Warmup (L3)

Create a cell in Jupyter that has a different behavior the first time you run it than the second time you run it.

Note 1: You may need to create and run another cell first to set up the environment

Note 2: Your cell may show an error when you run it, but as long as it *also* has different output it works

Bonus: Make the cell have a different behavior *every* time you run it

Bugs, bugs, bugs!

CS114 L3 (M1)

Debugging

- Time for debugging!

```
from math import sqrt
def pythagoras(a, b):
    sqrt(a**2 + b**2)
def pythagoras3(a, b, c):
    return sqrt(pythagoras(a, b)**2 + c**2)
print(pythagoras3(4, 5, 6))
```

Debugging

One thing to be careful of when debugging: errors happen when code *runs*, so if you don't run a buggy function, you won't see the problem!

```
from math import sqrt
def pythagoras(a, b):
    sqrt(a**2 + b**2)
def pythagoras3(a, b, c):
    return sqrt(pythagoras(a, b)**2 + c**2)
print(pythagoras(4, 5))
```

Avoiding bugs

- Develop incrementally (one small step at a time), using `print` to spot-check
- Let's make a `distance` function (distance between two points) incrementally...

```
def distance(x1, y1, x2, y2):
    ...
```

Avoiding bugs: printing

- Make sure whatever you print is descriptive/unique enough that you know which is which

```
def pythagoras(a, b):
    asquared = a ** 2
    print("a squared:", asquared)
    bsquared = b ** 2
    print("b squared:", bsquared)
    r = sqrt(asquared + bsquared)
    print("result:", r)
    return r
```

Avoiding bugs: printing

Don't be afraid to introduce variables just so that you can print something from the middle of a calculation!

```
def pythagoras(a, b):
    asquared = a ** 2
    print("a squared:", asquared)
    bsquared = b ** 2
    print("b squared:", bsquared)
    r = sqrt(asquared + bsquared)
    print("result:", r)
    return r
```

Avoiding bugs: printing

- When you're done debugging, make sure you remove the prints. They'll confuse our tests!
- You can also *comment out* prints, so you can remove them without forgetting them

```
def hypotenuse(a, b):
    # print("a was", a)
    return sqrt(a**2 + b**2)
```

Avoiding bugs: documentation

- Part of avoiding bugs is *good documentation*
- You shouldn't name a function "`pythagoras`"
  - The Pythagorean theorem is an implementation detail
  - When you're calling the function, you don't care how it's implemented
  - It should've been named "`hypotenuse`"

Avoiding bugs: documentation

- Remember the `help` function?
- We can (and should!) document our functions in the same way, with *docstrings*
- Let's add a docstring to `pythagoras`

Avoiding bugs: documentation

```
def pythagoras(a, b):
    """
    Return the length of the
    hypotenuse of a right
    triangle with side lengths a
    and b.
    """
    return sqrt(a**2 + b**2)
```

Docstrings

- The docstring is the first statement in a function
- Triple-quote can be used to make a multi-line string anywhere; it's only a docstring when it's the first statement

```
def sillystring():
    return """
    Hello world!
    """ # This is not a docstring
```

Docstrings

- In this course, you must write a docstring for every function you write
- Your code is graded not just for correctness (doing what it's supposed to) but for documentation and readability!

Docstrings

- Conventions for good docstrings:
  - Refer to parameters by name

```
"""… with side lengths a and b."""
```

not

```
"""… with the two sides."""
```

Docstrings

- Conventions for good docstrings:
  - The docstring should be a command for the function to obey

```
"""Return the length…"""
```

not

```
"""Computes the length…"""
```

Docstrings

- Conventions for good docstrings:
  - Describe what it does, not how it does it

```
"""Return the length…"""
```

not

```
"""Return sqrt(a**2+b**2)"""
```

Comments

- Comments (with #) should be used to clarify.
- It's assumed that whoever's reading the code knows English and Python, so
  - If your variable names are good and the steps are obvious, you may not need comments to make the code understandable.
  - but, err on the side of caution: use comments where it *might* be confusing.
  - Assume the reader is as stupid as possible while still being able to read English and Python

Avoiding bugs: types

- We've seen numbers and we've seen strings
- Look at the result of `hypotenuse`: note how `hypotenuse(3, 4)` is written as 5.0, not just 5
- Internally, Python stores integers and rationals differently
- It is often useful to distinguish between them

Numeric types

- In Python, we can store a number as an `int` or a `float`. `int` corresponds to integers. `float` corresponds to rationals (and is used to approximate reals).
- Since all integers are rationals, we can store an integer as a `float`.
  - If a number *could have been* a non-integer, it’s usually a `float`, even if it *is* an integer in practice!

Numeric types

- In Python, we can store a number as an `int` or a `float`. `int` corresponds to integers. `float` corresponds to rationals (and is used to approximate reals).
- Note "*corresponds to*" here. An `int` can store any integer (as long as your computer has enough memory!), but `float`s have limited capacity. It's hard to intuit about, so just remember: it's rounded!

Float?

- “float” stands for “floating point”.
- It means the point (the dot separating the integer part from the fractional part) can float (be anywhere within the number)
- Don’t overthink the name. It’s just a name.

Numeric types

- It mostly won't matter how a number is stored (`float` or `int`)
- … but it can. We'll see situations later where it matters.
- You can convert in a few ways:

```
float(42) # 42.0
int(41.999) # 41
round(41.999) # 42
```

Documenting types

- It's often useful to document what type you expect something to be
  - If it's the wrong type, your code will do something unexpected!
- When writing a function, you can (and should!) document the types of its parameters and the type it returns

Documenting types

```
def hypotenuse(a: float, b: float) -> float:
    return sqrt(a**2 + b**2)
```

- These *type annotations* are documentation. Even if they're wrong, your code will run.
- If you set up Jupyter with `Assignment-00.ipynb`, it will warn you when they're wrong

Documenting types

- Although documentation, annotations are so-called *checked documentation*
- That is, they don’t change your code (just document it), and yet we can check that they’re correct
- If you don’t use types as you describe them, you’ll get typing errors, but your code will still run

int vs float

- Every integer is a rational, so it's tempting to write `float` everywhere
- This is poor documentation! If you expect an integer, write `int`

```
def stableNeutrons(protons: int) -> float:
    """
    Return approximately how many neutrons
    are needed to stabilize a nucleus with
    this many protons.
    """
    return protons * 1.5
```

Type errors

- Using an unexpected type won’t always stop your code from running
- But it can. E.g., trying to treat None like a number will cause an error

```
print(42)/(7) # Note wrong parentheses
```

- Let’s explore the errors reported by the above code (it’s a lot!)

Typing errors vs type errors

- Typing errors (or type-checking errors) are about the *documented types*. Code doesn’t have to run to check, and errors don’t prevent code from running.
- Type errors happen while code is running, and stop it from running further.
- Both are about types, and the names are confusing, but they’re distinct

Learning to read errors

- Making errors readable and understandable is an area of active research (no, really!)
- For typ*ing* errors:
  - Look for “expected” and “got”.
  - Think about which way data is moving (into parameters, out of returns)
- For *type* errors and other errors:
  - Start from the *end* and work your way backwards to understand where it’s happening

Learning to read errors

- Let’s explore some errors
- ```
  print(42/7
  def hello():
      print("World")
  ```
- ```
  def return():
      return 42
  ```
- ```
  def curious():
      return sqrt(9)
  ```
- ```
  fancy:fancy:variable = 42
  ```
- ```
  def weird(x):
      x = x * 5
    return x
  ```
  ```

Testing

CS114 L3 (M1)

Testing

- Bad development: keep poking at it until it looks right
- Good development: write examples of how it should behave, then write it until it does behave
- Running these examples is *testing*

Assertions

- Python has a built-in technique for testing: `assert`

  as·sert (asserted; asserting; asserts)
  transitive verb
  1a: to state or declare positively and often forcefully or aggressively
  (— Merriam Webster dictionary)

- You assert (declare) something to be true, then fix your code ’til it is ☺

Assertions

```
assert hypotenuse(3, 4) == 5, "3-4-5 triangle"
```

The test
(what you want to be true)

Name for the test
(Documentation)

Assertions

```
assert hypotenuse(3, 4) == 5, "3-4-5 triangle"
```

Since = is variable assignment, == is equality

Assertions

```
assert hypotenuse(3, 4) == 5, "3-4-5 triangle"
```

But *do not use* == *with floats!!!*
(Remember, they're rounded!)

Assertions

```
assert hypotenuse(3, 4) == 5, "3-4-5 triangle"
```

Note no parentheses. assert is a keyword, not a function! If you add parentheses with the test name, it won't do what you think!

Let's look at

```
assert(1 == 0, "Math is broken")
```

Testing with floats

- `float`s have limited precision
  - Internally, they're scientific notation in base-2, so have limited base-2 significant figures, but don't try to intuit about base-2 scientific notation…
- The precision can go wrong in very surprising ways:

```
assert 0.3-0.2 == 0.1, "Math too simple to fail"
```

Imprecise numbers, imprecise tests

- Always test floating points with a range, called the *tolerance*, to avoid precision problems
- Simplest way to test with tolerance is

$$|result - expected| < tolerance$$

- In Python, that looks like this:

```
assert abs((0.3 - 0.2) - 0.1) < 0.001, "Precision problems!"
```

Imprecise numbers, imprecise tests

Result

Expected

```
assert abs((0.3 - 0.2) - 0.1) < 0.001, "Precision problems!"
```

`abs` is the absolute value function

< is less-than

Range here is 0.001
(Don’t overthink it)

Imprecise numbers, imprecise tests

- Use similar code any time your numbers won't be integers

```
assert abs(hypotenuse(4, 5) - 6.4031242) < 0.001, "..."
```

Code style

- In this course, every function must have
  - A docstring,
  - parameter and return types, and
  - at least two test assertions (other than the ones we provide you)

Code style

- We set two tests as a *minimum*
- For most functions, it won’t be enough
- Think about corner cases
- … but don’t overthink it. Just make up some tests.

Other assertions

- **`assert`** is often used for tests, but can also be used for other checks
- For instance, if we want to make sure our arguments are positive:

```
def hypotenuse(a: float, b: float) -> float:
    assert a > 0, "a must be positive"
    assert b > 0, "b must be positive"
    return sqrt(a**2 + b**2)
```

Module summary

CS114 L3 (M1)

Module summary

- We'll be writing Python in Jupyter
- Imperative programming language: give your computer a sequence of commands
- Calculation in Python
- Computation = calculation + repetition + decision-making
- Functions box up behavior
- Programming is mostly fighting bugs