INFO 450 Spring 2021

Week 5

Dicts, Functions, Args and Kwargs

Agenda

- None
- Dicts
- Functions
    - parameters
    - args, **kwargs

None

The sole value of the type NoneType. None is frequently used to represent the absence of a value, as when default arguments are not passed to a function. Assignments to None are illegal and raise a SyntaxError.

https://docs.python.org/3/library/constants.html

```
$ python3
Python 3.8.5 (default, Jul 21 2020, 10:48:26)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> None = 3
  File "<stdin>", line 1
SyntaxError: cannot assign to None
>>> x = None
>>> x
>>> if x == None:
...     logging.debug("Hi")
...
Hi
>>> if x: logging.debug("Hi")
...
>>>
>>> if not x:  logging.debug("Hello")
...
Hello
```

Dictionaries

Python dictionaries are a foundational data type in the language

Dictionaries (dict) can be thought of as a 'map' structure

Dictionaries store other values internally, like a list, but…

Values are assigned to keys within the dictionary.

- Can be used to group data together, such as information about a person

Simple Syntax

```
alien_0 = {"color": "green", "points": 5}
logging.debug("Aliens color is: ", alien_0["color"])
logging.debug("Aliens point value is: ", alien_0["points"])
```

- Dicts are created with {} as the 'constructor'
- Compare to lists, which are notated as: []

Dicts can store 0 or more 'keys' and their values.

- Keys must be 'hashable' - (strings, numbers, tuple *)
- Values can be 'anything'

Creating a dictionary

```
d = {}
d = {"key": "value"}
```

syntax:

Keys are paired to their values with colons.

multiple key/values can be set and separated by commas

```
alien_0 = {"color": "green", "points": 5}
```

Modifying data in a dictionary

How do we set key values in a dict?

```
d = {}
d['age'] = 35
d['colors'] = ['blue', 'green', 'black']
```

How do we remove keys from a dict?

```
>>> d = {"age": 35}
>>> d
{'age': 35}
>>> del d['age']
>>> d
{}
```

Accessing data in a dict

Get a known key (REALLY known, guaranteed, etc. rarely used):

```
>>> d = {"age": 41}
>>> d['age']
41
>>> d['height']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'height'
```

Safely get a key:

```
>>> d = {"age": 41}
>>> d.get('age')
41
>>> d.get('height')
>>> #Nothing above is the same as None
>>> x = d.get('age')
>>> x
41
```

Get a key, but if it doesn't exist, get a default value:

```
>>> x = d.get('height', 99)
>>> x
99
```

Chaining Dictionaries

(not in book that I saw)

Common practice can potentially nest dictionaries. Looking at an example from Spotify API:

https://developer.spotify.com/documentation/web-api/reference/artists/get-artists-albums/

- note: I fudged the structure a little to make it fit my desired example

```
{
  "href": "https://api.spotify.com/v1/artists/1vCWHaC5f2uS3yhpwWbIA6/albums?offset
  "album":
    {
      "album_group": "appears_on",
      "album_type": "album",
      "artist":
        {
          "external_urls": {
            "spotify": "https://open.spotify.com/artist/0LyfQWJT6nXafLPZqxe9Of"
          },
          "href": "https://api.spotify.com/v1/artists/0LyfQWJT6nXafLPZqxe9Of",
          "id": "0LyfQWJT6nXafLPZqxe9Of",
          "name": "Various Artists",
          "type": "artist",
          "uri": "spotify:artist:0LyfQWJT6nXafLPZqxe9Of"
        }
.
.
.

  "previous": null,
  "total": 308
}
```

```
artist_name = spotify_value.get('album', {}).get('artist', {}).get('name')
```

Better than:

```
album = spotify_value.get('album')
if album:
    artist = album.get('artist')
    if artist:
```

Functions/docs for Dictionaries

https://docs.python.org/3/library/stdtypes.html#typesmapping

- list(d) - Returns a list of all keys
- len(d) - Returns the number of items - (CHF) - number of keys
- key in d: Tests whether the key is present in the d
- k not in d: Tests (inverse) of 'key in d'
- iter(d) - Iterator over the keys in the dictionary
- clear(d) - removes all items from the dict
- copy(d) - shallow copy (doesn't copy values)
- items(d) - returns a view of the data in the items, (key, value) - tuples
- keys(d) - returns a list of keys
- values(d) - returns a list of keys
- … and more

Looping on dicts

https://docs.python.org/3/tutorial/datastructures.html#dictionaries

```
>>> product = {"product_id":"14322", "name": "Big 65 inch TV", "price": 499.99}
>>> for key, value in product.items():
...     logging.debug(f"{key}: {value}")
...
product_id: 14322
name: Big 65 inch TV
price: 499.99
>>> for key in product.keys():
...     logging.debug(f"{key}: {product[key]}")
...
product_id: 14322
name: Big 65 inch TV
price: 499.99

>>> for key in product:
...     logging.debug(f"{key}: {product[key]}")
...
product_id: 14322
name: Big 65 inch TV
price: 499.99
```

Generating a unique list of values

Wrap the 'values' with a 'set' command, to convert the 'list' datatype to a 'set'

Set: A set object is an unordered collection of distinct hashable objects.

```
>>> favorite_languages = {"jen": "python", "sarah": "c", "edward":"ruby", "phil":
>>> for language in favorite_languages.values():
...     logging.debug(f"This is a favorite language: {language}")
...
This is a favorite language: python
This is a favorite language: c
This is a favorite language: ruby
This is a favorite language: python
>>> for language in set(favorite_languages.values()):
...     logging.debug(f"This is a favorite language: {language}")
...
This is a favorite language: ruby
This is a favorite language: python
This is a favorite language: c
```

TO READ:

Read the end of Chapter 6.

Using dicts in a list, using lists in a dict

```
aliens = []
aliens.append({"color":"green", "points": 5})
aliens.append({"color":"red", "points": 10})
aliens.append({"color":"yellow", "points": 15})
```

```
person = {}
person['name'] = 'Chris'
person['favorite_colors'] = ["purple", "gold"]
```

Functions

Let's deep dive into functions so we can understand the homework assignments and online practices.

- A function is an executable statement.
- Its execution binds the function name in the current local namespace to a

Defining a function

- The keyword `def` introduces a function definition.
- It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function’s documentation string, or docstring. (More about docstrings can be found in the section Documentation Strings.)

- There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it’s good practice to include docstrings in code that you write, so make a habit of it.

Function to print a help menu

help.py

```
def help():
    print("To use this program, execute it with a python3.7+ interpreter")

if __name__ == "__main__":
    help()
    h = help
    h()
```

```
$ python3 help.py
To use this program, execute it with a python3.7+ interpreter
To use this program, execute it with a python3.7+ interpreter
```

Yes! You can assign a function 'name' to a variable if you want.

Function with a parameter

- Required parameters are listed in the paranthesis.
- Multiple parameters must be comma separated.

```
import logging
# create logger
logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.DEBUG)

def fib(n):
    first = 0
    second = 1
    for x in range(n):
        logging.debug(first)
        first, second = second, first+second

if __name__ == "__main__":
    logging.debug("Calling fib(3)")
    fib(3)
    logging.debug("Calling fib(8)")
    fib(8)
```

Output of fib

```
$ python3 fib.py
DEBUG: Calling fib(3)
0
1
1
DEBUG: Calling fib(8)
0
1
1
2
3
5
8
13
```

Multiple parameters

Comma separated to allow multiple values to be passed into a function

add_them.py

```
import logging
# create logger
logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.DEBUG)

def add_them(first, second):
    return first + second

if __name__ == "__main__":
    logging.debug("add_them(1, 2)")
    logging.debug(add_them(1, 2))
    logging.debug("add_them(3, 4)")
    logging.debug(add_them(3, 4))
    logging.debug("add_them(\"dog\", \"type\")")
    logging.debug(add_them("dog", "type"))
```

```
$ python3 add_them.py
DEBUG: add_them(1, 2)
DEBUG: 3
DEBUG: add_them(3, 4)
DEBUG: 7
DEBUG: add_them("dog", "type")
DEBUG: dogtype
```

Optional Parameters

hello_world.py

```
import logging
# create logger
logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.DEBUG)

def hello_world(name=None):
    if not name:
        logging.debug("Hello, World!")
    else:
        logging.debug(f"Hello, {name}!")

if __name__ == "__main__":
    logging.debug("No parameters.")
    hello_world()
    logging.debug("Explicit None")
    hello_world(None)
    logging.debug("Chris")
    hello_world('Chris')
    logging.debug("name=\"Mom\"")
    hello_world(name='Mom')
```

```
$ python3 hello_world.py
DEBUG: No parameters.
Hello, World!
DEBUG: Explicit None
Hello, World!
DEBUG: Chris
```

Mix and Match

parrot.py

```python
import logging
# create logger

def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    logging.debug("-- This parrot wouldn't", action, end=' ')
    logging.debug("if you put", voltage, "volts through it.")
    logging.debug("-- Lovely plumage, the", type)
    logging.debug("-- It's", state, "!")

if __name__ == "__main__":
    parrot(1000)                                          # 1 positional argument
    parrot(voltage=1000)                                  # 1 keyword argument
    parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
    parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
    parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
    parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keywor

    #The following fail
    # parrot()                     # required argument missing
    # parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
    # parrot(110, voltage=220)     # duplicate value for the same argument
    # parrot(actor='John Cleese')  # unknown keyword argument
```

Execute parrot.py

```
$ python3 parrot.py
-- This parrot wouldn't voom if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't voom if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't VOOOOOM if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't VOOOOOM if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't jump if you put a million volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's bereft of life !
-- This parrot wouldn't voom if you put a thousand volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's pushing up the daisies !
```

Returning values from a function

add_them.py

```
import logging
# create logger
logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.DEBUG)

def add_them(first, second):
    return first + second

if __name__ == "__main__":
    logging.debug("add_them(1, 2)")
    logging.debug(add_them(1, 2))
    logging.debug("add_them(3, 4)")
    logging.debug(add_them(3, 4))
    logging.debug("add_them(\"dog\", \"type\")")
    logging.debug(add_them("dog", "type"))
```

```
$ python3 add_them.py
DEBUG: add_them(1, 2)
DEBUG: 3
DEBUG: add_them(3, 4)
DEBUG: 7
DEBUG: add_them("dog", "type")
DEBUG: dogtype
```

- Return values can be of any type
- Return values can be None

Empty return statements are essentially None

return_none.py

```
import logging
# create logger
logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.DEBUG)

def return_none():
    return None

def empty_return():
    return

if __name__ == "__main__":
    a = return_none()
    logging.debug(a)
    b = empty_return()
    logging.debug(b)
```

```
$ python3 return_none.py
DEBUG: None
DEBUG: None
```

Returning multiple values

multiple_return.py

```
import logging
# create logger
logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.DEBUG)

def return_multiple():
    return 1, 2, 3

if __name__ == "__main__":
    logging.debug("All: ")
    a, b, c = return_multiple()
    logging.debug(a)
    logging.debug(b)
    logging.debug(c)

    logging.debug("only first")
    x, _, _ = return_multiple()
    logging.debug(x)

    logging.debug("what it does")
    d = return_multiple()
    logging.debug(d)
```

```
$ python3 multiple_return.py
DEBUG: All:
DEBUG: 1
DEBUG: 2
```

Errors and Exceptions

- Syntax Errors
- Exceptions

Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>>
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected.

The error is caused by (or at least detected at) the token preceding the arrow: in the example, the error is detected at the function print(), since a colon (':') is missing before it.

File name and line number are printed so you know where to look in case the input came from a script.

Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.

Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs.

Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Handling Exceptions

It is possible to write programs that handle selected exceptions.

Look at the following example, which asks the user for input until a valid integer has been entered.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

fake_number.py

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")
```

```
$ python fake_number.py
Please enter a number: asdf
Oops!  That was no valid number.  Try again...
Please enter a number: 333
```

```
$ python fake_number.py
Please enter a number: alskdjf
Oops!  That was no valid number.  Try again...
Please enter a number: fdslakj
Oops!  That was no valid number.  Try again...
Please enter a number: ^CTraceback (most recent call last):
  File "fake_number.py", line 3, in <module>
    x = int(input("Please enter a number: "))
KeyboardInterrupt
```

- The user to interrupt the program (using Control-C or whatever the operating system supports); note that a user-generated interruption is signalled by raising the KeyboardInterrupt exception.

Can handle multiple Exception types

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Handle 'other' Exceptions

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

Exception information

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Clean up actions

The try statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances.

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

Command Line Arguments

Common utility scripts often need to process command line arguments.

These arguments are stored in the sys module’s argv attribute as a list.

For instance the following output results from running

```
$ python3 command_line.py one two three
```

command_line.py

```python
import sys
print(sys.argv)
```

at the command line:

```
$ python3 command_line.py one two three
['command_line.py', 'one', 'two', 'three']
```

`sys.argv` is a `list` representing the command line.

`sys.argv[0]` == python file name

`sys.argv[1+]` == optional, and other parameters

Homework

Same as last week, but a new project!

https://github.com/vcu-chfauerbach/week5homework

Due March 8st, by 11:59:59 PM

Same rules apply. Make it green!