

Warmup (L2)

Print "Hello, world!", with the space, *without* putting a space in your string.

That is,

```
print("Hello, world!")
```



But, no space allowed here!
Get it to print the space without writing this space.

The power of change

- Variables are *variable*. You can not only set them, but *change* them.

```
x = 7
print("x is", x)
x = x * 6 # Huh???
print("x is", x)
```

- Mathematicians hate it!
(This isn't what "variable" or "=" mean in math)

One step at a time

- Remember, we're running commands *in order*
- If you change a variable after it's been used, that use is in the past

```
x = 7
y = x * 2
x = x * 3
print("x is", x)
print("y is", y)
```

Variable names

- Variable name must
 - start with a letter or underscore, and
 - contain only letters, underscores, and numerical digits. In particular, no spaces.
- Nothing stops you from overwriting `print` or other functions (other than common sense), so try not to step on your own toes!

Variable names

- Multiple words in a variable name:
`snake_case` and `camelCase` are both common.
- Style conventions:
 - Use either `snake_case` or `camelCase`, but be consistent (except when an assignment demands)
 - Use descriptive names (not `x`) wherever possible

Variable names

Names are case sensitive

(a_b_c is different from a_B_c or A_b_c)

Jupyter is weird

- Enter this into a cell and run it:

`z = 42`

- Then clear the cell, and replace it with this:

`z / 7`

- You'll notice that it shows a result. It remembered `z`.

Jupyter is weird

- This “hidden state” is there so that cells can work together (set z in one cell, use it in another)
- But, can allow cells to work together through time (set z in one cell, delete that assignment, then use it anyway)
- That can be confusing

Jupyter is weird

To make sure you're not relying on this, occasionally use the "run all" button

- This resets all the variables (called "restarting the kernel")

Comments

- In an earlier example, I snuck in the question “Huh?”:
 - `x = x * 6 # Huh???`
- This is called a *comment*. You can make your code easier to read by describing what’s happening using comments.
- Starts with a `#`, goes to the end of the line
- Can be a line of its own

More power

- Right now we have a five-function calculator
- The functions in a scientific calculator are available in a *module* called `math`
- *Modules* are bundles of functions, values, variables, or anything else, grouped together for organization

Pythagoras

- Let's calculate the length of the hypotenuse of a right triangle given the length of its two sides

Pythagoras

- Let's calculate the length of the hypotenuse of a right triangle given the length of its two sides

```
import math  
math.sqrt(a**2 + b**2)
```

Smarter imports

- You can also import a specific name

```
from math import sqrt, log  
sqrt(a**2 + b**2)
```

- There's no consistent style for whether to import specific names or whole modules. Use whatever is readable. You *do not* have to be consistent if importing multiple modules.

Too many things!

- The `math` module provides a *lot* of functions... and there are a *lot* of modules!
- Find what's in it with the *tab* key
- The `help` function gives you help on anything you've imported
`help(sqrt)`
- You can also use the `dir` function (instead of *tab*) to find what's in a module
`dir(math)`

Reusing (functions)

CS114 M1

Back to Pythagoras

- Note how I wrote the code:

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

- What's this a and b ?
- We want to use this code for any value of a and b
- Right now, if we need it twice, we have to rewrite this code!

Functions

- We already saw functions such as `print` and `sqrt`
- Now let's write one of our own!

Functions

- We already saw functions such as `print` and `sqrt`
- Now let's write one of our own!

```
def pythagoras(a, b):  
    return sqrt(a**2 + b**2)
```

Functions

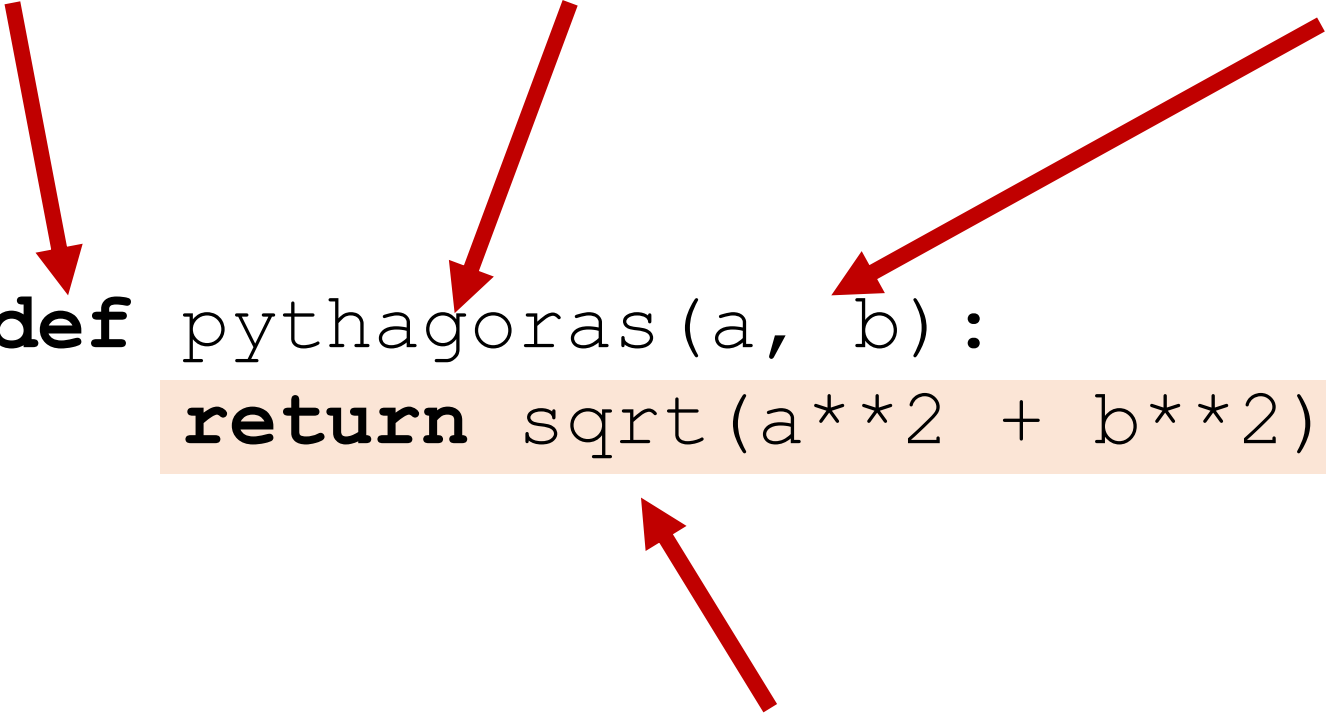
- We **define** a function with the keyword **def**
 - (A keyword is a word that has a special meaning in the programming language, other than just a name)
- We want this function to be *parameterized*
 - Similar to `sqrt`. We need to be able to tell `sqrt` what we want to square-root, and we need to be able to tell `pythagoras` the sides of our triangle

Functions

Define

Function name

Parameters



```
def pythagoras (a, b) :  
    return sqrt (a**2 + b**2)
```

Body of the function (what it actually does)

Functions

Header



```
def pythagoras(a, b):  
    return sqrt(a**2 + b**2)
```



Body is *indented* past the header
(that's how Python knows it's the body!)

Functions

- Defining the function doesn't make anything happen
- We have to *call* the function for it to run

```
def pythagoras(a, b):  
    return sqrt(a**2 + b**2)  
[...]  
pythagoras(3, 4)
```

Functions

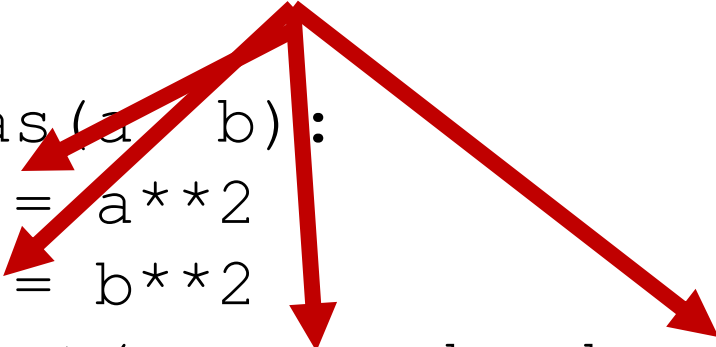
- A function can be any number of steps

```
def pythagoras(a, b):  
    asquared = a**2  
    bsquared = b**2  
    return sqrt(asquared + bsquared)
```

Functions

Functions can define and use their own variables

```
def pythagoras(a, b):  
    asquared = a**2  
    bsquared = b**2  
    return sqrt(asquared + bsquared)
```

A diagram consisting of four red arrows originates from a single point above the function definition. One arrow points to the parameter 'a', another to 'b', a third to the variable 'asquared' in the second line, and a fourth to the variable 'bsquared' in the third line. A fifth arrow points from the same origin point down to the 'sqrt' function call in the return statement, illustrating that the function uses its own local variables.

Functions and variables

- They are the function's *own variables*
- They don't exist outside of the function:

```
def pythagoras(a, b) :  
    asquared = a ** 2  
    bsquared = b ** 2  
    return sqrt(asquared + bsquared)  
pythagoras(3, 4)  
print(asquared) # ERROR!
```

Functions and variables

- Parameters are also variables
- You can overwrite them

```
def pythagoras(a, b):  
    a = a ** 2  
    b = b ** 2  
    return sqrt(a + b)
```

- This is usually confusing and should usually be avoided, but use common sense

Local vs. global

- Variables within a function are *local variables* of that function
- Variables outside the function (defined for the whole kernel) are *global variables*

Pedantry aside

- I've used the terms "parameter" and "argument"
 - In `pythagoras`, `a` and `b` are parameters
 - The value you actually pass in is the argument: the argument fills the parameter
- The confusion: if we talk about what `pythagoras` does, we'll talk about `a` and `b` as their future values, the arguments
- Often used interchangeably, but technically not the same

Functions calling functions

- Our `pythagoras` function calls `sqrt`
- We can also call our own functions in functions

```
def pythagoras(a, b):  
    return sqrt(a**2 + b**2)  
[...]  
def pythagoras3(a, b, c):  
    return sqrt(pythagoras(a, b)**2 + c**2)
```

Returning

- Here, we use a call to `pythagoras` as a value in a calculation. What actual number is used in the calculation?

```
def pythagoras(a, b):  
    return sqrt(a**2 + b**2)  
[...]  
def pythagoras3(a, b, c):  
    return sqrt(pythagoras(a, b)**2 + c**2)
```

Returning

- The **return** statement of the `pythagoras` function gives a value to whoever called `pythagoras`

```
def pythagoras(a, b):  
    return sqrt(a**2 + b**2)  
[...]  
def pythagoras3(a, b, c):  
    return sqrt(pythagoras(a, b)**2 + c**2)
```



Returning

- ***WARNING!*** Returning *ends* the function, even if there are more statements left!

```
def pythagoras(a, b):  
    return sqrt(a**2 + b**2)  
    print("This will never be printed!")
```

- Why would you want this? We'll see when we get to decision-making.

Functions calling functions

- `print` is also a function
- Very useful for understanding and debugging code!

```
def pythagoras3(a, b, c):  
    h = pythagoras(a, b)  
    print("Hypotenuse of one face:", h)  
    return sqrt(h**2 + c**2)
```

Understanding

- I cannot overstate enough the value of using `print` to understand and debug your code!
- `printing` what's happening while your code runs is your first line of defense! Use it! A LOT!

To return or not?

- A function doesn't have to explicitly return

```
def pythagoras(a, b):  
    sqrt(a**2 + b**2)
```

- But, if it doesn't, it still returns a value: "None", a special value that means "there is no value". ... but it is still a value...

None is Hell

- Why have functions return `None`?
- We want functions that are just there to *do* something, rather than *returning* something...
- but Python has no way of distinguishing these two kinds of functions.
- So, Python needed *something* for
`x = print("Whoops!")` to do.

None is Hell

- You can store `None` in a variable and Python won't report anything wrong
- You can pass `None` as an argument to a function, and it'll work until (and unless!) it's used
- Forgetting to **return** causes a problem to arise *distant from the actual error in the code*

Bugs, bugs, bugs!

CS114 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)
- Test every small part by making sure it behaves as you expect. This can be **returning** what you expect or just doing what you expect.
- Use `print` to spot-check

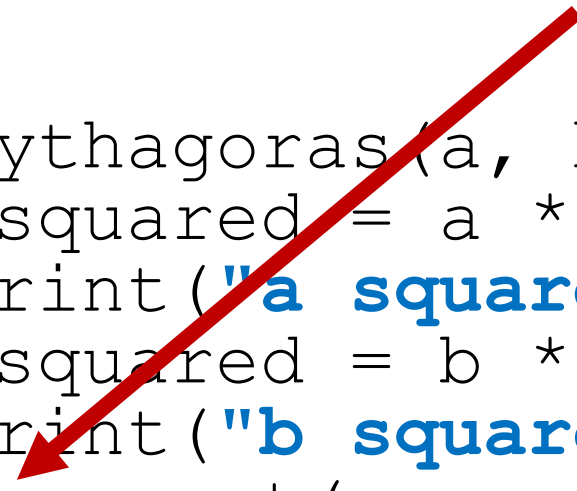
Avoiding bugs: printing

- 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:
 - Describe what it does, not how it does it

`"""Return the length..."""`

not

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

Docstrings

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

`"""Return the length..."""`

not

`"""Computes the length..."""`

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