Welcome.

Everyone:

- Pull the updates from the course GitHub repo:
  - `cd <46120-PiWE repo>`
  - `git pull upstream main` ← you might have “upstream2” instead

Physical students:

- Sit with your P0 Team.
- Turn off laptop volume (mute). ←IMPORTANT!
- Log into the Zoom meeting.
  - Link is on Learn webpage, under “PiWE Links” on main page.
- Double-check your microphone is muted. You can keep your camera off.

46120: Scientific Programming for Wind Energy

Functions and tests

Jenni Rinker

Agenda for today.

- Pull new course material ✔

- Round robin.
- Functions.
- Tests.

- Begin teamwork on Week 2 homework.

Round robin

Share solutions with your peers and give feedback.

How Round Robins work.

- We will tell you which teams should enter which BORs. Usually 2 to 3 teams per room.
- Physical students:
  - Everyone is on Zoom and in the BOR.
  - Sit with physical team members somewhere you can hear/speak.
- Sometimes will be multiple rounds, so you would switch BORs partway through.
- Teams take turns sharing screen and explaining their work.
  - Other teams provide feedback, take notes.
- Discuss both solutions but also what was challenging/confusing.
- TAs/instructors will drop into/out of BORs just to listen in.
- Afterwards, we will discuss as a class interesting things, remaining questions, etc.

Time to review and collaborate.

- 1 round of 40 minutes.
- 5 minutes: chaos.
- 20 minutes: preclass_assignment/ solutions.
  - Team A screenshares & presents their solutions. Teams B & C provides feedback.
  - Switch which group presents/provides feedback.
- 15 minutes: git questions.
  - Go through GitAnswers.md and discuss your answers.
  - Be ready to present a few sentences on (1) what you thought was interesting and (2) any answers you are still unsure about.

- Teams in breakout rooms (BORs):

	Teams
BOR 1	0, 7, 9, 18
BOR 2	1, 10, 19
BOR 3	2, 11, 20
BOR 4	3, 12, 21
BOR 5	4, 13, 22
BOR 6	5, 14, 23
BOR 7	6, 15, 24
BOR 8	8, 16, 17

Notes in plenum.

- Merge conflicts. Happens when there are commits on the same branch on two different copies of the repo that have conflicting changes.
  - What happens if you insert a line in a file? Probably won't a conflict, but we would need to test.
- Zip function.
  - Allows you to iterate over elements in multiple iterables at the same time. E.g., zip(list_a, list_b).
- Could you pull request off a single file even if multiple files have been committed?
  - No, PRs are off of whole branches. Would need to check out into an earlier commit and make a particular branch, or use something called "cherry picking" to make a new branch, and PR off that.

Python functions

Why copy-paste code when you could reuse it?

A function is a black box.

- Anything defined inside the box cannot be accessed outside the box, even if we can visually see it in the code. Only outputs are accessible.

```
def heyo(s):
    print('Heyooooooo', s)
    return
```

Variable s is an argument (input) into the function.

No variables are returned. (Equivalent to `return None`.)

```
heyo('friend')
```

`'Heyooooooo friend'`

```
print(s)
```

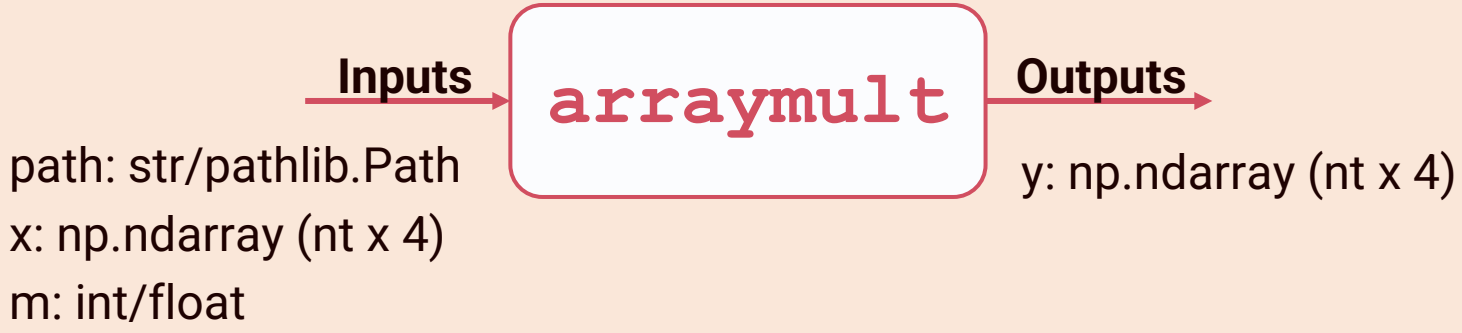
Variable `s` was only defined inside the function, so this raises an error!!!

Drawing your black box.

- An essential tool for discussing code architecture.
  - You are going to do this on your homework and on your programming projects.

- You should include variable types with inputs and outputs! And shapes of arrays/dataframes if relevant.

- An example:



Some exceptions to the black-box rule.

Extra slides on local vs. global variables at end of slide deck

- Variables defined outside of a function (called **global variables**) can be read inside the function.
  - In certain situations, their values can be updated.
- Example of this

```
>> c = 1
>> def myprint():
>>     print(c)
1
```

- Why might using global variables be a **bad** idea?
  - What should you do instead?

Keyword arguments.

- We often want the option to pass optional parameters into a function with a default value, called a **keyword argument**. Example:

```
>> def writehello(path, s='Hello!'):
>>       with open(path, 'w') as f:
>>               f.write(s + '\n')
>> writehello('test.txt')
>> writehello('text2.txt', s='Goodbye!')
```

Argument `s` is a keyword argument, with a default value of "Hello!".

Writes "Hello!" to the file.

Writes "Goodbye!" to the file.

- Keyword arguments can be passed in as a dictionary – see tutorials below.
- More tutorials:
  - https://realpython.com/python-kwargs-and-args/
  - https://www.educative.io/answers/what-are-keyword-arguments-in-python

Good function names.

- There are only few “rules” on what to call a function.
  - Cannot start with number or be same as reserved Python keyword.
- There are further guidelines, however.
- PEP8: Function name should be lower case, with underscores.
  - E.g., not `plotTimeSeries()` but rather `plot_timeseries()`.
- Start with a verb.
  - Verbs “get” and “set” are used a lot in programming in general.
- Name should be specific enough to convey meaning.
- Example of a good function name: `make_lowercase()`.

Importing functions.

- Clean up and reuse code by placing reusable functions in a different file.
- E.g., function `double` in file `myfuncs.py`* can be imported/used:

```
from myfuncs import double

y = double(2)
print(y)
```

Don't include ".py"!

- *Note that this required `myfuncs.py` to be in the same directory as the main script.
  - Later we will learn about packaging, which removes this requirement.
- **Important!** Only place functions/classes in the module you are importing from.
  - If you really want code, use a special if statement to protect it: https://realpython.com/if-name-main-python/

In summary.

- View your function as a black box, where you only know the inputs and outputs.
  - Should be able to diagram it with types!
- Don’t use global variables!
  - More detailed explanation (and links to online tutorial) at the end of this slide deck.
- Keyword arguments are useful when you only want to update a variable sometimes.
- Organize your code by placing reused functions in a module and importing them.

Questions?

Let’s take a break.

Agenda for today.

- Pull new course material ✔️

- Round robin. ✔️
- Functions. ✔️
- Tests.

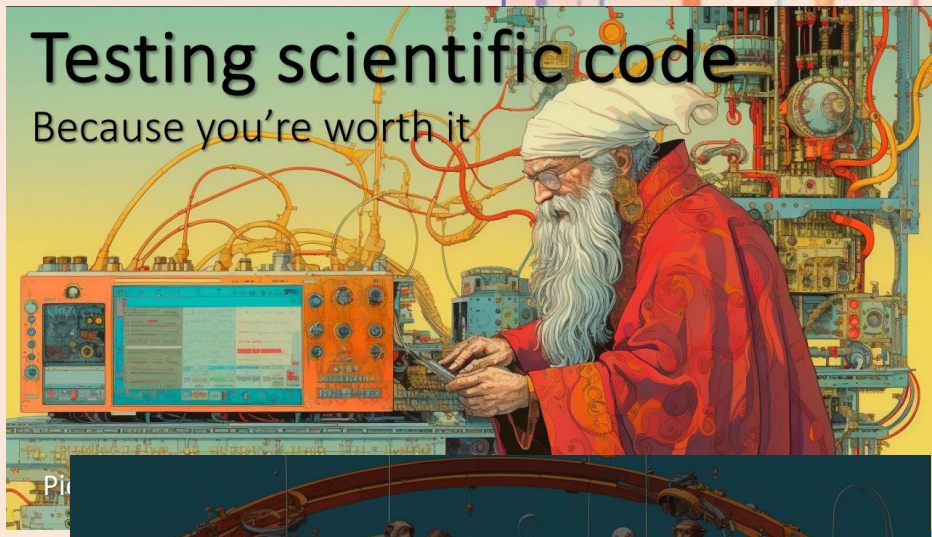
- Begin teamwork on Week 2 homework.

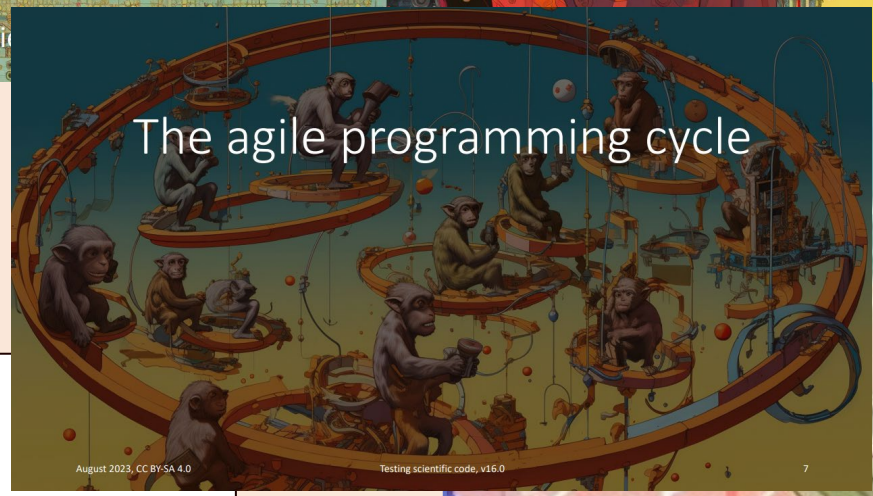
Tests

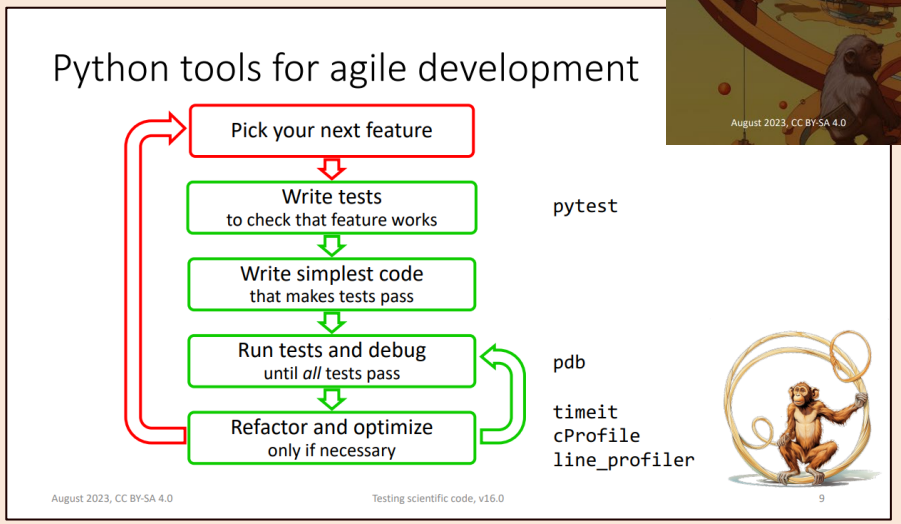
Make sure your code does what it should.

Before I get started.

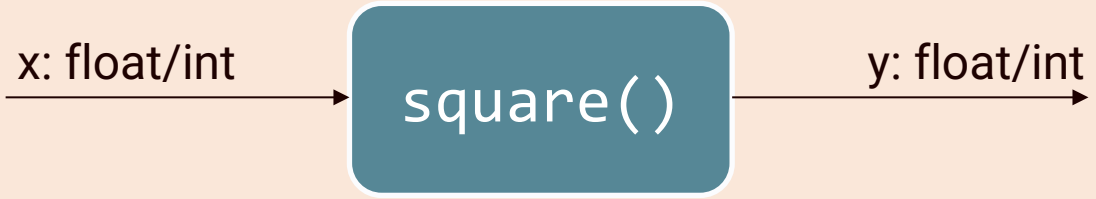
- I highly, highly recommend reading the slides from the lecture on testing and debugging by Pietro and Lisa [1].
  - Part of a summer school in Advanced Scientific Python Programming [2].
  - This lecture is heavily inspired by how Pietro taught me testing in ASPP 2017.
- Pietro and Lisa's slides have more technical details (and gorgeous images) than I will present here.
  - Worth reading over the summer.







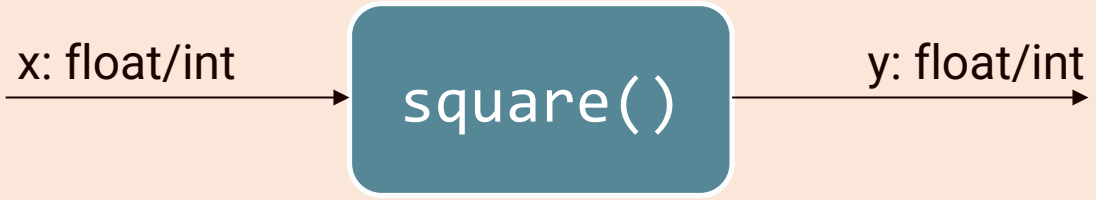
So. We want to make sure a function “works”.



Individually, take 2 minutes and list some behaviors of this function we could/should test.

- Think beyond just calculations – what about object types? What should happen if we get an unexpected input?

Possible things to test.



- Behaviors to test:
  - If an unexpected type is given in (e.g., a list), it should raise and error with a message about unexpected type.
  - If we give in 2, we should get 4.
  - If more than 1 argument is passed in, raise an error with a message.
  - Maybe also test other test inputs, similar to what we did for 2.
  - If I give an integer, it should return an integer. UP TO US.

Content taken from [1]

Anatomy of a test.

- A test is just another function.
- Structure:
  - **Given**. Put your system in the right state for testing.
    - Create data, initialize parameters, define constants, expected output, etc.
  - **When**. Execute the feature that you are testing.
    - Typically, one or two lines of code.
  - **Then**. Compare outcomes with the expected ones.
    - Define the expected result of the test.
    - Use of *assertions* that check that the new state of your system matches your expectations.
- Assert statement raises an error if a provided expression is false.
  - So the test function will check something looks as expected and raise an error if it doesn't.

Let’s see this example for square.

Live-coding!

bold/* = follow along with me

1. Open VS Code in the testing folder.*
2. Examine `square()`, which is part of `arithmetic.py`.
3. Look at (and run) **`test_arithmetic.py`**.*
4. Make the test fail, then fix it.

```python
"""Check some of the functions in arithmetic.
"""
from arithmetic import square


def test_square_integer():
    """Test that the square function returns the correct value for an
    integer input."""
    # given
    x = 2
    y_theo = 4
    # when
    y = square(x)
    # then
    assert y == y_theo


# code to execute only if Python is executed directly on this module,
# NOT on import
if __name__ == '__main__':
    test_square_integer()
```

Exercise: write a test for a float input.

Use the integer-test as an example, make a new function to test floats.

- Individually or in pairs*, write a new function in `test_arithmetic.py` called `test_square_float()` that tests the following values:
  - *Virtual students: find a BOR with another student or 2. Ask for help in Slack if you can't find a BOR.
  - Input is 3.4, expected output is 11.56.
- Remember to add your function to the if block at the end of the module -- otherwise your function will not run!
- Execute `test_arithmetic.py` again. Does your test function pass?
- **To get help**: Post in Slack / #debugging if you want a TA to enter your BOR.

SPOILERS! It's not so easy this time.

  - You have "finished" this exercise when you have written a test that you think *should* work. We'll come together and discuss how to make the test *actually* pass.

Let’s live-code the “solution” together.

- Tell me, what shall I write?

Common pitfalls of scientific testing.

This slide is mostly for reference. Please see slides 23 and higher in [1] for more details.

- Floating-point numbers.
  - Use `np.isclose` and change the tolerance if needed.
- Numpy arrays.
  - Use `np.testing.assert_equal` or `assert_allclose`.
- NaNs.
  - They aren't equal to themselves. Use `np.isnan`.

Downsides of the current test configuration.

- There are some drawbacks with the current testing module:

1. If any test fails, none of the subsequent tests are run.
2. Adding lines at end of file adds extra chances for human error.
    - I.e., someone writes a test but forgets to add the line at the bottom of the file.
3. Placing all tests in a single module can be cumbersome for big projects.
4. We can’t run a subselection of tests, e.g., only run the smoke tests.
5. We can’t test for more advanced behaviors, e.g., if a particular error is raised.
6. There is no way to know how much of our code is untested.

A hero to the rescue!

pytest and pytest-cov.

- pytest:
  - “[A] mature full-featured Python testing tool that helps you write better programs” [4].
  - Collects all of your test functions (assuming a certain naming convention), runs them all, then assembles the information into a report.
  - Rich flexibility for parameterization and other test fixtures.
  - Can also run two other common types of test suites (unittest and nose) out of the box.
  - Can be expanded via plugins.
- pytest-cov:
  - A plugin for pytest that lets you calculate percentage of code covered by a test suite.
  - We’ll use this later in the semester. ☺

How to run pytest.

- Let’s do it together!

- Run pytest on a file or folder:

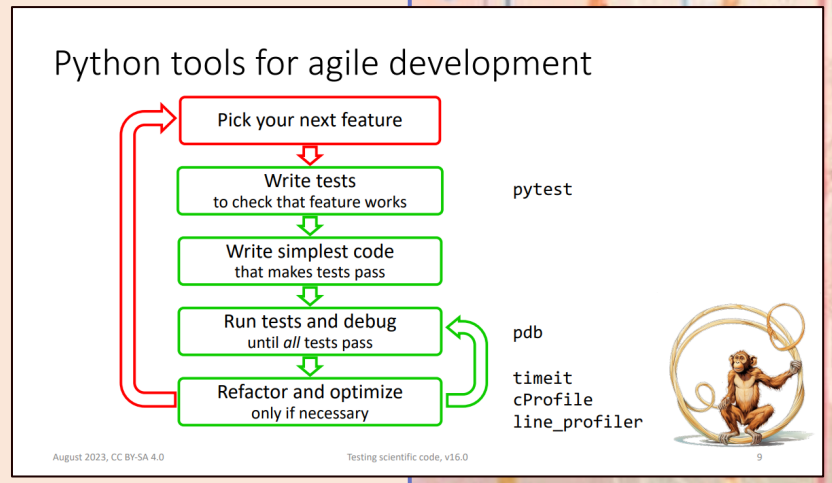
  `pytest filename.py` ← run pytest on a particular file (executes all functions that start with `test_`)

  `pytest foldername/` ← collects all files who start with “test_”

  `pytest .` ← collect files in current folder that start with “test_”

Final notes.

- I only scraped the surface in this lecture.
  - Did not discuss unittest or nose tests.
  - Also didn't discuss integrating your test suite with CI/CD pipelines.
  - Nor test-driven (agile) development.
- Pytest has a bunch of really cool utilities that I did not discuss.
    - Labelling tests so you can run a subselection by name.
    - Labelling tests that are expected to fail, e.g., if a feature is not yet implemented.
    - Parameterizing tests so you can check multiple inputs with the same test.
    - Creation of temporary files/directories for testing.
    - So much more!



"test-driven development"

Homework for this week

Time to get your hands dirty!

Overview.

*Remember, you're expected to work **about 6 hours outside of class**. Schedule time for that.*

- As always, your homework is detailed on the course GitHub repo.
  - **Short summary**: restructure your `preclass_assignment` into functions and a main script, diagram your functions, and write tests for them.

In a moment, we will open BORs, one for each team.

- Each team enters their BOR (same as your team ID). Perhaps find a physical spot outside the auditorium.
- Complete **Part 1** of the weekly assignment in class, then move on as agreed with your team.
- **To get help during class**: Post in Slack / #debugging if you want a TA to enter your BOR or come find your group.

Any questions?

References.

1. Pietro and Lisa's GitHub on Testing and Debugging from 2023 GitHub - ASPP/2023-heraklion-testing-debugging
2. Summer School in Advanced Scientific Python Programming ASPP2024/start
3. Tutorial on effective testing with pytest Effective Python Testing With Pytest – Real Python
4. Pytest docs https://docs.pytest.org/

More on global vs. local variables

For the curious.

Namespaces and scopes.

- Really good article on this: https://realpython.com/python-namespaces-scope/
- The distinction between global and local variables is tied up into broader concepts called namespaces and scopes.
- A **namespace** is a place Python stores the symbolic names of the variables you define in a code.
- There are actually four different namespaces (built-in, global, enclosing, local) where Python can look for a variable you have defined.
  - The last line where you assign a value to your variable (e.g., “x = 4”) determines its namespace.
- The order in which Python looks for a defined variable is Local, Enclosed, Global, Built-in.
  - Referred to as **LEGB rule**.

Example of a non-explicit global variable that works.

Using global variables like this is NOT recommended. See last slide for discussion on why.

```
c = 1


def myfun():
        print(c)


myfun()
```

Here is our assign statement, so Python will look for this variable in the global space.

Here we tell Python to acquire the value of c, then print the value.

Python looks first in the local namespace but does not find a “c” there. So then it looks in the enclosed namespace – same story.

Finally it looks in the global namespace and finds that a symbolic “c” has been defined, with value 1.

So this code would print “1”.

Example of a non-explicit global variable that DOESN'T work.

```
c = 1


def myfun():
        c = c + 2
        print(c)


myfun()
```

Here is our last assign statement, so Python classifies this as a local (not global) variable.

Here we tell Python to acquire the value of c, then print the value.

The problem now is that the interpreter believes that c is a local variable, but the *value* of c has never been defined in the local scope!

So if we execute this code, we will get an UnboundLocalError.

Example of an explicit global variable that works.

Using global variables like this is NOT recommended. See last slide for discussion on why.

```
c = 1

def myfun():
        global c
        c = c + 2
        print(c)

myfun()
```

Here is our assign statement, so Python will look for this variable in the global space.

We can (but shouldn't) use the "global" declaration to declare that a variable is defined in the global scope.

Now the interpreter knows where to access the value of c.

This code would print "3" upon execution.

Why you shouldn’t use global keywords.

- An example: I define a function `myfun` in a module that takes a variable x and increments its value by 2. But let’s say you don’t know that. Suddenly, this code would be extremely confusing to you:

```
>> x = 4

>> myfun(x)

>> print(x)

6
```

- Similarly, let’s say I share with you a function that looks like this:

```
def myfun2(x):

        return m*x
```

  - Maybe I always define a global variable “m” before calling this function, but you don’t know that! Suddenly you can’t run my code! Oh no!

Why you shouldn't use global keywords.

- The use of global variables likely makes your code less (a) debugable and (b) shareable.
- It is much better practice to pass your variables in as arguments. If we rewrite the `myfun2` function from the last slide, it looks like this:

```
def myfun2(x, m):

    return m*x
```

  - So, now we explicitly passed in the "m" variable, making the code more clear and easier to share. Nice!
- Finally, I highly recommend reading this tutorial. It explains these concepts in even more detail and with several code examples.
  - https://realpython.com/python-namespaces-scope/

More examples and tutorials of global and local variables.

- Global examples:
  - Python Global Keyword (With Examples) (programiz.com)
  - Global and Local Variables in Python – GeeksforGeeks
- Why you get error:
  - UnboundLocalError: local variable referenced before assignment in Python. | by VINTA BHARATH SAI REDDY | Medium
- Why you shouldn’t do this:
  - Why Is Using Global Variables Considered a Bad Practice? | Baeldung on Computer Science