

# **Python Workshop Series Session 2: *Functions & Logic***

Mea Trahan  
Research Computing

Slides: [https://github.com/ResearchComputing/Python\\_Fall\\_2021](https://github.com/ResearchComputing/Python_Fall_2021)



Research Computing  
UNIVERSITY OF COLORADO **BOULDER**

**Be Boulder.**

**Official Python 3  
Documentation:  
<https://docs.python.org/3/>**

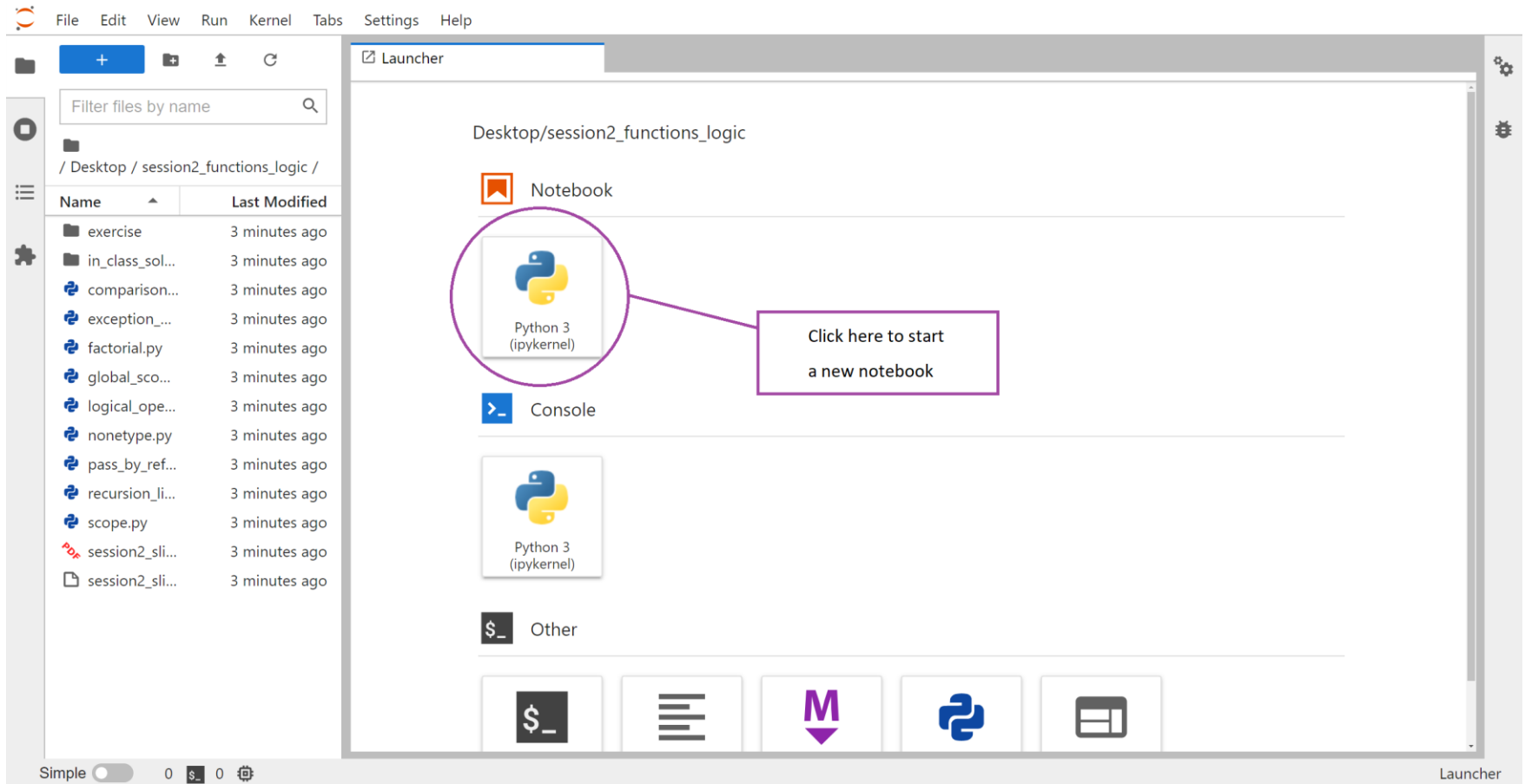


# JupyterLab

- Today's workshop has various code samples
- I suggest cutting and pasting them into the Jupyter notebook
- Recall that to open the notebook:
  - Access your shell (“anaconda prompt” in Windows)
  - Type: `conda activate pyclass21`
  - Type: `jupyter lab` ← note the “Y”
  - Follow along
- Note: to close the notebook, close your browser and then type `ctrl+c`



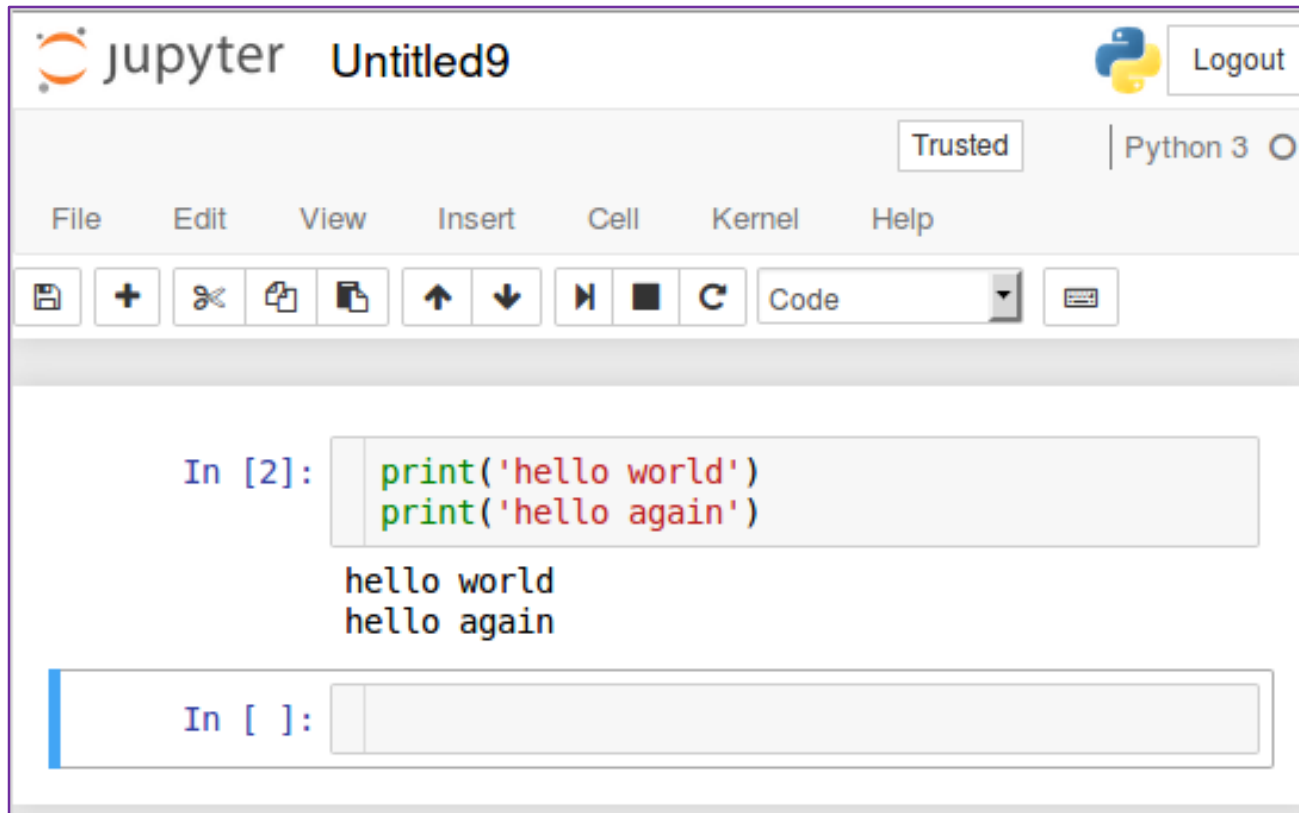
# The Jupyter Interface



- Jupyter supports different interactive notebook types (e.g., R, Python 2.x etc.)
- Start a Python 3 notebook



# The Jupyter Interface



- Pressing 'enter' starts a new line
- Pressing 'shift' + 'enter' executes all lines of code within a cell



# Outline

- Functions
- Conditionals
- Recursion
- Exception Handling



# Defining a Function

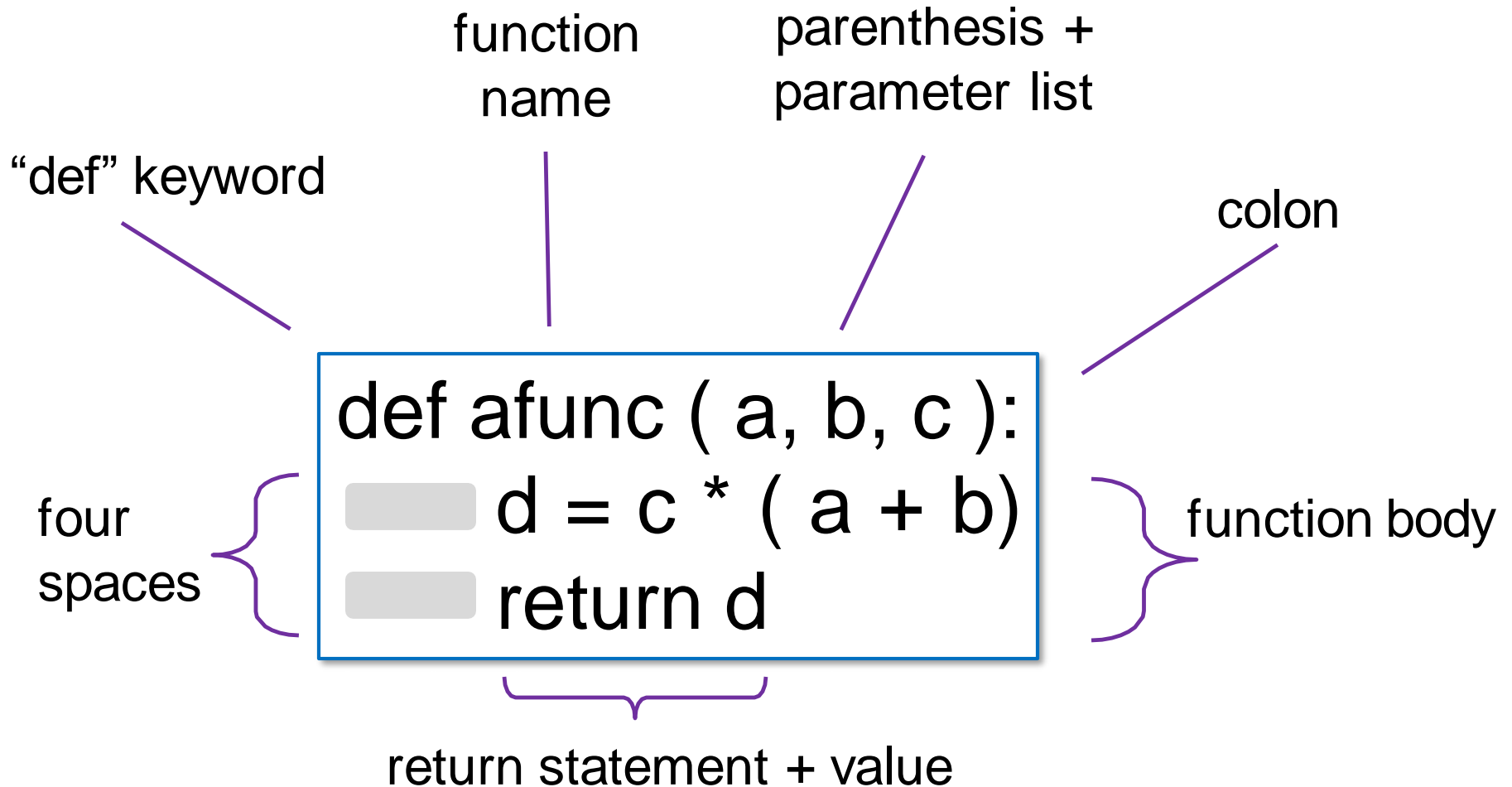
- Functions must be defined before they are called
- Definition example:

```
def afunc ( a , b , c ):  
    d = c * ( a + b )  
    return d
```

- Several things to note...



# Defining a Function





# Calling Functions

```
def afunc ( a , b , c ):
```

```
    d = c * ( a + b)
```

```
    return d
```

```
myval = afunc( 1 , 2 , 4 )
```

- Functions may be called once defined
- Value of **d** assigned to **myval** via return statement



# Exercise 1

- Write a function that :
  1. accepts two parameters
  2. returns the difference of those two parameters.
- Test it out with various parameter combinations

Sample function definition  
and calling syntax

```
def afunc ( a , b , c ):  
    d = c * ( a + b )  
    return d  
  
myval = afunc( 1 , 2, 4 )
```



# Exercise 2

- Write a function that accepts two parameters:
  - name : a string value
  - age : an 'int' value

```
def afunc ( a , b , c ):  
    d = c * ( a + b )  
    return d  
  
myval = afunc( 1 , 2, 4 )
```

Sample function definition / calling syntax

- It should return:
  - msg : a string with value "{name} is {age} years old."
- Hint: use the "str" type conversion function



# Multiple Return Values

```
def afunc ( a , b , c ):
```

```
    d = c * ( a + b )
```

```
    e = c * ( a - b )
```

```
    return d , e
```

```
myval1, myval2 = afunc( 1 , 2 )
```

- Multiple scalar values may be returned.
- Separate values with commas
- $d \rightarrow \text{myval1}$       $e \rightarrow \text{myval2}$



# The NoneType Class

```
def afunc ( a ):  
    print ( a )
```

```
afunc(2)  
g = afunc(2)  
print(g)
```

Open and run “nonetype.py”

- Functions need not return a value
- Even if no “return” statement, functions will return Nonetype
- NoneType:
  - empty datatype
  - print() displays “None”



# Optional Parameters

```
def afunc ( a, b, c = 1):  
    d = c * ( a + b )  
    return d  
  
var  = afunc( 1, 2 )  
var2 = afunc( 1, 2 , c=2 )
```

- Optional parameters specified by indicating default value
- **c** does not have to be passed to *afunc*
- Defaults to value of 1



# Optional Parameters

```
def afunc ( a , b = 1 , c = 1):  
    d = c * ( a + b)  
    return d
```

- Optional parameters can be specified implicitly by position (no “=” needed)

afunc(3,b=2)   afunc(3,2,c=1)   afunc(3,2,1)

afunc(3,2)   afunc(3,b=2,c=1)   *equivalent function calls*



# Pass by Value or Reference?

- General rule of thumb
  - Scalar variables **behave** as though passed by value
  - Most everything else is passed by reference
- In reality, everything is passed by reference.
- Scoping rules dictate behavior of assignment etc.
- Open and run [pass\\_by\\_reference\\_or\\_value.py](#)





# Scope

- Scope behaves more or less intuitively in Python.
- Variables defined within a function are invisible to the program unit that called the function.
- When a name is used in a function, it is resolved using the nearest enclosing scope...
- ... i.e., the block of code that defined the function – then on up the chain
- Open and run “scope.py”



# Scoping Gotcha!

Try this...

```
def func():  
    print(a)
```

```
a = 1  
func()
```

... and this...

```
def func():  
    a+=1
```

```
a = 1  
func()
```

What gives?  
Examine scope.py.



# Scoping Gotcha!

*Python Documentation:* If a *name binding operation* occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block.

In other words, assignment always creates a new **local** variable.

```
def func():  
    a = a+1
```

equivalent



```
a = 1  
func( )
```

```
def func():  
    func_a = func_a+1
```

```
a = 1  
func( )
```

*The problem:* `func_a` is referenced before it has been assigned a value.



# Global Variables

- Set variables you wish to be global in the top-level namespace (effectively the main-program area in our examples so far)
- If you want a function to modify a global variable, declare the variable as **global** inside the function
- The value from the top-level or *builtins* (technical) namespaces will be used.
- Unlike local variables, no other namespace (e.g. containing functions) will be searched.
- Examine **global\_scope.py**

```
def func( ):
    global a
    a = a+1

a = 1
func( )
print(a)
```



# Logical Operators

- Boolean expressions have value True or False
  - Note the capital 'T' and 'F'
  - true and false are not Boolean values in Python
- Boolean values can be combined to yield a Boolean expression via logical operators:
  - and
  - or
  - not
- Open and run logical\_operators.py



# Comparison operators

- Numeric values can be combined to yield a Boolean expression via comparison operators:
  - `==` “equals”
  - `!=` “not equal”
  - `>` “greater than”
  - `>=` “greater than or equal to”
  - `<` “less than”
  - `<=` “less than or equal to”
- Open and run `comparison_operators.py`
- The `==` and `!=` also work with string variables



# Conditionals 1: if

- Syntax is similar to function definition syntax:

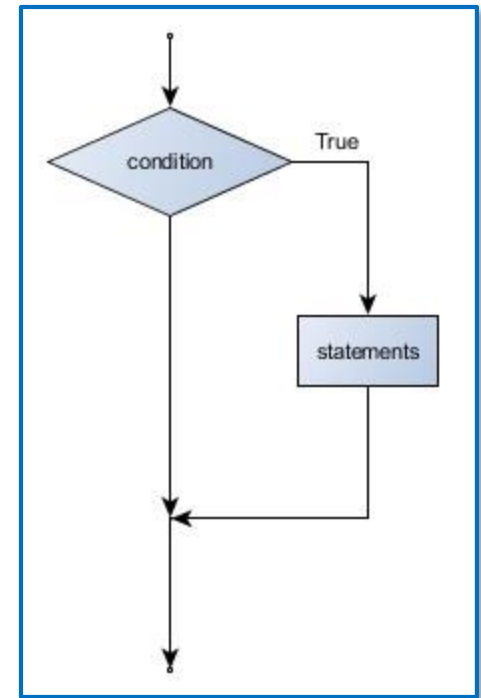
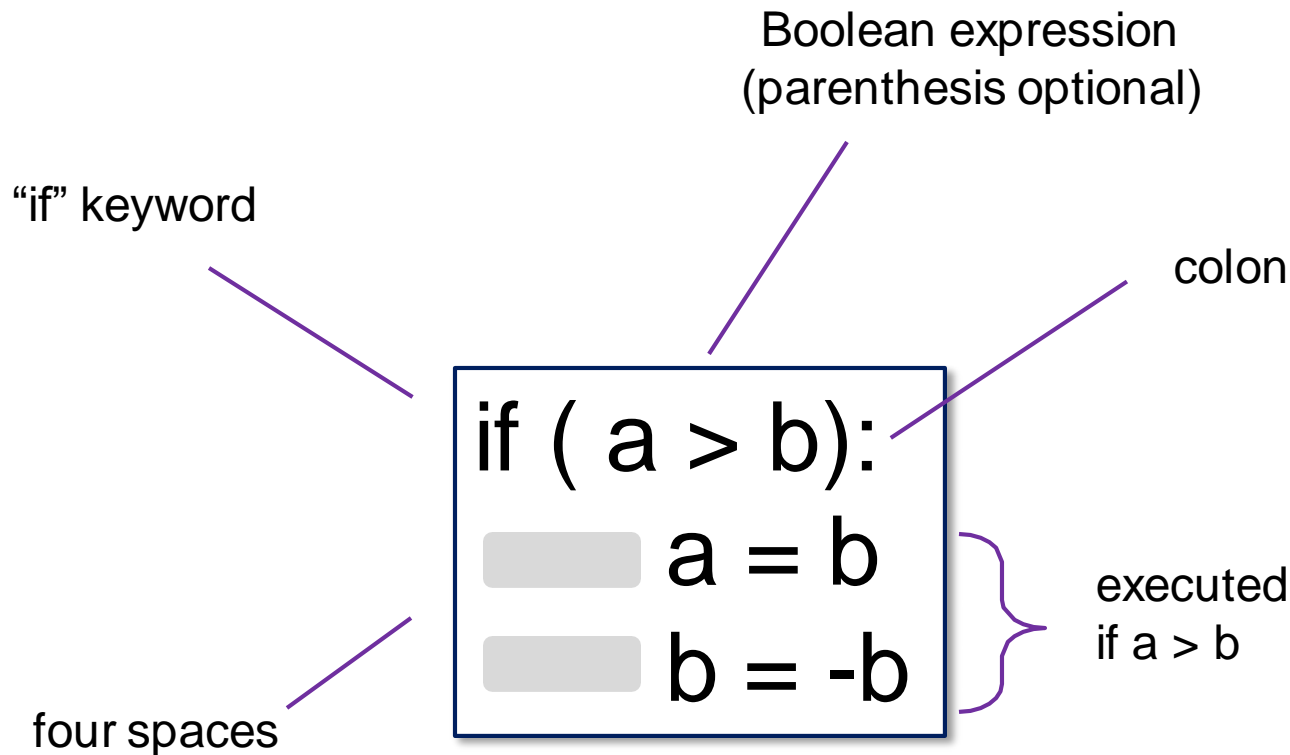


image credit: HTLCS



# Exercise 3

- Write a function named *ispositive* that:
  - Accepts a single, numeric parameter
  - uses if (without else) to return:
    - True if the input parameter is positive.
    - False otherwise

```
def ispositive(a):  
     if ( expr ):  
          statement 1  
         statement 2
```





# Conditionals 2: if / else

- Can add an “else” clause to our if statement

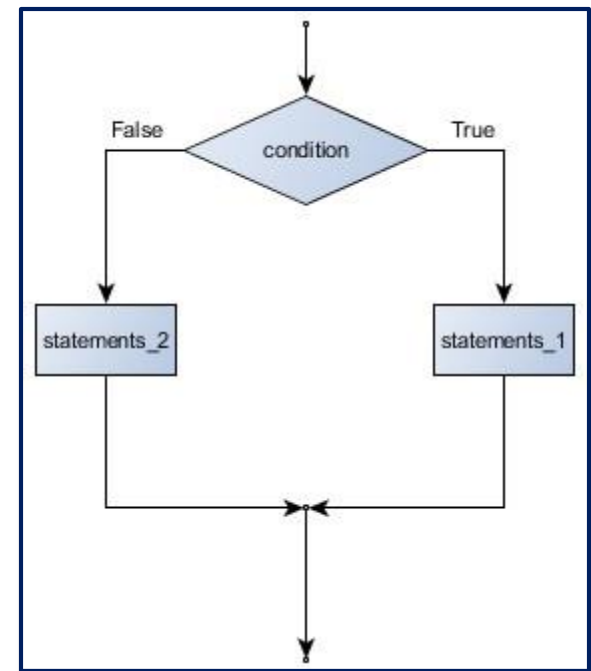
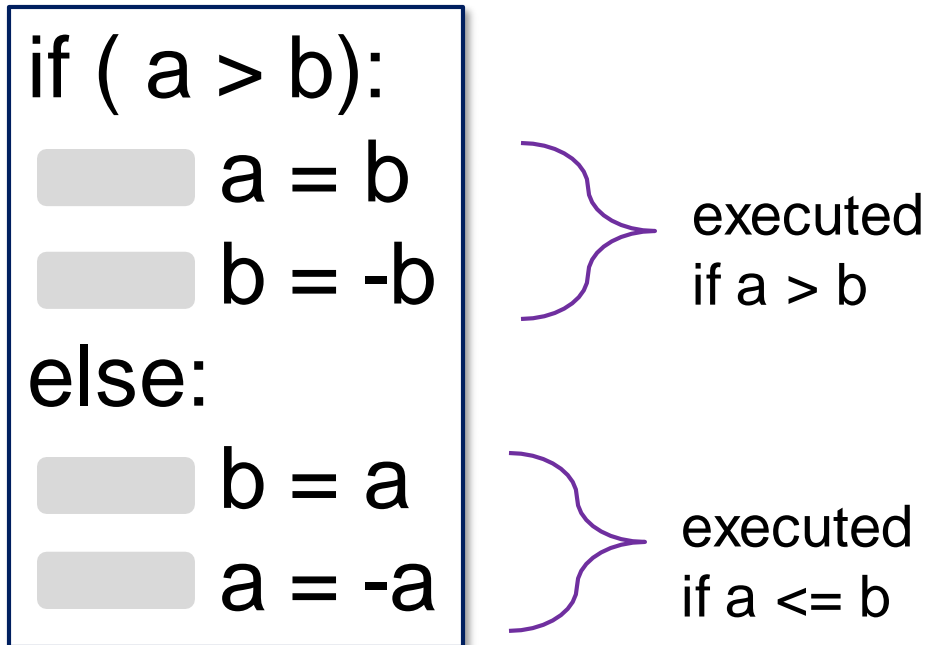


image credit: HTLCS



# Conditionals 3: elif

- Can also add an else-if clause(s) via “elif”

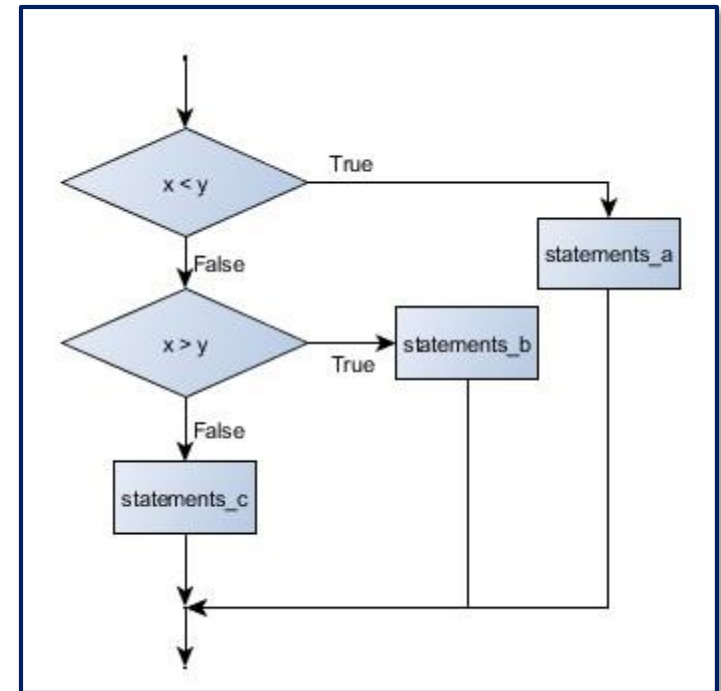
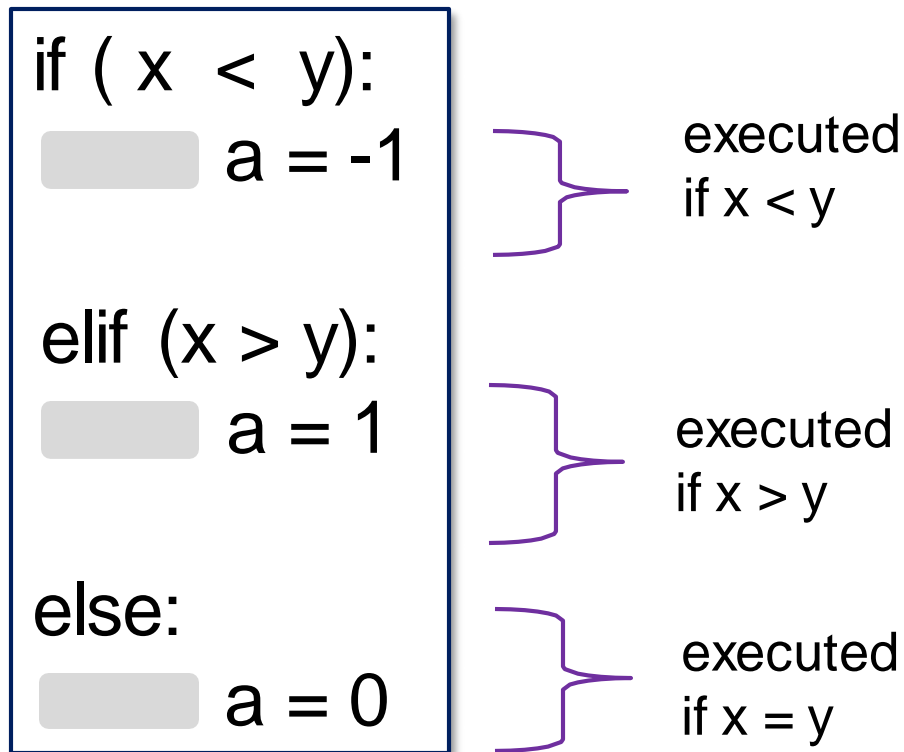


image credit: HTLCS



# Exercise 4

- Using if / elif / else write a function that takes a number between 0 and 100 and returns the associated letter grade.
- e.g, grade(75) will return 'C'

```
def ispositive(a):  
    if ( expr ):  
        statement 1  
    statement 2
```

*function definition syntax*

```
if ( x < y):  
    a = -1  
elif (x > y):  
    a = 1  
else:  
    a = 0
```

*elif syntax*



# Recursion in Python

- Python allows the user to define recursive functions.
- No extra keywords needed.
- The function is recursive by virtue of calling itself:

```
def afunc ( parameters):  
    if ( expr):  
        statement group 1  
        return something  
    else:  
        statement group 2  
        afunc(new parameters)
```



# Recursion Example: Factorial

- Open factorial.py

```
def factorial(n):  
    if (n <= 1):  
        return 1  
    else:  
        return n*factorial(n-1)
```

- Quick exercise:  
copy/modify to compute sum of numbers 1 through n
- Test these numbers:
  - 10
  - 100
  - 1000
  - 10000



# Recursion Depth

- Python has a maximum recursion depth. Code will crash if reached.
- Can set via `sys.setrecursionlimit`
- Try it!
- Useful sometimes
- Generally inefficient; use loops
- Also: `sys.getrecursionlimit`

```
import sys
def factorial(n):
    if (n <= 1):
        return 1
    else:
        return n*factorial(n-1)

mstr = input("Enter a number: ")
m = int(mstr)
sys.setrecursionlimit(m+2)
print('m! is' , factorial(m) , '.')
```



# Exception Handling

- Occasionally, you may get some wonky input.
- The program doesn't have crash: use try/except
- Open `exception_handling.py`

try:

    [ ] thing you want to do

except:

    [ ] thing to do if that fails  
remainder of program



# Next Time

- Lists, tuples, and dictionaries
- Iteration
- Slides: [https://github.com/ResearchComputing/Python\\_Fall\\_2021](https://github.com/ResearchComputing/Python_Fall_2021)
- Survey: <http://tinyurl.com/curc-survey18>

