MONASH University

MONASH
INFORMATION
TECHNOLOGY

# FIT9136: Algorithms and programming foundations in Python

## Week 3:
## Control Structures & Function

GROUP OF EIGHT AUSTRALIA

# Week 3 Learning Outcomes

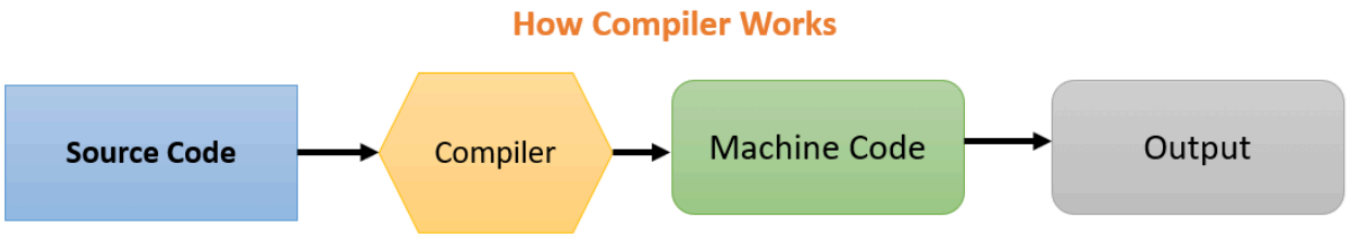
- After working your way through this week, you should be able to:
  - Identify and use various programming constructs used in Python
  - Modularize your program into functions
  - Efficiently write simple Python program with the correct grammar

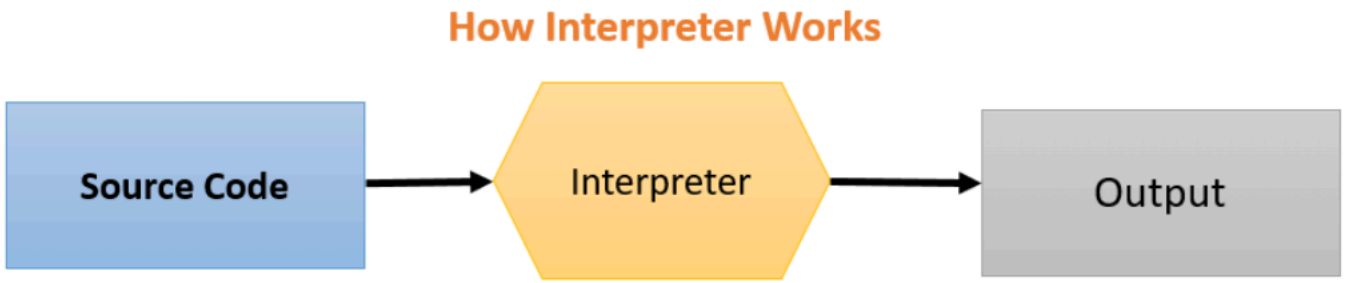
MONASH University

# Basic Elements of Python: Statements and Assignments

# Compiler vs Interpreter

- What is program?
  - In computing, a program is a specific set of ordered operations for a computer to perform.

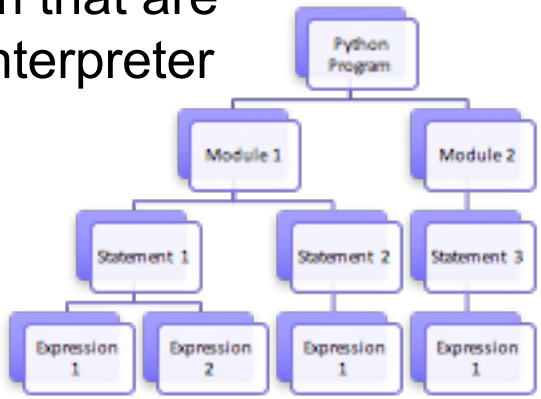



Compiler vs Interpreter

# Python Statements

- Statements:
  - Instructions (commands) of a Python program that are interpretable and executable by the Python interpreter




- Assignment statements:
  - Binding a data object (representing specific type of value) to a variable
  - Assigning the result of an expression to a variable

```
message = "Welcome to FIT9133"
temp_F = temp_C * 9 / 5 + 32
bool_result = value > 0 and value < 100
```

# Single-Line Statements

- Single-line statements:
  - Each Python statement spans a single line of code

- You could split a single line statement across multiple lines
  - Append a backslash ('\') at the end of each line

```
bool_result =  value > 0 \
               and value < 100 \
               and value % 5 == 0
```

https://www.python.org/dev/peps/pep-0008/#maximum-line-length

# Statement Blocks

- Statement blocks:
  - Certain programming constructs that span across multiple lines of code
  - Control structures that determine the flow of program execution

```
flag = bool(input("I love programming. True/False?"))

if flag == True:
    print("YES")
    print("It is true!")

else:
    print("NO")
    print("It is false!")
```

- Important: the symbol ':' denotes the beginning of a statement block, i.e. the subsequent *indented* statements are part of that block
- Statement blocks are usually concluded with a new blank line

Indentation is semantically meaningful in Python programs.

# Control Structures

# Selection Constructs

- **`if-else`** statements:
  - Using logical expressions as selection conditions to determine which statement block to be executed

```
if the condition is True:
    do this statement block
else:
    do this statement block
```

- Note: Indentation is important in defining the "scope" of a block of statements.

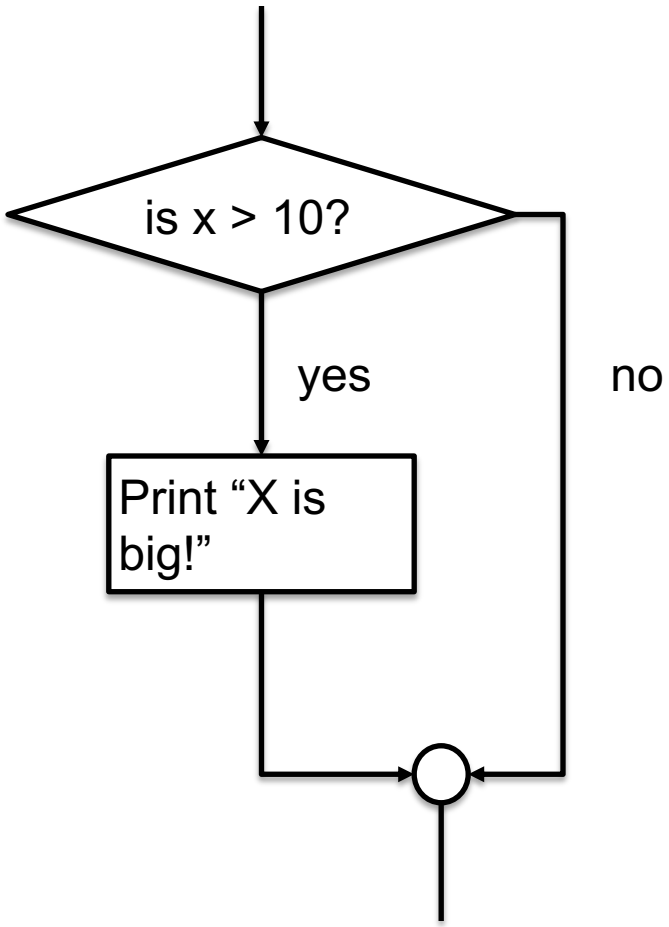
```
message = "Welcome to FIT9136"
letter = 'o'
count = message.count(letter)
if count < 1:
    print(letter + " doesn't exist in " + message)
else:
    print(letter + " exists in " + message)
    print(letter + " occurs " + str(count) + " times")
```

# Selection Constructs

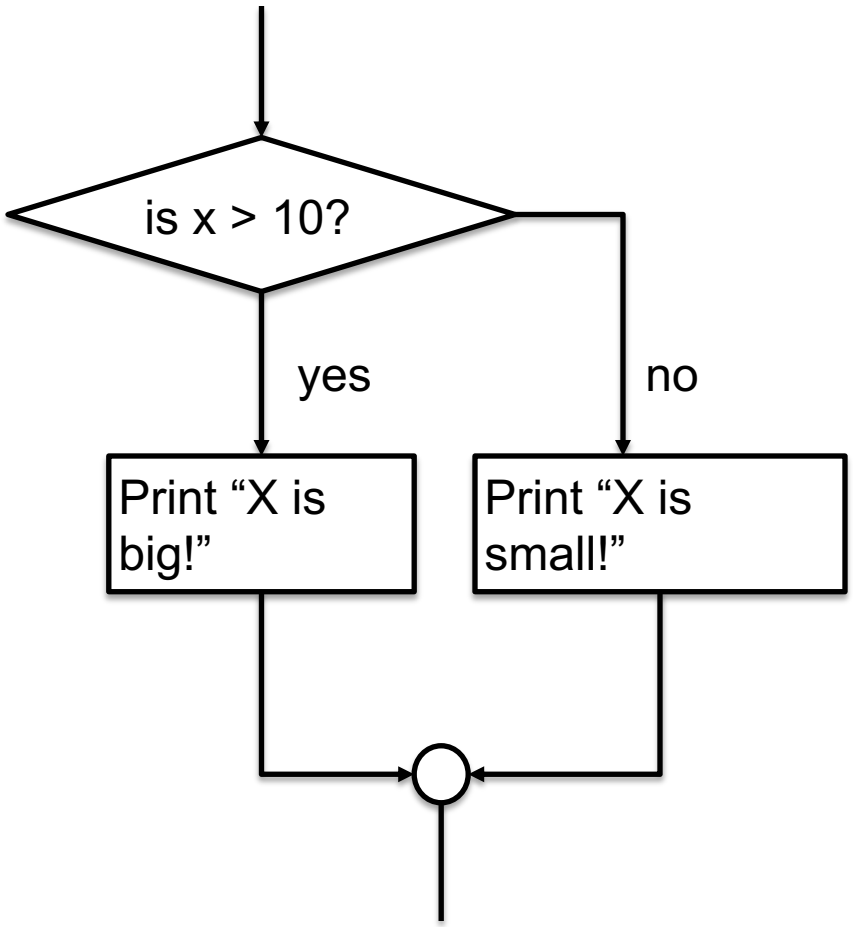
- **`if`** statements:
  - Not all logic requires an action when a condition is false. In this case, **`else`** can be omitted.

```
if the condition is True:
    do this statement block
```

# If statement



# If-else statement



- Nested **`if`** statements:
  - Useful for when multiple conditions need to be considered

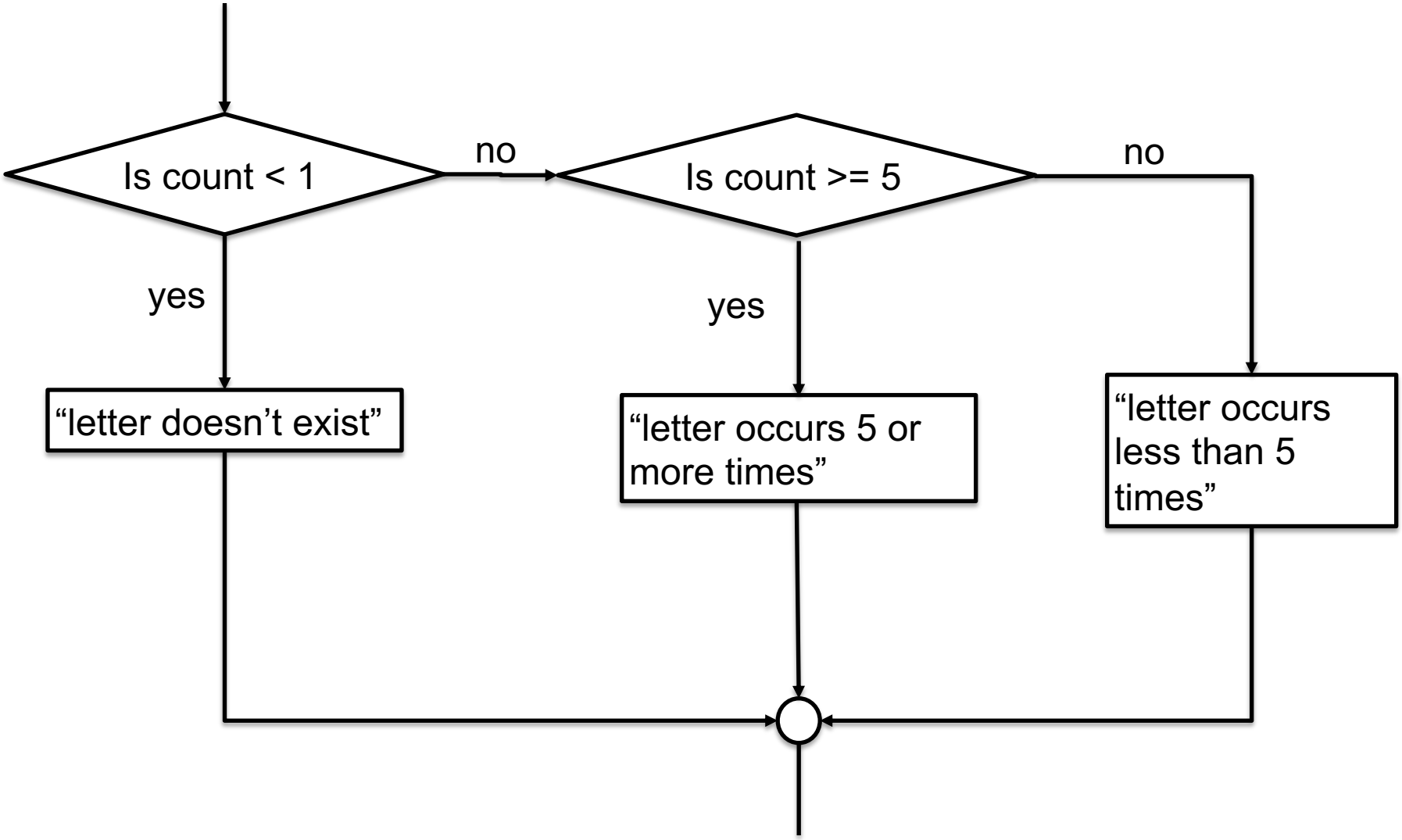
```
message = "Welcome to FIT9133"
letter = 'o'
count = message.count(letter)
if count < 1:
    print(letter + " doesn't exist in " + message)
else:
    print(letter + " exists in " + message)
    if count >= 5:
        print(letter + " occurs 5 times or more")
    else:
        print(letter + " occurs less than 5 times")
```

# Selection Constructs

- **elif** statements:
  - Similar to nested **if** statements; combining an **if** with an **else**

```
message = "Welcome to FIT9133"
letter = 'o'
count = message.count(letter)
if count < 1:
    print(letter + " doesn't exist in " + message)
elif count >= 5:
    print(letter + " exists in " + message)
    print(letter + " occurs 5 times or more")
else:
    print(letter + " exists in " + message)
    print(letter + " occurs less than 5 times")
```

# If-else statement



# Question

- Try writing the following in Python code:
  - If x is odd and y is even print “yay”

```
x = int(input(“What is x?”))
y = int(input(“What is y?”))
???
```

# Question

- Try writing the following in Python code:
  - If x is odd and y is even print “yay”

```
x = int(input(“What is x?”))
y = int(input(“What is y?”))

if x % 2 == 1:
    if y % 2 == 0:
        print(“yay”)
```

# Question

- However, we can use **and** from last lecture:

```
x = int(input("What is x?"))
y = int(input("What is y?"))

if x % 2 == 1 and y % 2 == 0 :
    print("yay")
```

# Iteration Constructs

- **`while`** loop:
  - A block of statements will be executed repeatedly as long as the governing condition is **`True`**

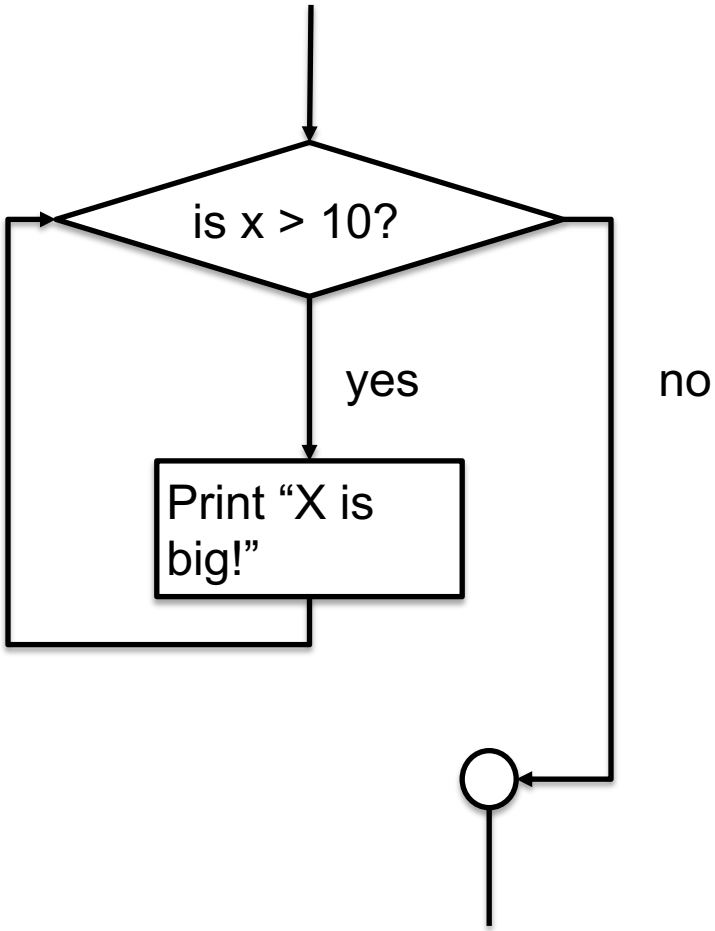
```
while the condition is True:
    do this statement block
```

- Note: The governing condition (logical expression) has to turn into **`False`** eventually; otherwise the loop will run *infinitely*.

```
number = 0
while number < 5:
    number += 1
    print(number)
```

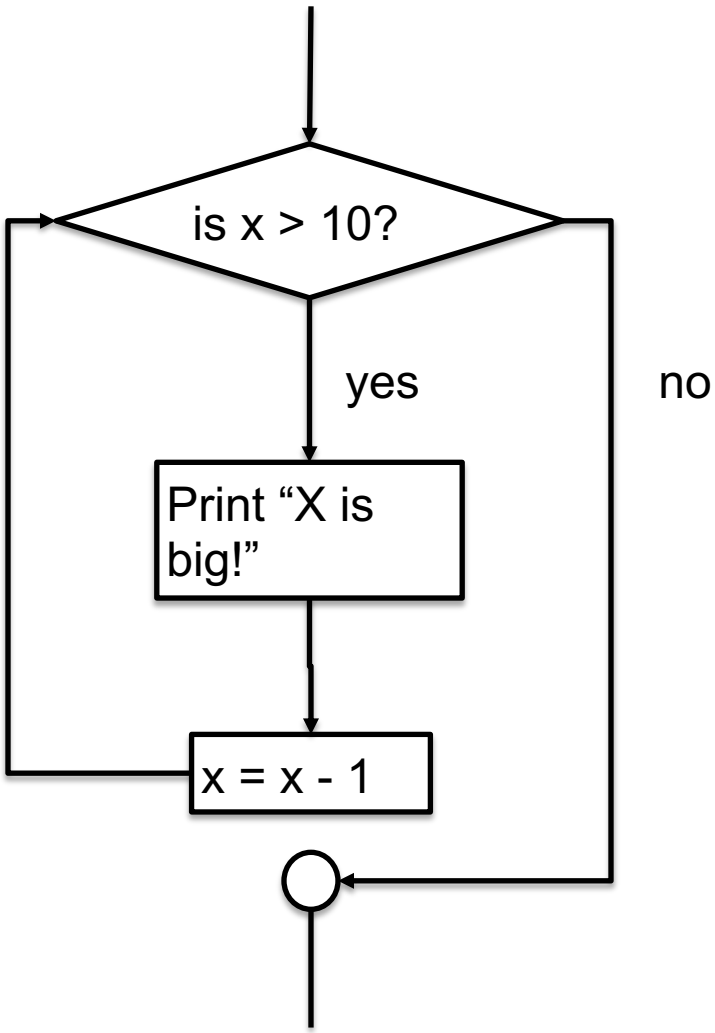
http://www.pythontutor.com/visualize.html

# Iteration constructs – While




```
……
while(x > 10):
        print(“X is big!”)
```

# Iteration constructs – While




```
……
while(x > 10):
        print(“X is big!”)
        x = x - 1
```

How many prints will we see if x = 13?

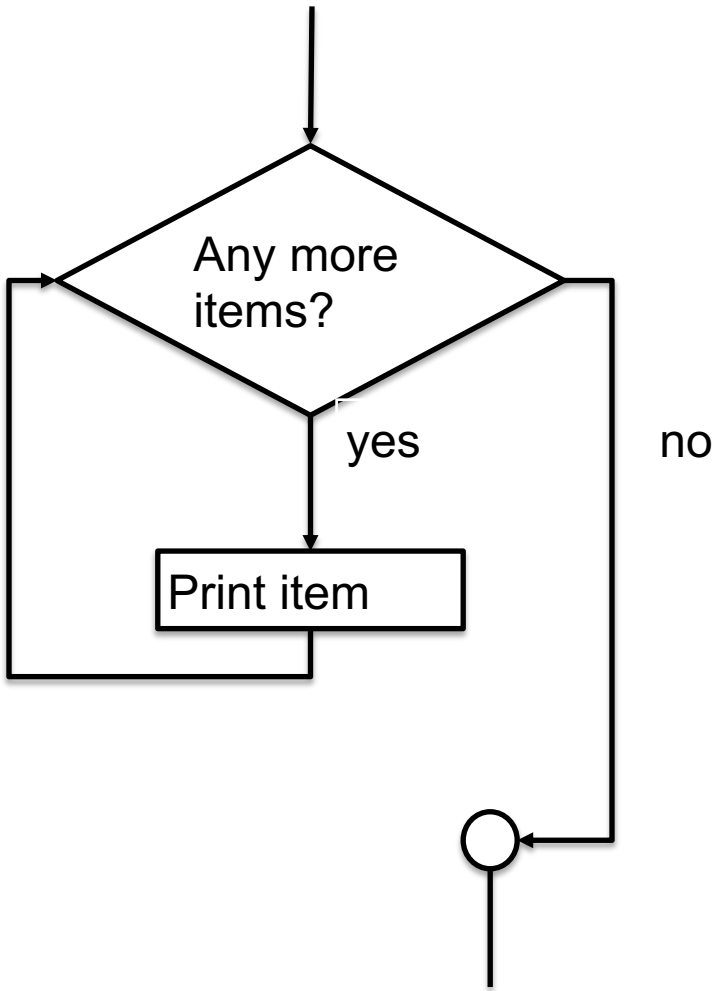
# Iteration Constructs

- **for** loop:
  - Similar to **while** loop; except that the governing condition (logical expression) does not need to be defined
  - Useful for iterating or traversing through a collection of items (e.g. Lists)

```
numbers = "12345"
product = 1
for item in numbers:
    product *= int(item)
print(product)
```

- Note: This special **for** loop structure with the **in** operator can be used on any *iterable* collective data type (e.g. String, List etc.).

# Iteration constructs - For




```
numbers = "12345"
for item in numbers:
    print(item)
```

# Indentation

# Indentation

- What is the indentation?
  - **Space** or **Tab** is both OK as indentation
  - 4 space or 1 tab is widely used in Python community
  - Tab:

```
with open(outputFile, "w") as fw:
	fw.write("GUIID,Element_Type,X1,Y1,X2,Y2,R,G,B,Score,Install,Category,Developer\n")

	for subdir, dirs, files in os.walk(inputDir):
		countValidFile = 0
		countAllFile = 1
		countUnduplicates = 0

		if "output" in subdir and subdir.endswith("ui") and "stoat_fsm_output" in subdir:
			path = os.path.normpath(subdir)
			pathComponent_list = path.split(os.sep)
			packageName = pathComponent_list[-3].split("_")[0]
```

  - Space

```
with open(outputFile, "w") as fw:
    fw.write("GUIID,Element_Type,X1,Y1,X2,Y2,R,G,B,Score,Install,Category,Developer\n")

    for subdir, dirs, files in os.walk(inputDir):
        countValidFile = 0
        countAllFile = 1
        countUnduplicates = 0

        if "output" in subdir and subdir.endswith("ui") and "stoat_fsm_output" in subdir:
            path = os.path.normpath(subdir)
            pathComponent_list = path.split(os.sep)
            packageName = pathComponent_list[-3].split("_")[0]
```

# Indentation

- Why indentation?
  - Python uses indentation as syntax, bad indenting causes crashes
  - Block code for easy understanding (its like paragraphs for an essay)

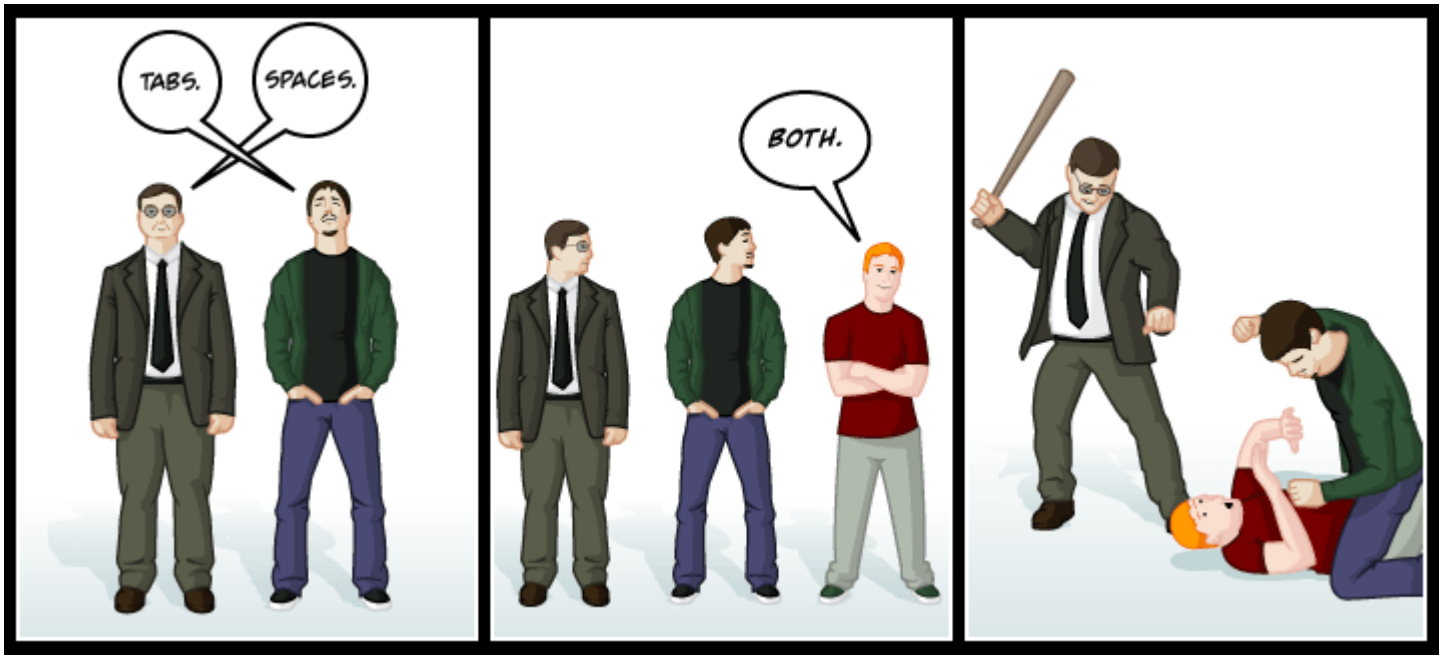
- Space or Tab?
  - **A Tab** could be a different number of columns depending on your environment (OS or editor), but also leading to small-size files.

  - https://www.youtube.com/watch?v=SsoOG6ZeyUI

  - PEP 8 -- Style Guide for Python Code
    - *"Use 4 spaces per indentation level"*
    - "*Spaces are the preferred indentation method. Tabs should be used solely to remain consistent with code that is already indented with tabs."*

# Indentation

- Please be consistent !!!
  - **Space** or **Tab**



# Termination of the loop: Continue, Break

# Continue

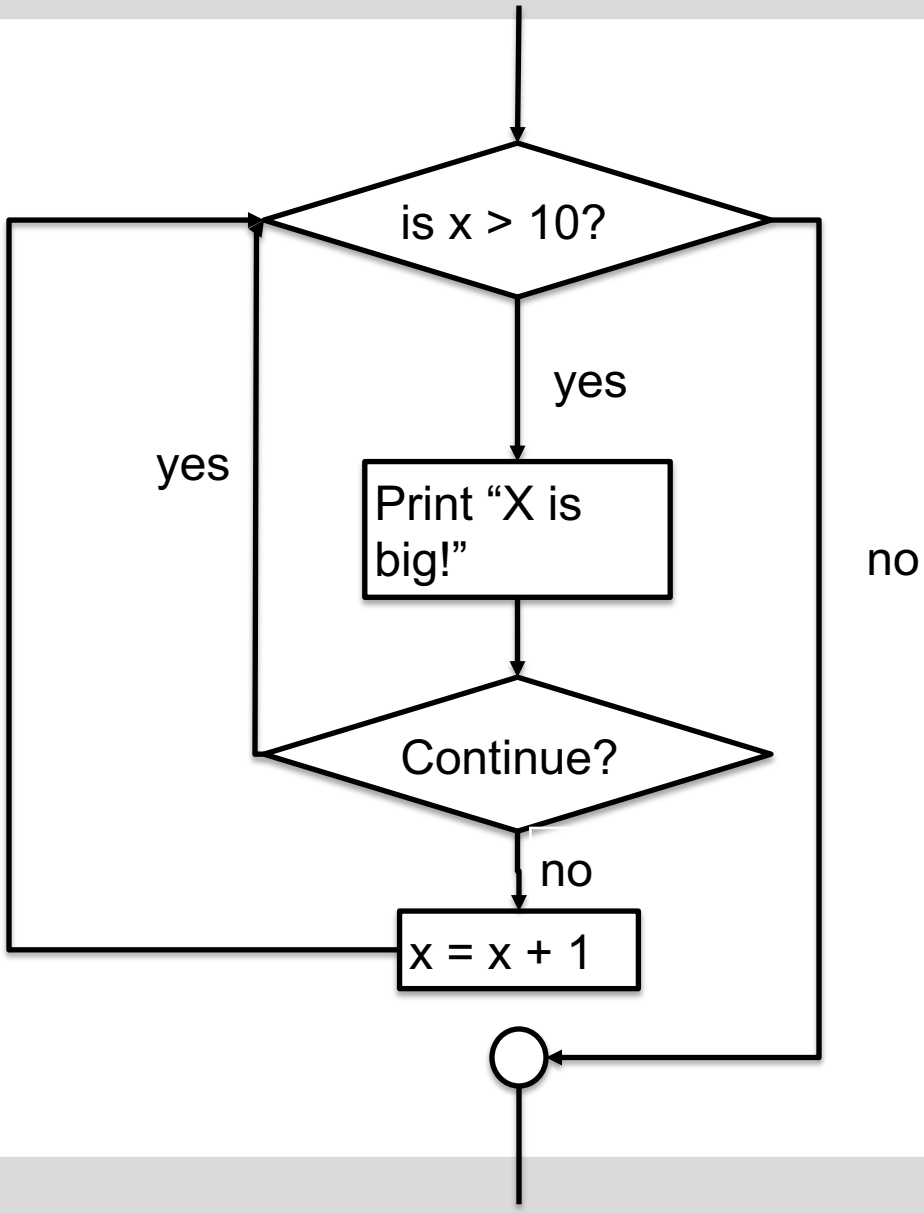
- The **continue** statement skips the current iteration of a <u>for</u> or <u>while</u> loop.

```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

---

```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

# Continue statement



# Continue

- Example about the **continue**:
  - Given a string and a target character, we want to remove the character of the string.

```
a_str = "helloWorld"
charToRemove = "e"
new_str = ""
for char in a_str:
    if char == charToRemove:
        continue
    new_str += char
print(new_str)
```

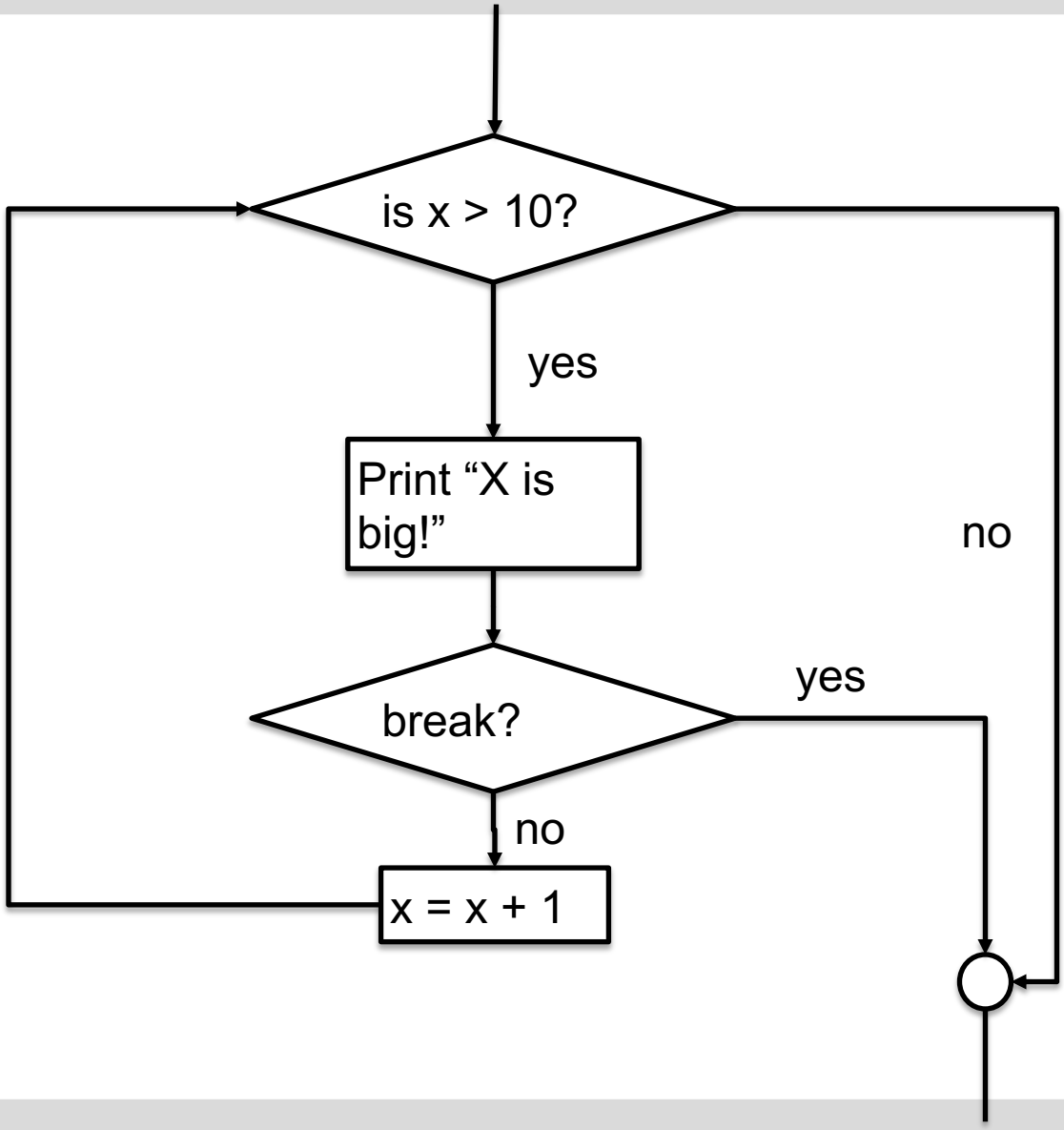
# Break

- The break statement terminates the loop containing it.

```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

# Break statement



# Break

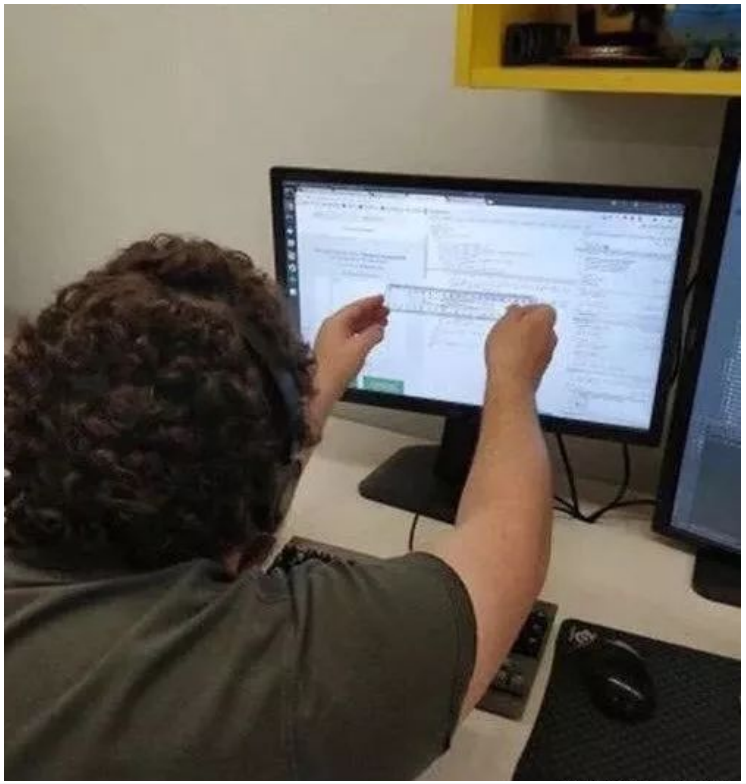
- Example about the **break**:
  - Given a list of numbers and a target number, we want to find that if the target number is in the list.
  - You can use **in** statement, but we want to use the for-loop and break in this example

```
number_list = [3, 11, 9, 7, 6, 5, 100, 20, 9, 6, 3, 1, 0]
target = 9
for number in number_list:
    if number == target:
        print("The target number is in the list")
        break
```

# A Reminder: Pay attention to Indentation

- When there are nested loops, please pay special attention to the usage of **continue** and **break** statement.
- Indentation can help you understand which loop you want to **continue** or **break**.

```
a_list = [1, 2, 3]
b_list = [2, 5, 6]
for itemA in a_list :
    for itemB in b_list:
        if itemA == itemB:
            break
        print(itemA, itemB)
```



# Functions

# Functions

Now that we have the ability to make useful code, how can we reuse it?

```
x = 5
y = 10

print(x,"+",y,"=",x+y)
print(x,"-",y,"=",x-y)
print(x,"/",y,"=",x/y)
print(x,"*",y,"=",x*y)
```

How can we reuse the above code for x = 9 and y = 4?

Functions allow for algorithms to be reused without the need to copy the code itself.

Call this algorithm **mathinfo**:

```
def mathinfo():
    x = 5
    y = 10

    print(x,"+",y,"=",x+y)
    print(x,"-",y,"=",x-y)
    print(x,"/",y,"=",x/y)
    print(x,"*",y,"=",x*y)
```

Doesn't work for x = 9 and y = 4 without editing. They should be passed to the function as attributes.

x and y are now positional arguments. x is in position 0 and y is in position 1.

```
def mathinfo(x, y):

    print(x,"+",y,"=",x+y)
    print(x,"-",y,"=",x-y)
    print(x,"/",y,"=",x/y)
    print(x,"*",y,"=",x*y)
```

# Let's compare copying vs calling:

```
def mathinfo(x, y):

    print(x,"+",y,"=",x+y)
    print(x,"-",y,"=",x-y)
    print(x,"/",y,"=",x/y)
    print(x,"*",y,"=",x*y)

mathinfo(5,10)
mathinfo(9,4)
```

```
x = 5
y = 10

print(x,"+",y,"=",x+y)
print(x,"-",y,"=",x-y)
print(x,"/",y,"=",x/y)
print(x,"*",y,"=",x*y)

x = 9
y = 4

print(x,"+",y,"=",x+y)
print(x,"-",y,"=",x-y)
print(x,"/",y,"=",x/y)
print(x,"*",y,"=",x*y)
```

# Example: A Simple Function

- Defining a function (without default values):
  - Take two parameters (arguments) and return a result

keyword

name

parameters or argument

specification, docstring

```
def addition(first_arg, second_arg):
    """
    Input: first_arg, second_arg, an int number
    Return the addition of two input number
    """
    result = first_arg + second_arg
    return result
```

body

```
sum = addition(1, 2)
```

function call

Let's look at another example from your practicals

```
def cel2far(temperature):
    print(temperature * (9/5) + 32)

cel2far(20)
```

What if I want to store the result of the function, rather than print it?

```
def cel2far(temperature):
    print(temperature * (9/5) + 32)

fahrenheit = cel2far(20)
print(fahrenheit)
```

Gives undesirable results.

Instead use **`return`**:

```
def cel2far(temperature):
    return temperature * (9/5) + 32

cel2far(20)
```

Without a print, nothing appears in the console.

```
def cel2far(temperature):
    return temperature * (9/5) + 32

fahrenheit = cel2far(20)
print(fahrenheit)
```

Now we can store the result to do further calculations.

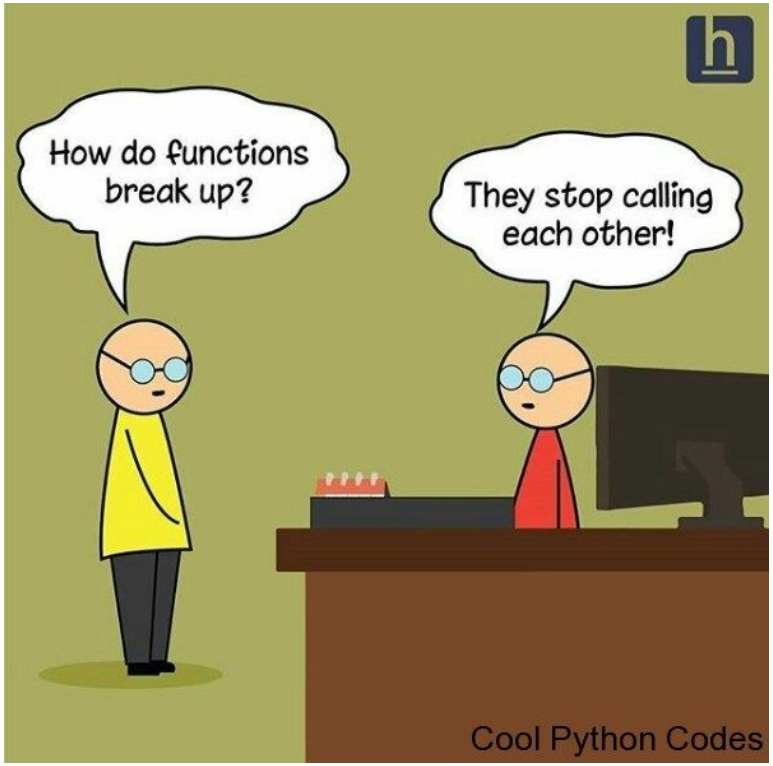
# Difference between Return and Print

| return | vs. | print |
|---|---|---|
| • Return only has meaning inside a function<br>• Only one return executed inside a function<br>• Code inside function but after return statement not executed<br>• Has a value associated with it, given to function caller | | • print can be used outside functions<br>• Can execute many print statement inside a function<br>• Code inside function can be executed after a print statement<br>• Has a value associated with it, outputted to the console |

# The Main Function

- **`main()`**:
  - Specify the flow of execution of a program
  - Define how functions are invoked in a specific order within a program

```
def function1():
    ...

def function2():
    ...

def function3():
    ...

def main():
    function2()
    function3()
    function1()

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



# Example: The Main Function

```
def addition_func(first_arg, second_arg):
    result = first_arg + second_arg
    return result

def subtraction_func(first_arg, second_arg):
    result = first_arg - second_arg
    return result

def main():
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))
    operator = input("Enter either + or -: ")

    if operator == '+':
        output = addition_func(num1, num2)
        print("The result is", output)
    elif operator == '-':
        output = subtraction_func(num1, num2)
        print("The result is", output)
    else:
        print("Invalid operator!")

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

1

2

3

http://www.pythontutor.com/