,2)  # seek to offset 0 before end of file (whence equals 2)
               print("== File Object's position = " + str(fin.tell()) + " ==")
               read_data = fin.read(5)
               print(read_data)

               print("File is closed? => " + str(fin.closed))
           print(".EXIT testReadFile8().")

       testReadFile8()
```

```
ENTER testReadFile8()…
== File Object's position = 0 ==
This
== File Object's position = 40 ==
secon
== File Object's position = 8 ==
first
== File Object's position = 65 ==
1 2 3
== File Object's position = 70 ==
 4 5
== File Object's position = 201 ==

File is closed? => False
.EXIT testReadFile8().
```

Only valid for BINARY MODE

[151]:
```
def testReadFile9():
    print("ENTER testReadFile9()...")

    with open('testfile.txt','rb') as fin:   # BINARY mode

        fin.seek(35)    # seek to offset 35 from beginning of file
        print("== File Object's position = " + str(fin.tell()) + " ==")

        fin.seek(5,1)   # seek to offset 5 from current position (whence equals␣
 ↪1)
        print("== File Object's position = " + str(fin.tell()) + " ==")
        read_data = fin.read(5)
        print(read_data)

        fin.seek(-37,1)   # seek to offset -37 from current position (whence␣
 ↪equals 1)
        print("== File Object's position = " + str(fin.tell()) + " ==")
        read_data = fin.read(5)
        print(read_data)

        fin.seek(-10,2)  # seek to offset -10 before end of file (whence equals␣
 ↪2)
        print("== File Object's position = " + str(fin.tell()) + " ==")
        read_data = fin.read(5)
        print(read_data)

        print("File is closed? => " + str(fin.closed))
    print(".EXIT testReadFile9().")

testReadFile9()
```

```
ENTER testReadFile9()…
```

```
== File Object's position = 35 ==
== File Object's position = 40 ==
b'secon'
== File Object's position = 8 ==
b'first'
== File Object's position = 191 ==
b'40 45'
File is closed? => False
.EXIT testReadFile9().
```

1.2.2 Saving structured data with `json`

- JSON (JavaScript Object Notation)
  https://www.json.org/json-en.html
- Python's standard module
  https://docs.python.org/3/library/json.html#module-json
- `dump()` serializes the object to a text file
- `load()` deserializes object from a text file

[163]:
```python
import json

myPets = ["cat", "dog", "rabbit"]
myData = [1, 3, 6, 8, 7, 4]
x = [myPets, myData]
y = 5, 6, 7, 1
z = x, y
print(z)
print(z[0])
print(z[1])
print(z[0][0][2])
print(z[1][2])

myStr = json.dumps(z)
print(myStr)

with open('testjson.txt', 'w') as fout:
    json.dump(z, fout)

with open('testjson.txt', 'r') as fin:
    read_data = json.load(fin)

print(read_data)
print(read_data[0])
print(read_data[1])
print(read_data[0][0][2])
print(read_data[1][2])
```

```
([['cat', 'dog', 'rabbit'], [1, 3, 6, 8, 7, 4]], (5, 6, 7, 1))
[['cat', 'dog', 'rabbit'], [1, 3, 6, 8, 7, 4]]
(5, 6, 7, 1)
rabbit
7
[[["cat", "dog", "rabbit"], [1, 3, 6, 8, 7, 4]], [5, 6, 7, 1]]
[[['cat', 'dog', 'rabbit'], [1, 3, 6, 8, 7, 4]], [5, 6, 7, 1]]
[['cat', 'dog', 'rabbit'], [1, 3, 6, 8, 7, 4]]
[5, 6, 7, 1]
rabbit
7
```

1.2.3 See also:

- pickle — Python object serialization https://docs.python.org/3/library/pickle.html#module-pickle

Contrary to `JSON`, `pickle` is a protocol which allows the serialization of arbitrarily complex Python objects. As such, it is specific to Python and cannot be used to communicate with applications written in other languages. It is also insecure by default: deserializing pickle data coming from an untrusted source can execute arbitrary code, if the data was crafted by a skilled attacker.

2 END OF The Python Tutorial -> Input & Output