Basic - Python

EECS 201 Fall 2024

Submission Instructions

This assignment will be submitted as a repository on the EECS GitLab server (see Basic - Git 1). Create a private, blank, **README-less (uncheck that box!)** Project on it with the name/path/URL `eecs201-basic-python` and add `brng` as a Reporter. The submission branch will be `rel`. If this branch is not already the default initial branch, you initialize the local repo with an additional argument: `git init --initial-branch=rel` or `git init -b rel` if your version of Git is recent enough. Otherwise you can create a branch with this name after your first commit.

To finish the submission process and get your grade, you will need to run the autograder. You will need to SSH into the course server (see Basic - Intro):
`local$ ssh uniqname@peritia.eecs.umich.edu`
(the `local$` referring to a shell prompt on your system)

and then run `eecs201-test`:
`peritia$ eecs201-test basic-python`
(the `peritia$` referring to a shell prompt on the server)

Preface

For this assignment you're going to need **Python 3**. The package name for it on Ubuntu is `python3` (the "normal" `python` package might be for Python 2.7, depending on your version of Ubuntu). Otherwise, there's various ways you can install it: look it up for your given platform if you aren't using WSL+Ubuntu. Go ahead and install it on your system if you wish to do this assignment locally.

In this assignment you'll be provided yet another zipped archive containing some starter files.

`https://www.eecs.umich.edu/courses/eecs201/fa2024/files/assignments/basic-python.tar.gz`

Note that this assignment is to be submitted on a remote branch called `rel`. Initialize a Git repository **inside of the extracted** `basic-python` **directory**; as noted above `git init -b <branch name>` or `git init --initial-branch <branch name>` can initialize the repo with a different branch name (e.g. `rel` as per the submission instructions). If your version of Git is too old for these options, you could create the `rel` branch afterwards after your first commit, or you can set the local branch's tracking information manually. Create a file called `report.txt` in this directory.

Add all of the present files and commit them.

Create a **private** project named `eecs201-basic-python` on the EECS GitLab (`gitlab.eecs.umich.edu`) and add the instructor `brng` as a **Reporter**. Set this UMich GitLab project as your remote: you'll be pushing to it in order to submit.

Ramping up

In this section you'll be getting familiar with forming Python expressions. This section has no submission component and is purely for exploring the syntax. If you are already familiar with Python, you may skip this section.

1. First we're going to work in Python's interactive shell. Run `$ python3` (or `$ python` if your system's default Python is Python 3).
2. In the interactive shell, much like MATLAB if you have used that, the value of the result of an entered expression will be printed out (remember from lecture that expressions can be statements!).
3. `>>>` will be used to indicate Python interactive shell prompts.
4. Run `>>> 6 * 7`. This will multiply 6 with 7.
5. Run `>>> "Hello " + "world!"`. This will produce a new `str` that concatenates the two.
6. Run `>>> len("Hello")`. This is an example of the built-in function `len()` being used on a `str`, which is a sequence.
7. Messing around with types is cool, but let's now save the values we create.
8. Run `>>> x = 6 * 7`. We are now binding variable/identifier `x` to an `int` object of value 42 (i.e. `x` equals 42).
9. We can print out an object as well (provided it has implemented a `__str__` function, which the built-in types have). Run `>>> print(x)`.
10. We can get information about the object `x` is bound to. Run `>>> type(x)`: this gets type of the object bound to `x`.
11. Run `>>> id(x)`: this gets the ID of the object bound to `x`.
12. Let's use `x` in an expression. Run `>>> y = x - 42`. Print out `y`.
13. Run `>>> z = "Hello world!"`
14. Run `>>> print(z)`. Here's our obligatory "Hello world!"
15. Let's now play with another built-in type: the list. Run `>>> l = [x, y, z]`. This will construct a list using the objects bound by x, y, and z. Note how we can stuff arbitrary objects into a list!
16. Run `>>> print(l)`. As mentioned before, the built-in types can be nicely printed.
17. A list can be indexed by an integer position: run `>>> print(l[0])`
18. Let's check if the objects referred to by `x` and `l[0]` are the same: run `>>> x is l[0]`. This performs an ID check between the two objects referred to by `x` and `l[0]` and returns a boolean (alternatively, `>>> id(x) == id(l[0])` works as well).
19. Try verifying that `y` and `z` are the same as `l[1]` and `l[2]`.
20. Let's manipulate this list (lists are mutable after all). Run `>>> l.append(3.14)`. Try printing out `l` again. With `append()` we have added a `float` to list `l`.
21. Lastly, let's play with dictionaries. Initialize an empty dictionary with `>>> dictionary = {}`.
22. Let's set the key-value pairs "foo"=0 and "bar"=1:
    `>>> dictionary['foo']=0`
    `>>> dictionary['bar']=1`
23. Let's retrieve the values associated with the keys:
    `>>> print(dictionary['foo']`
    `>>> print(dictionary['bar']`
24. To quit the interactive shell, run `>>> quit()` or, like in most other shells, send an EOF (end-of-file) by hitting Ctrl-D (`^D`)

If you had 0 experience with Python going in, hopefully this small intro put you more at ease: Python has a fairly simple syntax, free from a lot of cruft. While it can serve as a scripting language, it definitely is a lot more familiar and intuitive compared to something like Bash.

1 Jotting things down

Much like Bash, we can put statements in at the interactive shell or we can put down our statements in a script to be run at our convenience.

1. Create a file named `loops.py`
2. If you want to, put: `#!/usr/bin/env python3`. This shebang will use the `env` executable to find `python3` and use that as the interpreter if you decide to execute the file directly after `chmod`ing it. I'm going to explicitly run these scripts under `python3`, so you don't actually have to do this step.
3. In the script, `import` the `sys` module. This module provides info about the current interpreter, such as the command-line arguments.
4. `sys.argv` is a list of command-line arguments provided to the script. Like many programming languages, the list contains the script file's name as the 0th argument.
5. Add a check to see that at least one additional command-line argument was added. Note that you can call `len()` on a list to get its length. Note that the 0th argument counts towards this length: `$ python3 loops.py` or `$ ./loops.py` will both have an argument list of `['./loops.py']` and a length of 1.

   If there isn't, exit with a status code of 1 via `exit()` or `sys.exit()`.
6. The 1st argument will serve as an integer: assign a variable to be that integer value. The built-in `int()` function is able to perform conversions of various types to integer.
7. In the script, add a `while`-loop to `print()` out the integers from 1 to the script's command-line argument.
8. Add a `for`-loop with `range()` to `print()` out the integers from 1 to the script's command-line argument.
9. Try running `$ python3 loops.py 10`. As a result of both loops, you should see 1 to 10 be printed out with each number on its own line, then 1 to 10 printed out again.

2 "Fun" is in "function"

We'll now play around with functions. Much like in Bash, functions can be declared and then called, albeit the calling syntax is a lot more familiar. For you folk who are used to Python, feel free to take full advantage of some of the more Pythonic expressions (like comprehensions) for these ;)

1. Create a file named `functions.py`. This file will contain some fucntions that can be called from elsewhere.
2. Create and implement a function called `lower_list` that takes in a string (`str`) a returns a list of the string's characters that are lowercase, in the order that they were in from the string (and with duplicates if the same letter comes up multiple times). For example, `lower_list("HelLo")` returns `['e', 'l', 'o']`. Note that there is a `str` function called `islower()`.
3. Create and implement a function called `list_to_string_dict` that takes in a `list` of `tuples` representing key-value pairs where the 0th element is a key and the 1st element is a value (beyond being an immutable sequence, tuples, like in some other languages, are often used to act as an object with unnamed variables). The function returns a `dict` made up of the key-value pairs in the input list that have **keys that are strings**. For example, `list_to_string_dict([('hello', 1), ('world', 42), (1234, 5)])` would return a `dict` that is representable by `{'hello':1, 'world':42}`. You can handle this by getting the `type()` of that object and checking if it `is` a `str`.
4. You can use the provided `test-functions.py` file to test out your functions (you'll have to write the code to do this; the file only contains some boilerplate). Note that with the depending on the way the file is imported, you may have to navigate the namespace e.g. `functions.lower_list`. The way `test-functions.py` imports it will allow you to just call `lower_list` and `list_to_string_dict`. You can also try them out in the interpreter shell by importing it :)

5. Since this file is meant to contain functions that will be called elsewhere (e.g. be imported), it's kind of poor form to simply add statements that call the functions at the bottom of the file: when files are imported, each line is run, and we wouldn't want to call these functions when someone else wants to import this file.

6. Another option can be to create an if-block in `functions.py` that says `if __name__ == '__main__':` and put your testing statements in there. How this works is when you run a script as the main program (e.g. via the command line), a special variable called `__name__` is assigned to the string `'__main__'`, allowing you to differentiate between when this script is being run from the command line or is being imported. There's some neat use cases of this: for instance you could write a script that's mostly meant to be run as a program, but offers an API (application-programming-interface) for more advanced users to take advantage of, or you could write a library intended to be imported but have the script run some test cases or a reference program when run.

3 Classes

Python also is object-oriented: so much so that everything is an object. We can define our own classes as well!

1. Create a file named `classes.py`. This file will contain some classes that will be usable from elsewhere.
2. Define a `Student` class whose instances are composed of the member variables `name` (`str`) and `grade` (`float`). The `__init__()` function should, after the object instance argument (e.g. `self`), have the arguments `name` and `grade` in that order.
3. Add a method to `Student` called `passing()` that returns `True` if their `grade` is greater than or equal to `70.0` and `False` if their grade is less than `70.0`.
4. There are also methods that you can define for your class that'll interface with some built-in functions like `str()`. For `str()` to work on your class, you'll have to implement the `__str__()` function (a so-called "dunder method"). Implement it so that when you call `str()` on a `Student`, it will return `<Name> - <Grade>` e.g. `Bob - 84`. Mind the space around the `-`! There's multiple ways to do this, such as appending strings together or using the `format()` function. Try running `print()` on a `Student` now :)

   On a sidenote, there's also `__repr__()` for "formal" string representations of a class. This StackOverflow post has a nice summary.
5. You can use the provided `test-classes.py` file to test out your functions. You can also try them out in the interpreter shell or via the `if __name__` from before.

4 The filesystem

Things aren't as interesting when we can't interact with anything.

1. Create a file named `files.py`.
2. In this script open the file named `data.txt` for reading. Try using the `with...as...` mechanism. If you go for the plain-old `open()...close()` paradigm, don't forget to close the file at the end.
3. Read and print out each line in the file. The `readline()` and `readlines()` functions may be helpful. Also note that you can iterate over a file with a `for`-loop.
4. Make sure that you don't double-print newlines: the `strip()` function for `str` objects that gets rid of whitespace at the beginning and end of a string may help out here.

5 Some extras

This section will be more for exploring things. No submission for these, but they're some things to keep in mind.

Running other commands

There's two relatively easy ways to run some other program or command from Python. From the `os` module we have `os.system()`, which allows you to shell commands (i.e. the C `system()` function). For example: `os.system("ls -ltr somedir")`. The downside is that there isn't a way to capture the output with this.

The `subprocess` module, however, has a rich set of utilities for this purpose. The `subprocess.run()` function is a high level function that allows you to run commands, capture their outputs, and check their statuses. For example: `subprocess.run(['ls', '-ltr', 'somedir'], capture_output=True)` will run the command specified by a list of arguments as well as capturing the output for the returned object. The object it returns is a `CompletedObject` which has a `stdout` attribute that is a `bytes` containing the output of the command. This `bytes` can then be encoded (e.g. via UTF-8) to give you a `str`.
Putting it together:
`subprocess.run(['ls', '-ltr', 'somedir'], capture_output=True).stdout.encode('utf-8')`

Alternatively, you can provide an encoding to `subprocess.run`. Try looking at the documentation for more details :)

With the power of being able to call other utilities on your system you might find Python a really neat way to automate lots of stuff :)

Being Pythonic with comprehensions

A word that comes up often is "Pythonic": this refers to the taking advantage of features and things built into Python that help clean up and express ideas that would otherwise be messier in other languages due to some quirk of syntax. For instance, you may be used to using an iterating index variable in C/C++ to index into an array, but the "Pythonic" way would be to just directly iterate over it. Sure, you could do `for i in range(len(somelist)):`, but unless you absolutely need an integer index for something, why not express it as `for item in somelist:`?

One of the neat features of Python that we brought up in lecture are "comprehensions". They allow you to easily build things like lists, sets, and dictionaries with iterable objects. For instance, if I wanted to get a list of the grades squared from a list of `Student`s named `students`:

```
grades = [student.grade ** 2 for student in students]
```

or maybe you want the names of students who are passing:

```
passing_students = [student.name for student in students if student.passing()]
```

Dictionaries have a similar syntax. Perhaps you have a dictionary but want to keep only pairs whose keys are strings:

```
only_strings = {k:v for k,v in somedict.items() if type(k) is str}
```

Try tackling #2 using comprehensions!

6 Conclusion

1. Add and commit the files you created.
2. Fill out the `report.txt` file in the following steps:
3. On the first line provide an integer time in minutes of how long it took for you to complete this assignment. It should just be an integer: no letters or words.
4. On the second line and beyond, write down what you learned while doing this assignment. If you already knew how to do all of this, put down "N/A".
5. Commit your `report.txt` file and push your commits to your remote.
6. Try running `eecs201-test` on this assignment on the course server! If you don't remember, check out the last part of Basic - Git.