Adding two numbers

Write a Python program called `add.py` that takes two integer values provided as positional command-line arguments and shows the result of adding them:

```
$ ./add.py 1 2
1 + 2 = 3
```

The program should present a usage statement if run with the `-h` or `--help` flags:

```
$ ./add.py -h
usage: add.py [-h] int int

Add two integer values

positional arguments:
  int         Two numbers to add

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

The program should print a brief usage and error message if it does not receive exactly two integer arguments. For instance, if the program is run with no arguments:

```
$ ./add.py
usage: add.py [-h] int int
add.py: error: the following arguments are required: int
```

If run with only one argument, it should likewise complain:

```
$ ./add.py 1
usage: add.py [-h] int int
add.py: error: the following arguments are required: int
```

If any of the arguments is not a valid integer value, the program should print an error and usage:

```
$ ./add.py 1.2 3.4
usage: add.py [-h] int int
add.py: error: argument int: invalid int value: '1.2'
```

Understanding Test-Driven Development

The idea of Test-Driven Development (TDD) [1] can be summarized as follows:

1. Add a test
2. Run all tests and see if the new test fails
3. Write the code
4. Run tests
5. Refactor code
6. Repeat

The idea is that we should first imagine what our program should or should not do under certain conditions. We **first** write a test that verifies these assumptions. The program is expected to fail the test *because we haven't yet written the code to satify the test*. So we write the code until it finally passes our new test. We also run all the other test to ensure we haven't broken something that worked before.

Installing and using `pytest`

In the spirit of TDD, we will **first** run the tests found in `test.py` using the `pytest` module, which you may need to install like so:

```
$ python3 -m pip install pytest
```

The `pytest` module will run all the functions with names starting with `test_` that are found in the provided `test.py`. The `test_` functions are run in the order in which they are declared in the file, and I usually order them in a way that leads the student through a logical progression. For instance, we first have to have a program with a given name, that program should be runnable, and it should produce help documentation, etc.

I have written several tests that will run the expected `add.py` program to ensure it meets the program specifications:

1. The program accepts only two valid integer values.
2. The program prints helpful "usage" statements upon request and errors.
3. The program correctly adds the two values and displays the sum.

The first test checks if there is a file in this directory called `add.py`. If we peek at the `test.py`, the relevant code looks like this:

```
prg = './add.py'   ①

def test_exists(): ②
    """exists"""
    assert os.path.isfile(prg) ③
```

① Since I will reference the name of the program several times, I put this into a `prg` variable.

② Only functions with a name starting with `test_` will be run by `pytest`. You can add other helper functions that will be ignored.

③ The `assert` statement will throw an Exception if the argument does not evaluate as "truthy" [2]. Here we use the `os.path.isfile()` function to return `True` or `False` depeninding on whether there is a file called `./add.py`.

We can run `pytest` on the `test.py` file to execute this function. You can run `pytest --help` to learn about the various options, but I like to run with the following flags:

- `-x`|`--exitfirst`: exit instantly on first error or failed test
- `-v`|`--verbose`: increase verbosity

The `-x` flag prevents the entire test suite running when it encounters an error. This is especially important for beginners as we should not overwhelm them with an outpouring of failing tests. If the first test fails — that there is no program to check — there's no point in running any further tests:

```
$ pytest -xv test.py
```

NOTE	I have provided a `Makefile` that has a `test` target so you can run `make test` as a shortcut.

However you run the `pytest` command, you should see that the first test fails showing that the `add.py` program does not exist:

```
$ make
pytest -xv test.py
============================= test session starts ==============================
...

test.py::test_exists FAILED                                              [ 20%]

=================================== FAILURES ===================================
_________________________________ test_exists __________________________________

    def test_exists():
        """exists"""

>       assert os.path.isfile(prg)
E       AssertionError: assert False
E        +  where False = <function isfile at 0x10b6f9310>('./add.py')
E        +    where <function isfile at 0x10b6f9310> = <module 'posixpath' from
'/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/posixpath.py'>.isfile
E        +      where <module 'posixpath' from
'/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/posixpath.py'> =
os.path

test.py:17: AssertionError
!!!!!!!!!!!!!!!!!!!!!!!!! stopping after 1 failures !!!!!!!!!!!!!!!!!!!!!!!!!
============================== 1 failed in 0.04s ===============================
```

It's a bit to decipher the test output, but part of what we're teaching is how to read error messages!

Getting started

In order to pass the first test, we need to create our program. This will actually suffice:

```
$ touch add.py
```

The `touch` command will create a new, empty file called `add.py`. Since the `test_exists()` function does nothing more than check for the existence of the file called `./add.py`, this will cause the test suite to now pass the first test and fail on the second:

```
$ make test
pytest -xv test.py
============================= test session starts ==============================
...

test.py::test_exists PASSED                                              [ 20%] ①
test.py::test_usage FAILED                                               [ 40%] ②

=================================== FAILURES ===================================
_________________________________ test_usage __________________________________

    def test_usage():
        """usage"""

        for flag in ['-h', '--help']:
            rv, out = getstatusoutput(f'{prg} {flag}')
>           assert rv == 0
E           assert 126 == 0
E             -126
E             +0

test.py:26: AssertionError
!!!!!!!!!!!!!!!!!!!!!!!! stopping after 1 failures !!!!!!!!!!!!!!!!!!!!!!!!
========================== 1 failed, 1 passed in 0.04s ==========================
```

① The `test_exists` now passes!

② The `test_usage` is failing.

We are using the `subprocess.getstatusoutput()` function to run `./add.py -h` and `./add.py --help` and checking that the return value (`rv`) and output (`out`) from the program match expected values. Here we expect the return value to be `0` which would indicate that it ran and exited successfully, but we are getting something that is not `0`.

If we attempt to run the program manually like this, we'll see the problem:

```
$ ./add.py -h
-bash: ./add.py: Permission denied
```

Creating a program with `new.py`

So we need to create a valid Python program that can be executed like `./add.py` and will print a "usage" statement when run either with `-h` or `--help`. I would suggest you try using the `new.py` program to do this:

```
$ ./new.py add.py
Done, see new script "add.py."
```

The resulting program will use the `argparse` module to accept and validate the arguments to your program. If you run `./add.py -h`, you should see something like this:

```
$ ./add.py -h
usage: add.py [-h] [-a str] [-i int] [-f FILE] [-o] str

Rock the Casbah

positional arguments:
  str                   A positional argument

optional arguments:
  -h, --help            show this help message and exit
  -a str, --arg str     A named string argument (default: )
  -i int, --int int     A named integer argument (default: 0)
  -f FILE, --file FILE  A readable file (default: None)
  -o, --on              A boolean flag (default: False)
```

These are not the correct arguments for our program, just an example of the kinds of arguments that `argparse` can handle. Open the `add.py` with your editor and change the `get_args()` function to match this:

```
def get_args():
    """get args"""

    parser = argparse.ArgumentParser(
        description='Add two integer values', ①
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('numbers',     ②
                        metavar='int', ③
                        nargs=2,       ④
                        type=int,      ⑤
                        help='Two numbers to add') ⑥

    return parser.parse_args()
```

① Add an informative description for the program.

② Define a *positional* argument called "numbers".

③ The `metavar` value will show in the usage as a placeholder for the argument.

④ We want exactly 2 positional arguments.

⑤ Each argument must be parsable as an `int`.

⑥ This is the longer description for the help.

Change the `main()` to this:

```
def main():
    args = get_args()
    print(args.numbers)
```

If you run your program now with no arguments, it should print a brief usage:

```
$ ./add.py
usage: add.py [-h] int int
add.py: error: the following arguments are required: int
```

And if you run with two integers, it should look like this:

```
$ ./add.py 1 2
[1, 2]
```

Refactoring with the tests

Now that we have a program that seems to work, it's time to run the tests!

```
$ make test
pytest -xv test.py
============================= test session starts ==============================
...

test.py::test_exists PASSED                                              [ 20%]
test.py::test_usage PASSED                                               [ 40%]
test.py::test_wrong_number_args PASSED                                   [ 60%]
test.py::test_not_numbers PASSED                                         [ 80%] ①
test.py::test_valid_input FAILED                                         [100%] ②

=================================== FAILURES ===================================
_______________________________ test_valid_input _______________________________

    def test_valid_input():
        """test with valid input"""

        for x, y, z in [[0, 0, 0], [1, 0, 1], [1, 2, 3], [2, 1, 3]]:
            rv, out = getstatusoutput(f'{prg} {x} {y}')
            assert rv == 0
>           assert out.rstrip() == f'{x} + {y} = {z}'
E           AssertionError: assert '[0, 0]' == '0 + 0 = 0' ③
E             - [0, 0]
E             + 0 + 0 = 0

test.py:63: AssertionError
!!!!!!!!!!!!!!!!!!!!!!!! stopping after 1 failures !!!!!!!!!!!!!!!!!!!!!!!!
========================= 1 failed, 4 passed in 0.43s ==========================
```

① All these tests are passing now! I always check with bad inputs, so these tests pass in the wrong number and type of arguments to ensure the program fails.

② When we finally run the program with valid inputs, it fails to produce the correct output.

③ Our program produced the string `'[0, 0]'` but the test expected to see `'0 + 0 = 0'`.

We actually ended up passing several tests by virtue of using `argparse` to define the program's parameters. If we look at the function called `test_wrong_number_args` in `test.py`, we see that it runs our program with 0, 1, and 3 integer values to ensure that all fail:

```
def test_wrong_number_args():
    """test for wrong number of arguments"""

    for k in [0, 1, 3]:
        args = ' '.join(map(str, random.sample(range(10), k=k)))
        rv, out = getstatusoutput(f'{prg} {args}')
        assert rv != 0
        assert out.lower().startswith('usage')
```

Likewise, the `test_not_numbers` function runs our program with a random string in position 1 and 2

to verify that the program will reject the value:

```
def test_not_numbers():
    """test for not providing numbers"""

    bad = random_string()
    args = [str(random.choice(range(10))), bad]

    for _ in range(2):
        args = list(reversed(args))
        rv, out = getstatusoutput(f'{prg} {" ".join(args)}')
        assert rv != 0
        assert out.lower().startswith('usage')
        assert re.search(f"invalid int value: '{bad}'", out)
```

If you chose to handle the validation of the arguments manually, then you will have found that your program would need to issue the expected error message and return a non-zero value for each type of error.

The validation of both the **number** and **type** of the arguments is crucial. For instance, all arguments from the command line are strings, so if they are not converted to `int` values, then we run the risk of printing nonsense like `3 + 8 = 38` instead of `3 + 8 = 11`!

Finally our program is run with several valid inputs and checks that the output is as expected. If we look closely at the error, we see that we are supposed to print a string showing the addition of the two arguments and the sum:

```
E           AssertionError: assert '[0, 0]' == '0 + 0 = 0'
E             - [0, 0]     ①
E             + 0 + 0 = 0  ②
```

① This is what our program printed.

② This is what it should print.

Change the `main()` function until it prints the correct output. Run the program or the test suite *after every change to the program*. Always change as little as possible about the program before running and testing!

The solution

The following `main()` would pass the test:

```
def main():
    args = get_args()
    n1, n2 = args.numbers ①
    print(f'{n1} + {n2} = {n1 + n2}') ②
```

① Unpack the two number into the variable `n1` and `n2`.

② Use an f-string to format the two number and their sum into the output string to `print()`.

Conclusion

Testing provides immediate feedback to the student, allowing them to proceed at their own pace. I use this framework for both in-class, live-coding examples as well as for at-home practice exercises. Some students may spend 20 minutes on an assignment, others several hours. The tests let them know what their mistakes are and what is expected to fix their programs.

Using test-driven development to teach programming is arguably harder than simply asking students to eyeball their programs. Most programming books and courses expect the novice to evaluate their own programs and find their own mistakes. While using tests and reading the output is intimidating to the beginner, my experience is that students quickly learn to rely on the tests and, in fact, find joy in each additional test that passes.

I believe we should be using TDD ideas in the classroom not only to provide valuable feedback to the students but also to teach them industry best practices. What I have presented here is an introduction to *integration testing* which tests a program as a whole from the outside, but TDD provides further benefits as we move to the level of *unit testing* where we teach students to write **functions** and **tests for those functions**. It's my sincere belief that teaching students how to debug and test at these finer levels leads to far greater understanding and confidence and will produce programmers who are truly ready to create documented, tested, and reproducible software.

Resources

If you would like to explore these ideas further, I have created the following resources:

- remote.python.pizza presentation and code
- Tiny Python Projects at Manning
- GitHub repo
- YouTube videos for TPP chapters

[1] https://en.wikipedia.org/wiki/Test-driven_development

[2] Either `True` or `False`, a non-zero numeric value, a string value other than the empty string, a non-empty `list` or `dict`, etc.