Intermediate Python

Benjamin Short

Agenda

More string formatting

Generators

JSON

File IO

Closures / Decorators

Variable-argument functions

Error handling

Command-line arguments

String Formatting

```
"This is an {} of a {} in Python".format("example", "formatted string")
```

```
'This is an example of a formatted string in Python'
```

Simple, default formatting

```
"This is an {1} {0} {2}".format("example", "indexed", "string")
```

```
'This is an indexed example string'
```

<- Indexed formatting

```
"This {0} repeats the word '{0}' multiple times to demonstrate {1} {0}s".format('string', 'example')
```

```
"This string repeats the word 'string' multiple times to demonstrate example strings"
```

v--Indexed with repeating

```
person = 'John Doe'
age = '24'
"We've found that {person} is {age} years old".format(person=person, age=age)
```

```
"We've found that John Doe is 24 years old"
```

Variable reference formatting

```
d = {'first': 'John', 'last': 'Doe', 'age': 24}
"We've found that {p[first]} {p[last]} is {p[age]} years old".format(p = d)
```

```
"We've found that John Doe is 24 years old"
```

Variable attribute reference formatting

String Formatting - Contd

```
person = {'first': 'John', 'last': 'Doe'}
age = 24
"{p[first]} {p[last]} is {a} years old and has ${0}".format(17, a=age, p=person)
```

```
'John Doe is 24 years old and has $17'
```

Combination Format

```
>>> w1 = "fun"
>>> w2 = "with"
>>> w3 = "spaces"
>>> f"{w1:6} {w2:6} {w3:2}"
'fun    with   spaces'
```

Additional formatting: Column width

```
>>> left = "left"
>>> right = "right"
>>> center= "center"
>>> f"{left:<7} {center:^10} {right:>7}"
'left      center     right'
```

Column width + justification

String Formatting - Contd

```
>>> left = "left"
>>> right = "right"
>>> center= "center"
>>> f"{left:_<7} {center:*^10} {right:_>7}"
'left___ **center** __right'
```

Custom filler character

```
>>> pos = 12345
>>> neg = -12345
>>> print(f"With Signs: '{pos:+}' '{neg:+}'")
With Signs: '+12345' '-12345'
>>> print(f"With Spaces: '{pos: }' '{neg: }'")
With Spaces: ' 12345' '-12345'
>>> print(f"Default: '{pos:-}' '{neg:-}'")
Default: '12345' '-12345'
```

Number sign formatting

```
>>> money = 123.456
>>> per = 86.23456
>>> print(f"We've made ${money:.2f}, which is a profit of {per:.3f}%")
We've made $123.46, which is a profit of 86.235%
```

Precision formatting + type specification

String Format Lab

- Ordered, indexed, named, named-dict formatting
- Specifying column width
- Number formatting

Generators

Gist: Used to “generate” the content on the fly, as opposed to stored all in memory

```
def fib():
    a = 1
    b = 1
    yield a
    yield b
    while True:
        a += b
        yield a
        b += a
        yield b
```

^-- Generator function

```
fib_seq = fib()
tmp = []
for i in range(0, 10):
    tmp.append(next(fib_seq))
print(tmp)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

← Usage of a generator to achieve results

Generator Lab

- Counter
- Fibonacci

JSON

JavaScript Object Notation - very useful for storage or transfer of objects

```
import json
test_arr = [1, 2, 3, 4]
test_dict = {'a': 1, 'b': 2, 'c':3}
saved_dict = {'arr': test_arr, 'dict': test_dict}
json_obj = json.dumps(saved_dict)

print("Python Obj:")
print("{} - {}".format(saved_dict, type(saved_dict)))
print()
print("JSON Obj:")
print("{} - {}".format(json_obj, type(json_obj)))
```

```
Python Obj:
{'arr': [1, 2, 3, 4], 'dict': {'a': 1, 'b': 2, 'c': 3}} - <class 'dict'>

JSON Obj:
{"arr": [1, 2, 3, 4], "dict": {"a": 1, "b": 2, "c": 3}} - <class 'str'>
```

JSON - Contd

```
import json
test_arr = [1, 2, 3, 4]
test_dict = {'a': 1, 'b': 2, 'c':3}
saved_dict = {'arr': test_arr, 'dict': test_dict}
json_obj = json.dumps(saved_dict)

reloaded_dict = json.loads(json_obj)

print("Same") if reloaded_dict == saved_dict else print("Differ")
```

```
Same
```

JSON is useful for the transfer or storage of data, because you can be sure that when you read it again, it is the same data

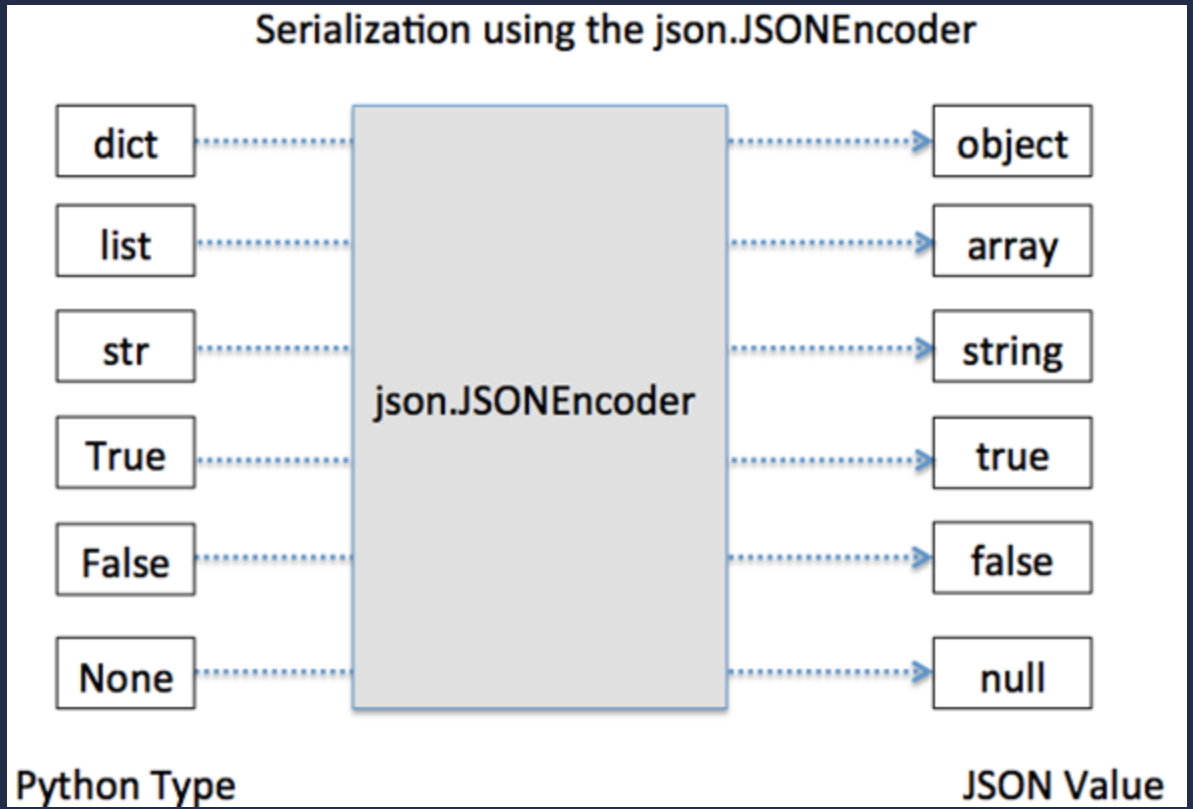
```
json.dumps(set())
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-19941246033c> in <module>
----> 1 json.dumps(set())
```

```
TypeError: Object of type 'set' is not JSON serializable
```

However, not all Python objects can be serialized into a JSON format; you will need to create your own parsers if you need to use custom typoes

JSON - Contd


Serialization using the json.JSONEncoder
dict
list
str
True
False
None
json.JSONEncoder
object
array
string
true
false
null
Python Type
JSON Value

JSON Lab

- Creating JSON strings
- Re-loading JSON strings

File IO

File Input/Output - Otherwise known as “Reading / Writing to files”
Myriad uses - Saving data? Loading configs? Automating inputs?
Also supports various operation types

r - read mode; fails if file does not exist

w - write mode; creates files that don’t exist, empties files that do exist

a - append mode, just adds to existing files, good for logs

[rwa]b - binary mode; basically the same, except it operates on bytes instead of strings

r+ - Read / write mode, opens files if they exist

w+ - Read / write mode, overwrites files if they exist

a[b][+] - Append mode, [binary], [plus read mode]

File IO - Contd

v-- Create a new file and write into it

```
f = open('myfile.txt', 'w')
f.write("A Test String")
#f.flush()
f.close()
```

v-- Create a new binary file and write into it

```
f = open('myfile.txt', 'wb')
f.write(b"A Test String\x01\x02\x03\x04")
f.close()
```

v-- Read data from our file

```
f = open('myfile.txt', 'r')
r = f.read()
print(r)
f.close()
```

```
A Test String
```

v-- Read binary data from our file

```
f = open('myfile.txt', 'rb')
r = f.read()
print(r)
f.close()
```

```
b'A Test String\x01\x02\x03\x04'
```

```
f = open('myfile.txt', 'r')
r = f.read(6)
print(r)
cur = f.tell()
f.seek(0, cur-6)
r2 = f.read(6)
print(r2)
f.close()
```

```
A Test
A Test
```

File Lab

- Create/write to a file
- Read from a file
- Iterate through file lines

Closures

It's a function in a function!
Well yes, but actually no...

Since functions can be declared anywhere, it's only a matter of time before someone tries to make functions within other functions. And as it turns out, this is perfectly legal! These examples show creating and calling functions within other functions:

```
def outerFunc(msg):
    def innerFunc():
        print(msg)
    innerFunc()

outerFunc("Hello Python!")
```

```
Hello Python!
```

This demonstrates a non-returning internal function, which simply prints a message. This also shows how inner functions retain the scope knowledge of their outer shells

This represents an internal function which both transforms data and returns a result. So, since an internal function retains scope knowledge and can return, how can we improve this technique to be more useful...

```
def outerFunc(val):
    def innerFunc():
        return val + 2
    return innerFunc()

res = outerFunc(3)
print(res)
```

```
5
```

Closures - Contd

```
def outerFunc(val):
    def innerFunc():
        return val + 2
    return innerFunc

res = outerFunc(3)
print(res)
```

```
<function outerFunc.<locals>.innerFunc at 0x7f34a0993d90>
```

Well this is interesting, it looks like we can return the actual function itself. But surely once we're out of scope, the internal function will lose hold of the external variable, right?

```
def outerFunc(val):
    def innerFunc():
        return val + 2
    return innerFunc

res = outerFunc(3)
print(res())
```

```
5
```

Turns out, not so! The function remembers the variables it has access to, even indirectly. Implementation details are left as a research exercise for the reader, but now we have the ability to create functions with "primed" variables. Can we take this even further to make something even more useful?

Closures - Contd

```
def outerFunc(val):
    def innerFunc(newVal):
        return val + newVal
    return innerFunc

res = outerFunc(3)
print(res(7))
```

```
10
```

As it turns out, yes we can. The internal function can ALSO accept parameters, allowing us to set some parameters to begin with, then customize the rest of the execution later on.

Closures++ - Decorators

Sometimes, you might want to augment a function; not change how it does its job, but add some extra functionality to it. This can be done with decorators.

```
def decorator(func):
    def wrapper():
        print("I'm before the function!")
        func()
        print("I'm after the function!")
    return wrapper

@decorator
def myFunc():
    print("Here's a function I made myself!")

myFunc()
```

```
C:\Users\Ben\Documents\Python Stuff>python decorator.py
I'm before the function!
Here's a function I made myself!
I'm after the function!
```

Closure Lab

- Logger closure
- Logging decorator

Variable Argument Functions

I mean this in two ways - 1. Functions which accept more than one argument, and 2. Functions which accept named arguments

```
def add(a, b):
    return a + b

print(add(3, 9))
```

```
12
```

```
def div(numerator, denominator):
    return numerator / denominator

print(div(denominator=3, numerator=9))
```

```
3.0
```

```
def div(a, b):
    return a / b

print(div(3, 9))
print(div(9, 3))
```

```
0.3333333333333333
3.0
```

Just like string format options, the function arguments can be combined to produce whatever effects you need. Defaults can also be specified.

```
def exFunc(firstArg, secondArg, thirdArg="Third"):
    print("First argument: {}".format(firstArg))
    print("Second argument: {}".format(secondArg))
    print("Third argument: {}".format(thirdArg))

exFunc(123, secondArg="Seven")
```

```
First argument: 123
Second argument: Seven
Third argument: Third
```

Error Handling

So far, there have been a few instances where doing something wrong has thrown an exception. There are many more ways to throw exceptions, including intentionally, so how can we handle them properly?

```
try:
    print("I execute without incident!")
except:
    print("Uh oh, something went wrong")
```

```
I execute without incident!
```

```
num = "2"
denom = 3
try:
    print("I execute without incident!")
    res = num / denom
    print("{} / {} = {}".format(num, denom, res))
except:
    print("Uh oh, something went wrong")
```

```
I execute without incident!
Uh oh, something went wrong
```

```
num = "2"
denom = 3
try:
    print("I execute without incident!")
    res = num / denom
    print("{} / {} = {}".format(num, denom, res))
except Exception as e:
    print("Uh oh, something went wrong")
    print(e)
finally:
    print("I execute regardless")
```

```
I execute without incident!
Uh oh, something went wrong
unsupported operand type(s) for /: 'str' and 'int'
I execute regardless
```

Functions / Error Handling Lab

- Error handled math

Command-Line Arguments

Programs are most useful when input is varied; this can be dynamically asking for input, but that requires a human to type it in and is slow. Command-line arguments are generally preferred, as these can be supplied by a helper program, removing the need for human interaction when a program needs to be run many times, or just allowing a user to easily re-use certain information easily

```
C:\Users\Ben\Desktop>python python_commandline_args.py 1 + 2
3

C:\Users\Ben\Desktop>python python_commandline_args.py 1 * 2
2

C:\Users\Ben\Desktop>python python_commandline_args.py 1 - 2
-1

C:\Users\Ben\Desktop>python python_commandline_args.py -h
usage: python_commandline_args.py [-h] first operator second

Process some integers.

positional arguments:
  first       The first operand
  operator    The operation to perform
  second      The second operand

optional arguments:
  -h, --help  show this help message and exit
```

```python
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('first', type=int, help='The first operand')
parser.add_argument('operator', help='The operation to perform')
parser.add_argument('second', type=int, help='The second operand')

args = parser.parse_args()
f = args.first
s = args.second
op = args.operator
if op == '+':
    res = f + s
elif op == '*':
    res = f * s
elif op == '/':
    res = f / s
elif op == '-':
    res = f - s

print(res)
```

Command-Line Arguments - Contd

However, this relies on the knowledge of positions; what if you don't want your users to be tied by that knowledge? Enter named arguments; now, with a specified flag, your users can simply enter their data however they like, and your program will properly interpret

```
C:\Users\Ben\Desktop>python python_commandline_args.py -o / -s 2 -f 16
8.0

C:\Users\Ben\Desktop>python python_commandline_args.py -h
usage: python_commandline_args.py [-h] [-f FIRST] [-o OPERATOR] [-s SECOND]

Process some integers.

optional arguments:
  -h, --help            show this help message and exit
  -f FIRST, --first FIRST
                        The first operand
  -o OPERATOR, --operator OPERATOR
                        The operation to perform
  -s SECOND, --second SECOND
                        The second operand
```

```python
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('-f', '--first', type=int, help='The first operand')
parser.add_argument('-o', '--operator', help='The operation to perform')
parser.add_argument('-s', '--second', type=int, help='The second operand')

args = parser.parse_args()
f = args.first
s = args.second
op = args.operator
if op == '+':
    res = f + s
elif op == '*':
    res = f * s
elif op == '/':
    res = f / s
elif op == '-':
    res = f - s

print(res)
```

Command-Line Lab

Questions?