COLUMBUS STATE
COMMUNITY COLLEGE

WIIT 7740: Scripting with Python

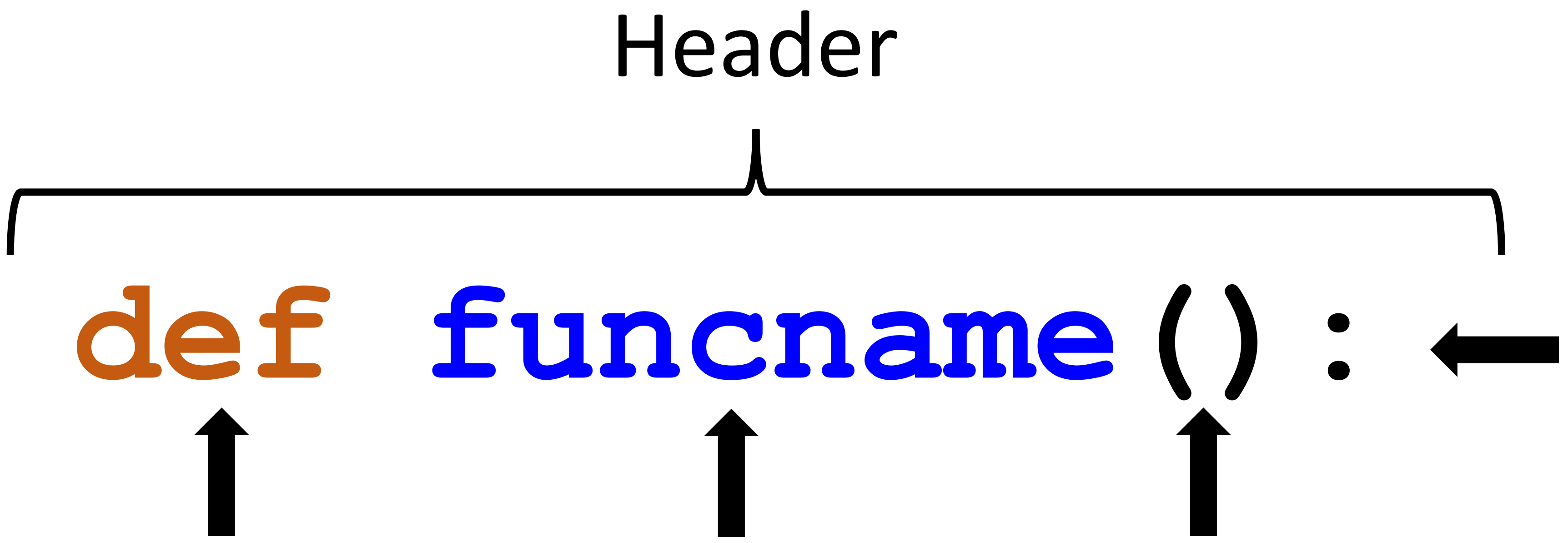
Week 3: Functions

Functions: Purpose

A function is reusable code that you can call by name

- Functions are the building blocks of code
- Name a group of statements and refer to them by name
- Programs easier to read, understand, and debug
- Make programs smaller by eliminating repetitive code
- Can be reused in multiple programs

Anatomy of a Function



```
def funcname():
```

All function definitions start with the term “def” because we are **defining** a NEW function.

The name of your function, which you will “call” later

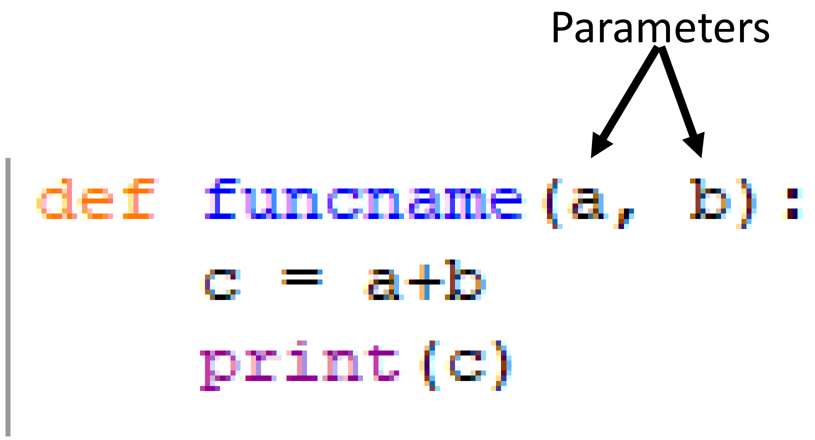
Parentheses. Sometimes, these enclose **parameters** which can accept **arguments** when you call the function.

Colon
Tells Python that everything indented below this point is part of this Function’s **body**.

Passing Data Into Functions

Parameters

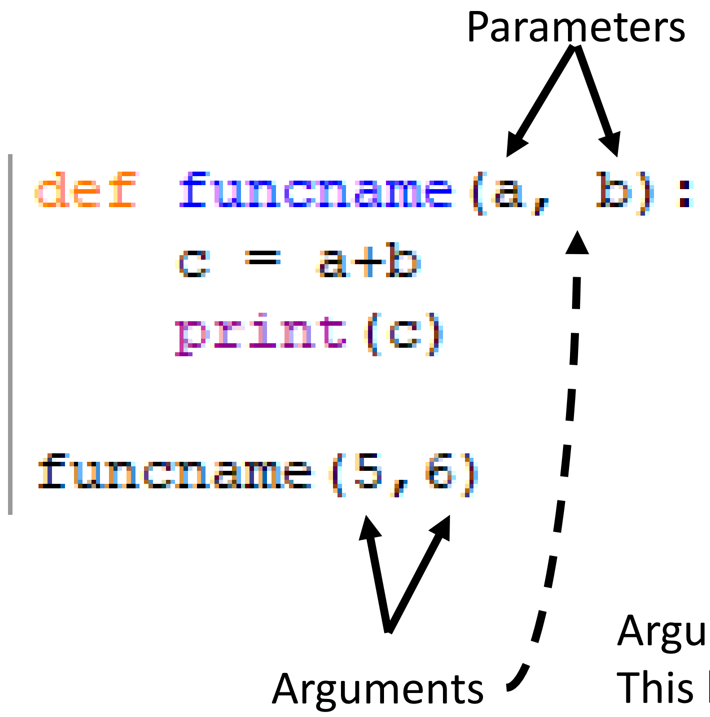
- Are variables defined within the function.
- Only are available inside of the function's **scope** (body of the function).
- May not be referred to outside of the function within your main program.
- Parameters accept the "arguments" passed into them and ideally do some work. For example, adding them together or calling another function like print().



```
def funcname(a, b):
    c = a+b
    print(c)
```

Passing Data Into Functions through Arguments

We pass data into Functions through **Arguments**



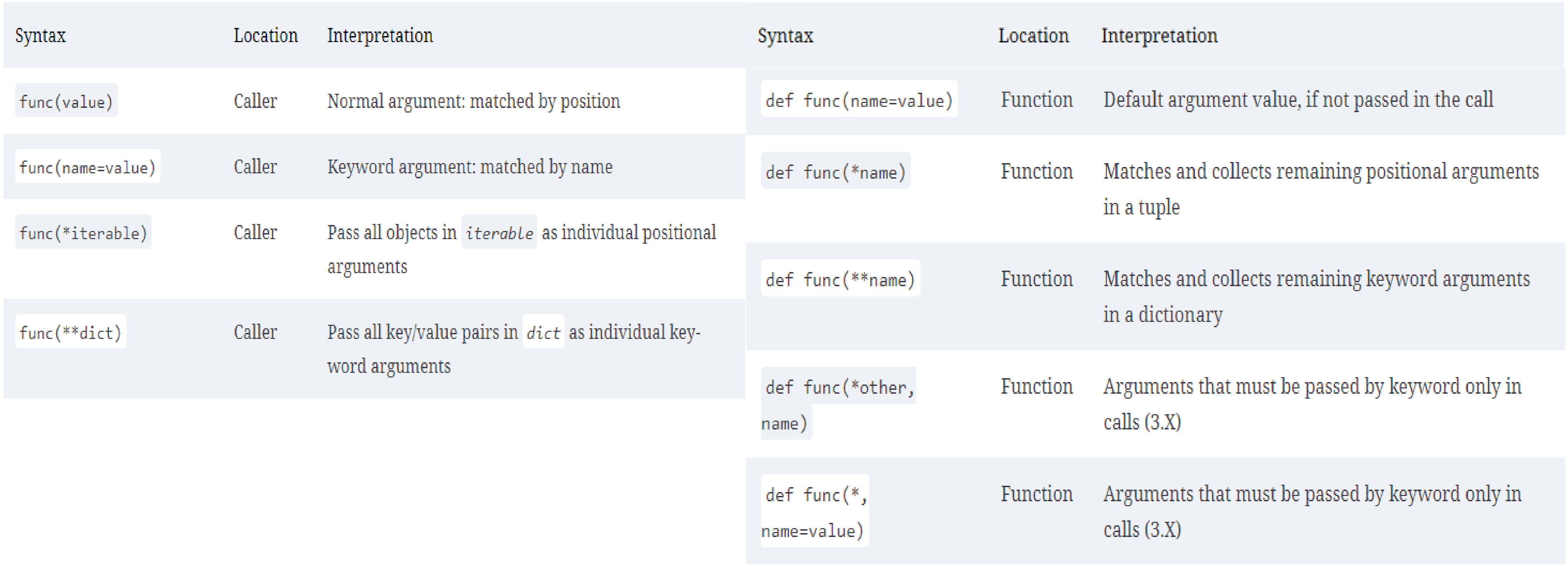
```
def funcname(a, b):
     c = a+b
     print(c)

funcname(5,6)
```

Arguments get "passed" into the function.
This lets the function to its job.

Passing Data Into Functions

We can pass Arguments several ways:



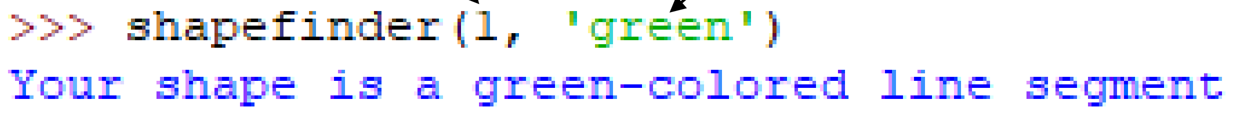
Syntax	Location	Interpretation
`func(value)`	Caller	Normal argument: matched by position
`func(name=value)`	Caller	Keyword argument: matched by name
`func(*iterable)`	Caller	Pass all objects in *`iterable`* as individual positional arguments
`func(**dict)`	Caller	Pass all key/value pairs in *`dict`* as individual keyword arguments

Syntax	Location	Interpretation
`def func(name=value)`	Function	Default argument value, if not passed in the call
`def func(*name)`	Function	Matches and collects remaining positional arguments in a tuple
`def func(**name)`	Function	Matches and collects remaining keyword arguments in a dictionary
`def func(*other, name)`	Function	Arguments that must be passed by keyword only in calls (3.X)
`def func(*, name=value)`	Function	Arguments that must be passed by keyword only in calls (3.X)

Passing Data Into Functions: Position & Keyword

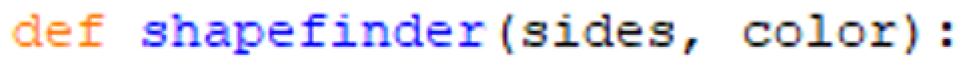
These methods utilize **Position** and **Keyword**.

1st position 2nd position



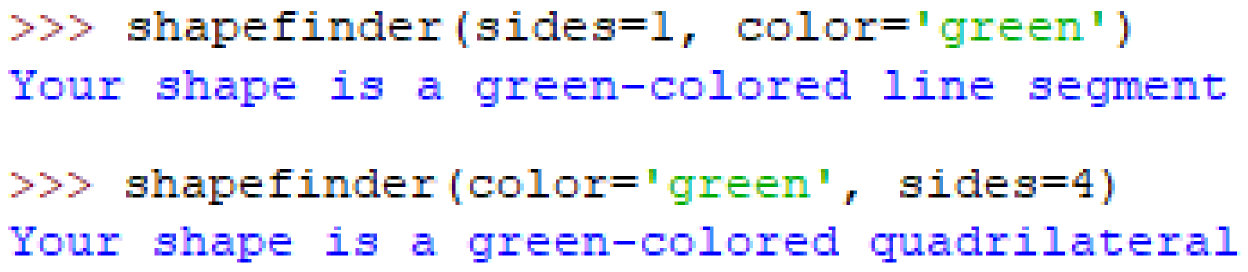
```
>>> shapefinder(1, 'green')
Your shape is a green-colored line segment
```

1st position 2nd position



```
def shapefinder(sides, color):
```

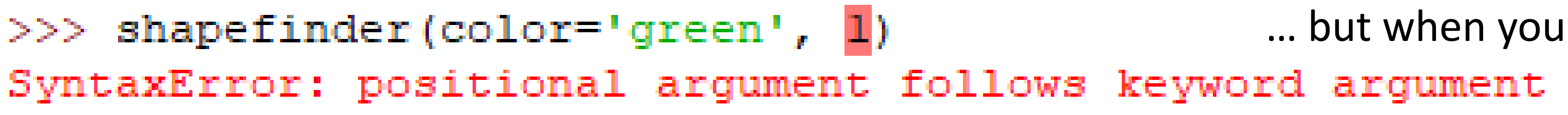
sides, color used as keywords:



```
>>> shapefinder(sides=1, color='green')
Your shape is a green-colored line segment

>>> shapefinder(color='green', sides=4)
Your shape is a green-colored quadrilateral
```

When you use only keywords, the position doesn't matter…

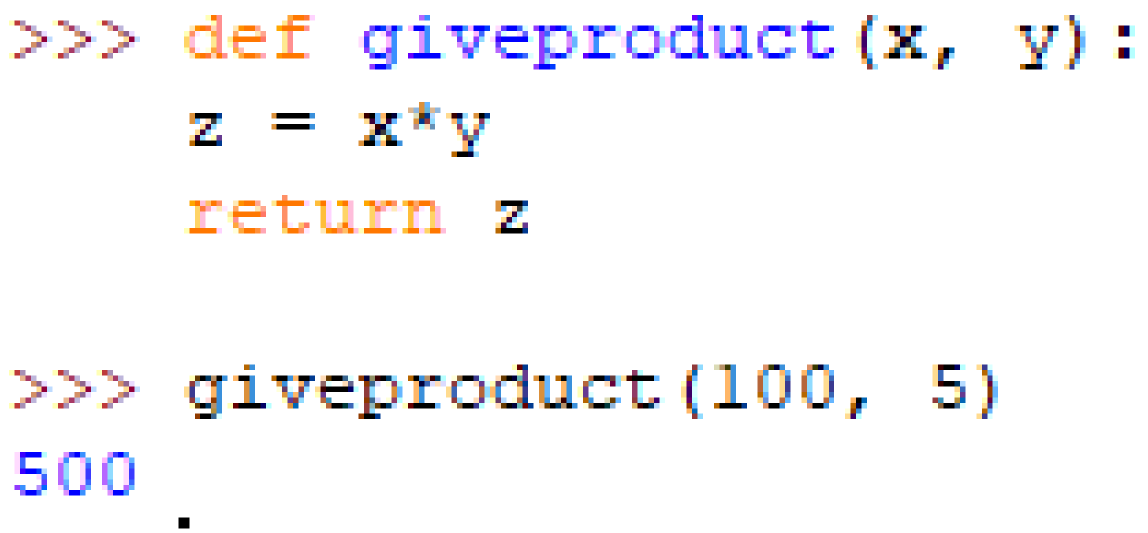


```
>>> shapefinder(color='green', 1)
SyntaxError: positional argument follows keyword argument
```

… but when you use a mixture, **order does matter.**

Getting Data Out

We get data out of Functions through the **return** command.



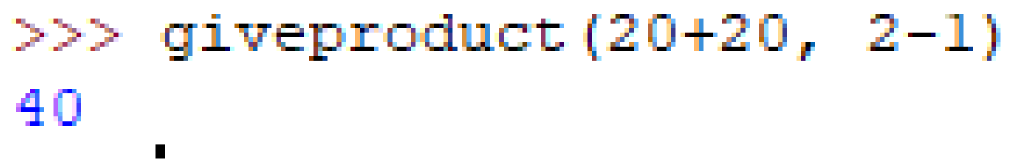
```
>>> def giveproduct(x, y):
        z = x*y
        return z

>>> giveproduct(100, 5)
500
```

We can also return multiple values!

```
return z, x, y
```

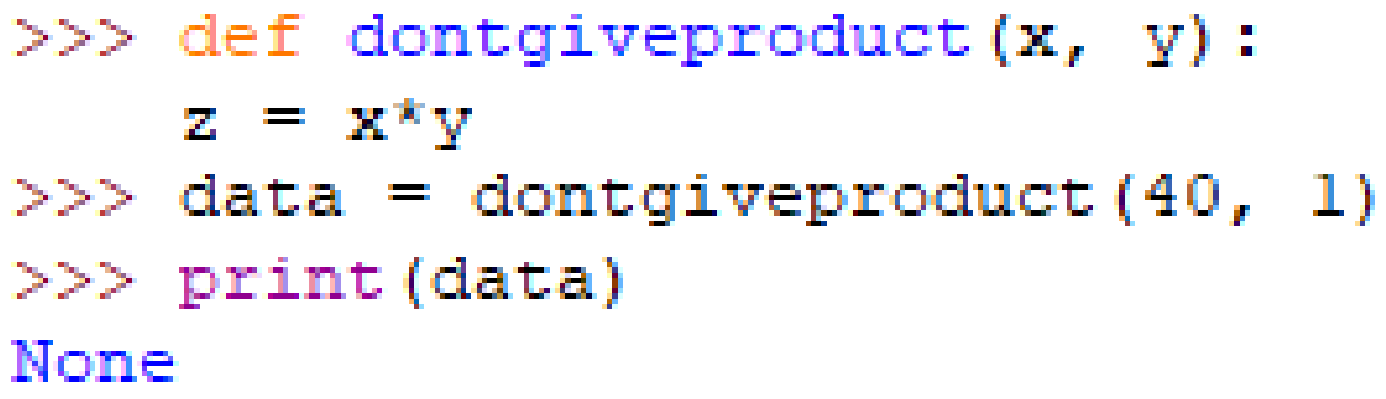
Note: you can pass calculated results into a function:



```
>>> giveproduct(20+20, 2-1)
40
```

Getting Data Out

If you don’t return something, the result of your function is a special type, called **`None`**.

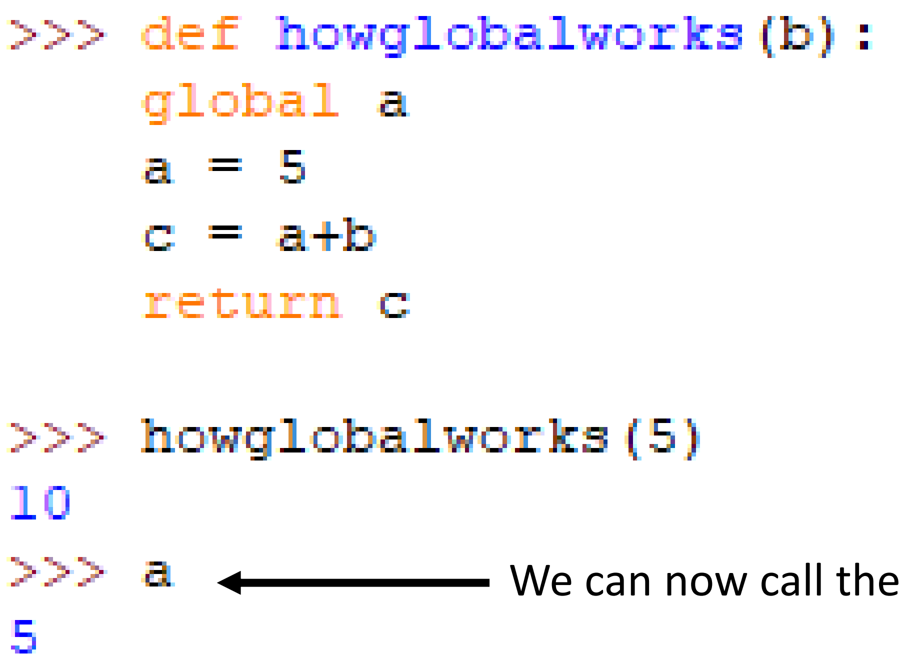


```
>>> def dontgiveproduct(x, y):
        z = x*y
>>> data = dontgiveproduct(40, 1)
>>> print(data)
None
```

Getting Data Out

You cannot affect variables in your main code from your function…

UNLESS you label the variable inside your function as a **`global`** variable.



```
>>> def howglobalworks(b):
    global a
    a = 5
    c = a+b
    return c

>>> howglobalworks(5)
10
>>> a
5
```

We can now call the **global** variable **a** from outside the function.

You must define your global variables:

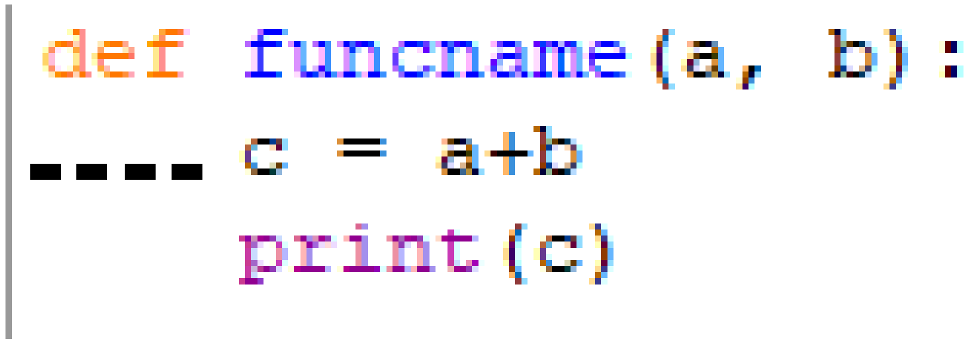
`global a`

You may define multiple global variables at once by separating them with commas:

`global a, fish, orderid`

Using Functions: Conventions & Rules

- Indents under the convention begin with four spaces
    - If you are using a text editor other than IDLE, make sure the automatic indent is 4 spaces
    - Tabs are not spaces! Sometimes text editors will default to using tabs instead of spaces when you press your [Tab] key

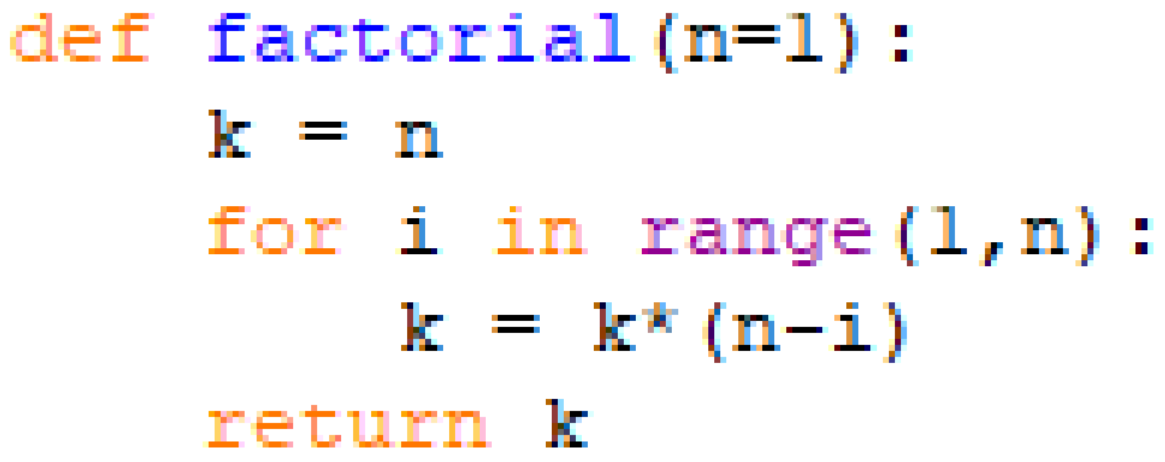


```
def funcname(a, b):
----c = a+b
    print(c)
```

COLUMBUS STATE
WORKFORCE INNOVATION

Using Functions: Conventions & Rules-Default Values

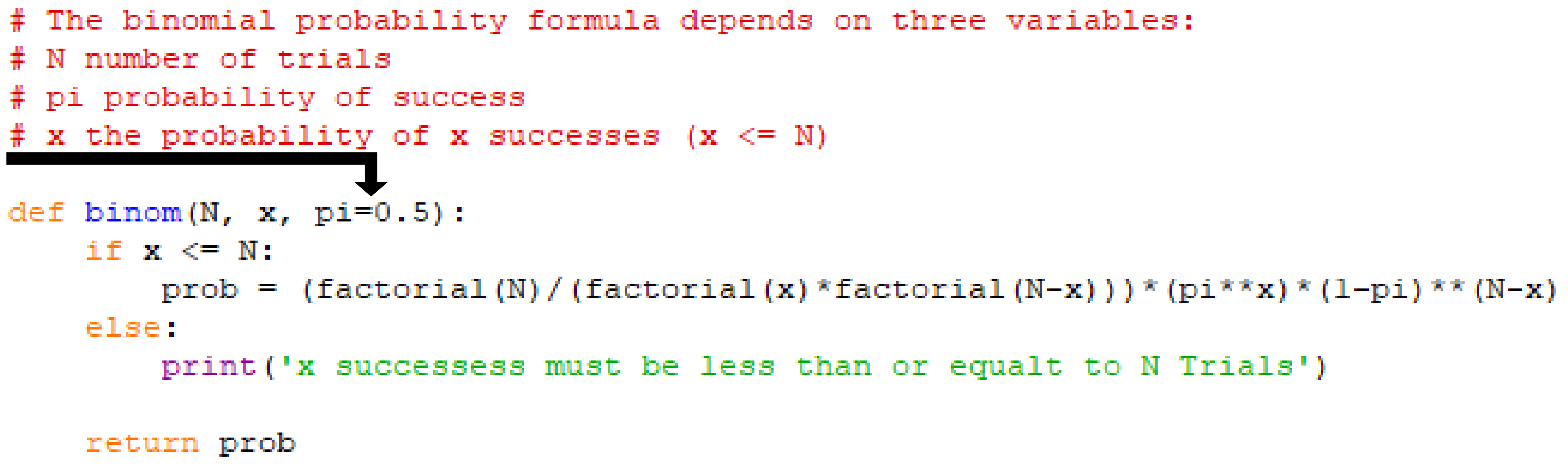
You can set default values in your functions.



```
def factorial(n=1):
    k = n
    for i in range(1,n):
        k = k*(n-i)
    return k
```

Using Functions: Conventions & Rules, Multiple Defaults

Defaults **MUST** be grouped as the right-most parameters in the function header.



```
# The binomial probability formula depends on three variables:
# N number of trials
# pi probability of success
# x the probability of x successes (x <= N)

def binom(N, x, pi=0.5):
    if x <= N:
        prob = (factorial(N)/(factorial(x)*factorial(N-x)))*(pi**x)*(1-pi)**(N-x)
    else:
        print('x successess must be less than or equalt to N Trials')

    return prob
```

Using Functions: Best Practices…

… **AND**, requirements for our class!

1. Functions Are Atomic (They only do ONE "thing")
2. Functions Return The Same Result, Given the Same Arguments ("Input")
3. They Do Not Have Unnecessary Side Effects

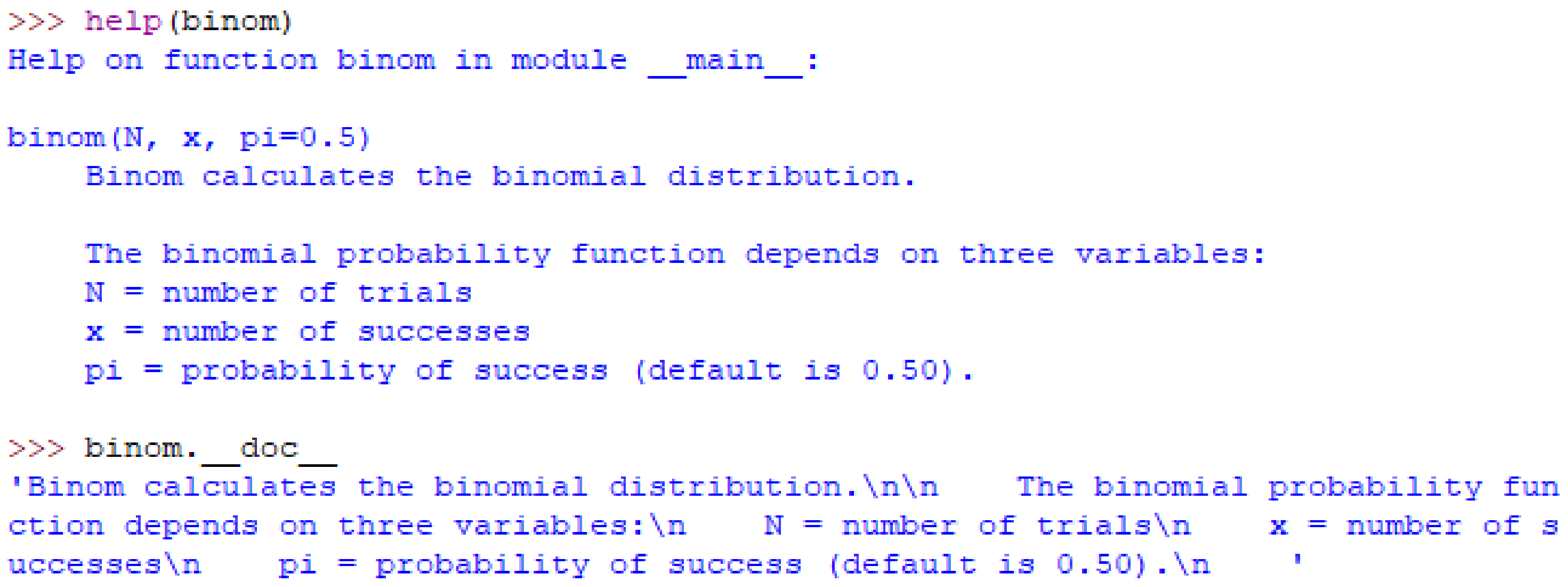
Docstrings in Functions

```
"""Docstrings begin with a capital letter and end with a period on the
first line.  After that, a blank line, followed by exposition as required."""
```

Docstrings called: Two Options

Docstrings can be called one of two ways.

- **`help(function)`**
- **`function.__doc__`**



```
>>> help(binom)
Help on function binom in module __main__:

binom(N, x, pi=0.5)
    Binom calculates the binomial distribution.

    The binomial probability function depends on three variables:
    N = number of trials
    x = number of successes
    pi = probability of success (default is 0.50).

>>> binom.__doc__
'Binom calculates the binomial distribution.\n\n    The binomial probability fun
ction depends on three variables:\n    N = number of trials\n    x = number of s
uccesses\n    pi = probability of success (default is 0.50).\n    '
```

Call functions from other programs

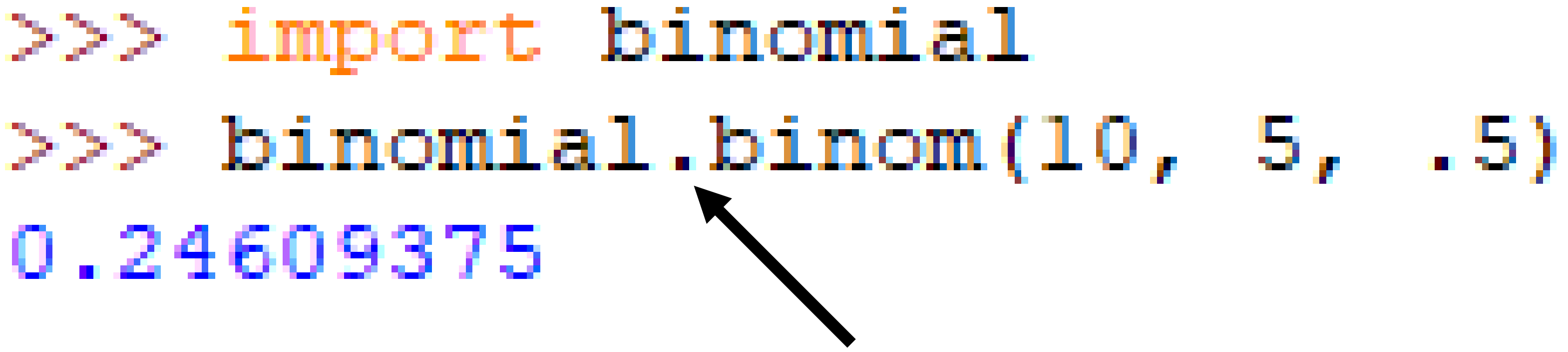
- MODULES

- We create modules by creating .py files.
- We can also create a file in another language and call it using Python.

- We’ve already been creating modules since Unit 1!

Call functions: Import statement

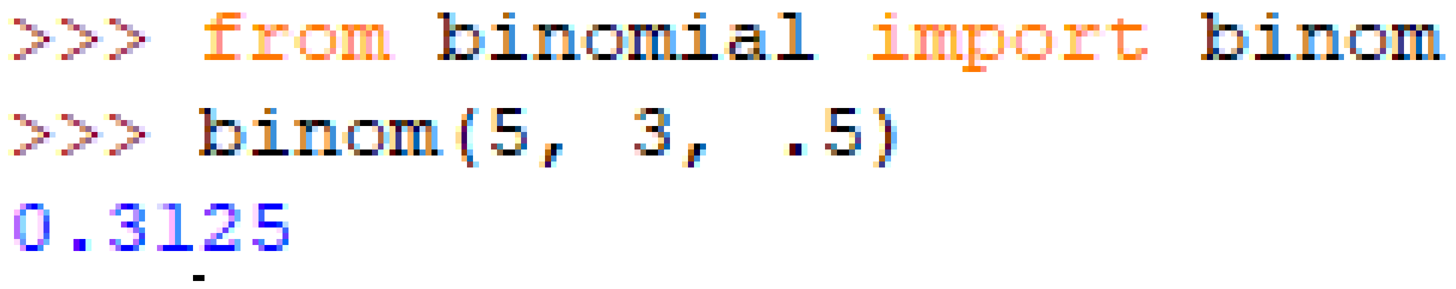
We import other modules by using the **`import`** statement. When we do this, we use the “dot notation” to call functions from the module …



```
>>> import binomial
>>> binomial.binom(10, 5, .5)
0.24609375
```

Call functions: from syntax

… or we can import functions and variables from other modules using the **`from`** syntax:



```
>>> from binomial import binom
>>> binom(5, 3, .5)
0.3125
```

About Module Usage…

- Modules must be in the same directory as your program, or in one of the directories listed in the system path.
- Modules should contain a "**`main()`**".
  - If the Module will be used on its own and used for importing, then the if statement shown below is necessary.
  - However, if the Module will only be used on its own or it will only be used for importing, then it is ok to omit the if statement shown below.

```
if __name__ == "__main__":
    main()
```

Glossary: Keywords & Terms

Keywords

`def`

denotes the beginning of a function definition

`global`

defines a variable such that it can be accessed from anywhere, not just it's current scope

`import`

binds a module to the local scope allowing code from outside the program to be used

`from`

allows access to a module to selectively import only necessary functions.

Example: `from my_module import my_function, my_other_function`

Terms

parameter

variables defined in the function definition

argument

variables passed to a function that match the defined parameters

module

a file containing python code

namespace

a collection of currently defined symbolic names (variables, functions, etc.); separates variables such that they don't overlap with one another.

Glossary: Operators, Special Variables

Operators

`" " "`

denotes a docstring used for code documentation

`():`

denotes the end of a function definition. Must be used in conjunction with the def keyword and doesn't require parameters.

Examples: `def my_function():`

```
def my_function(my_parameter):
```

Special Variables/Methods

`__name__`

defines the namespace the Python module is currently running in

`__main__`

defines the main function or "entry point" of the module

`__doc__`

will return the docstring that appears in the class or method

Function

`help(function)`

will return the docstring that appears in the class or method

COLUMBUS STATE
COMMUNITY COLLEGE

Practice Coding:
Class Activity