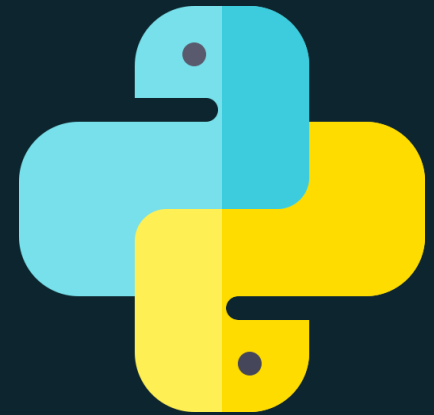


# Brief Python

## Python Course for Programmers



## Learn Python Language for Data Science

CHAPTER 2: STRUCTURED PROGRAM

DR. ERIC CHOU

IEEE SENIOR MEMBER



# Topics

---

- Conditional and Logic
- Function
- Loops



# Overview

---

LECTURE 1



# Overview

---

In this chapter, we focused on all the control structures in Python language. It includes

- Selection
- Repetition
- Functions

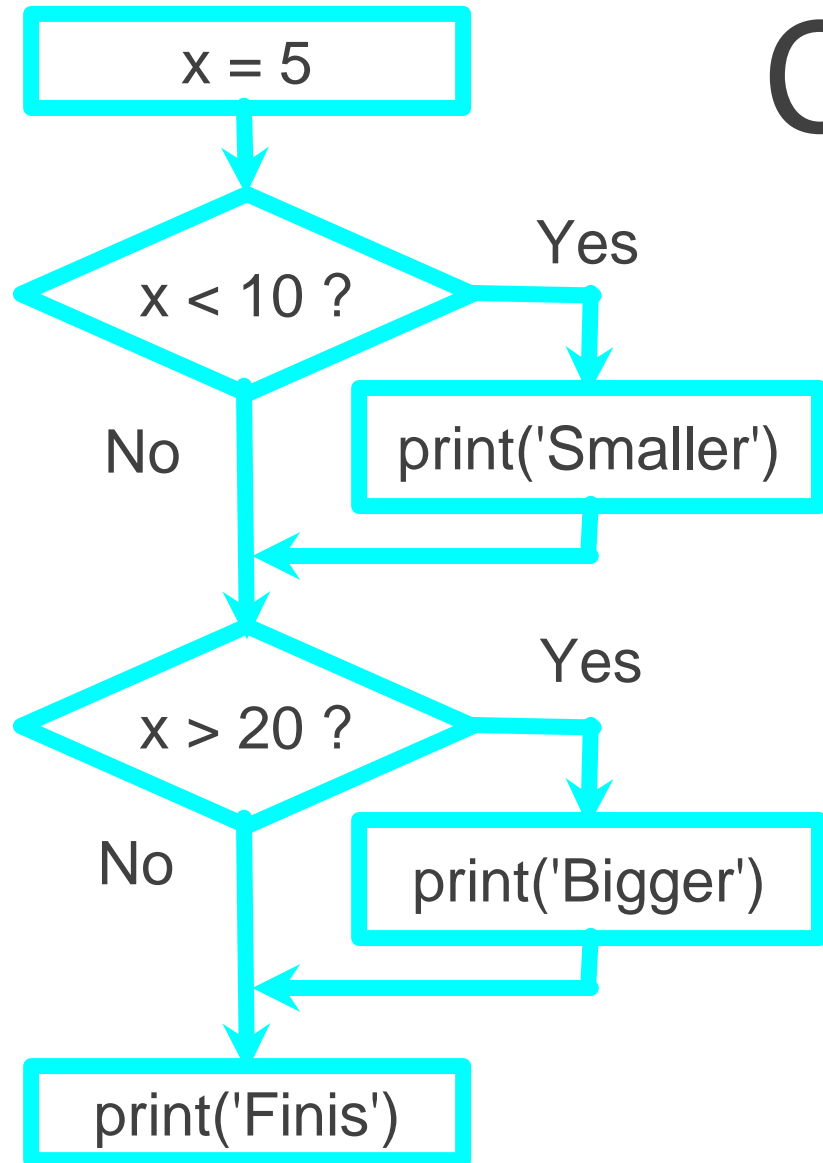


# Conditional and Logic

---

LECTURE 1

# Conditional Steps



## Program:

```
x = 5
if x < 10:
    print('Smaller')
if x > 20:
    print('Bigger')

print('Finis')
```

Output:

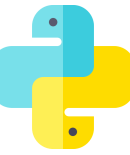
Smaller  
Finis



# Boolean

---

ACTIVITY



# Comparison Operators

- Boolean expressions ask a question and produce a Yes or No result which we use to control program flow
- Boolean expressions using comparison operators evaluate to True / False or Yes / No
- Comparison operators look at variables but do not change the variables

Python	Meaning
<	Less than
<=	Less than or Equal to
==	Equal to
>=	Greater than or Equal to
>	Greater than
!=	Not equal

Remember: “=” is used for assignment.

[http://en.wikipedia.org/wiki/George\\_Boole](http://en.wikipedia.org/wiki/George_Boole)





# Comparison Operators

---

```
x = 5
if x == 5 :
    print('Equals 5')
if x > 4 :
    print('Greater than 4')
if x >= 5 :
    print('Greater than or Equals 5')
if x < 6 : print('Less than 6')
if x <= 5 :
    print('Less than or Equals 5')
if x != 6 :
    print('Not equal 6')
```

Equals 5

Greater than 4

Greater than or Equals 5

Less than 6

Less than or Equals 5

Not equal 6



# One-way Selection

---

ACTIVITY

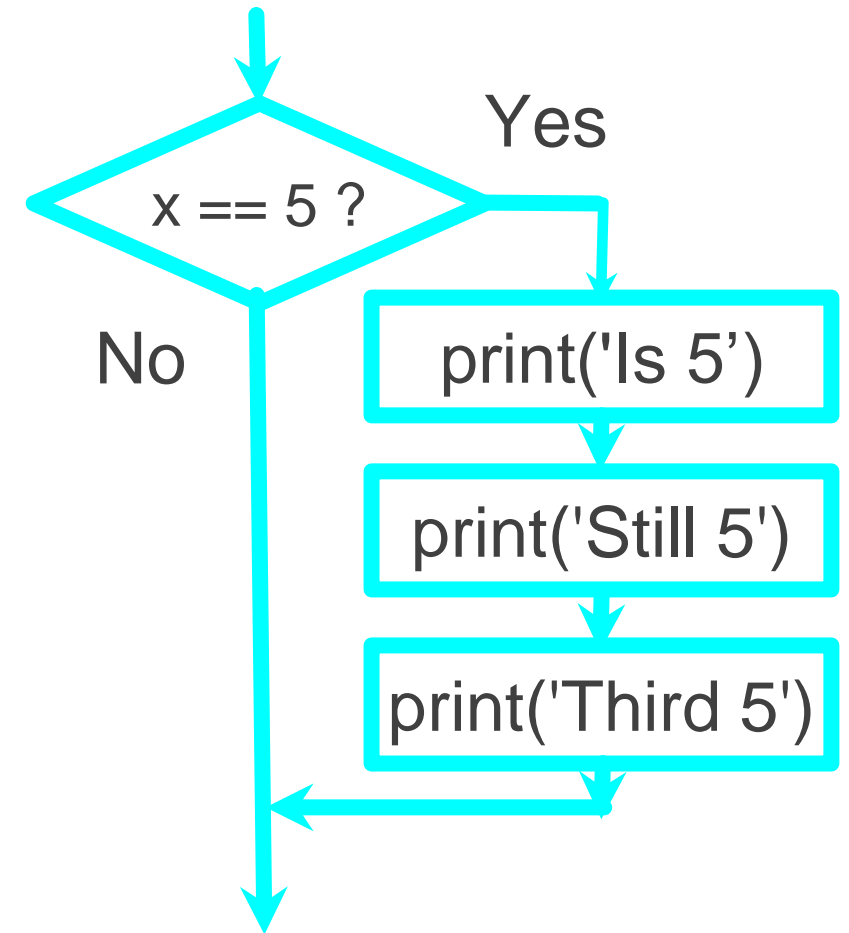
# One-Way Decisions

```
x = 5
print('Before 5')
if x == 5 :
    print('Is 5')
    print('Is Still 5')
    print('Third 5')
print('Afterwards 5')
print('Before 6')
if x == 6 :
    print('Is 6')
    print('Is Still 6')
    print('Third 6')
print('Afterwards 6')
```

Before 5

Is 5  
Is Still 5  
Third 5  
Afterwards 5  
Before 6

Afterwards 6

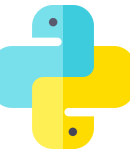




# Indentation

---

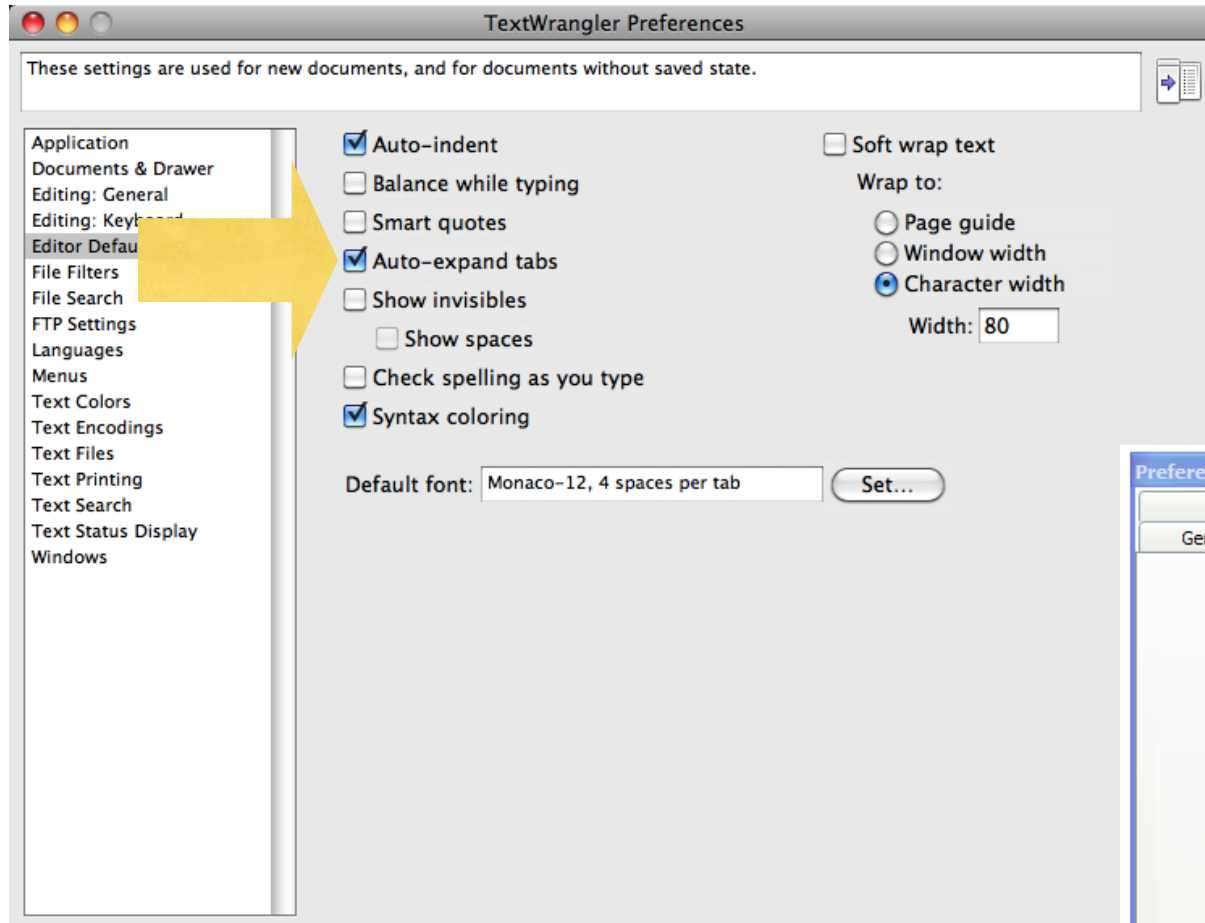
- Increase indent after an if statement or for statement (after : )
- Maintain indent to indicate the scope of the block (which lines are affected by the if/for)
- Reduce indent back to the level of the if statement or for statement to indicate the end of the block
- Blank lines are ignored - they do not affect indentation
- Comments on a line by themselves are ignored with regard to indentation



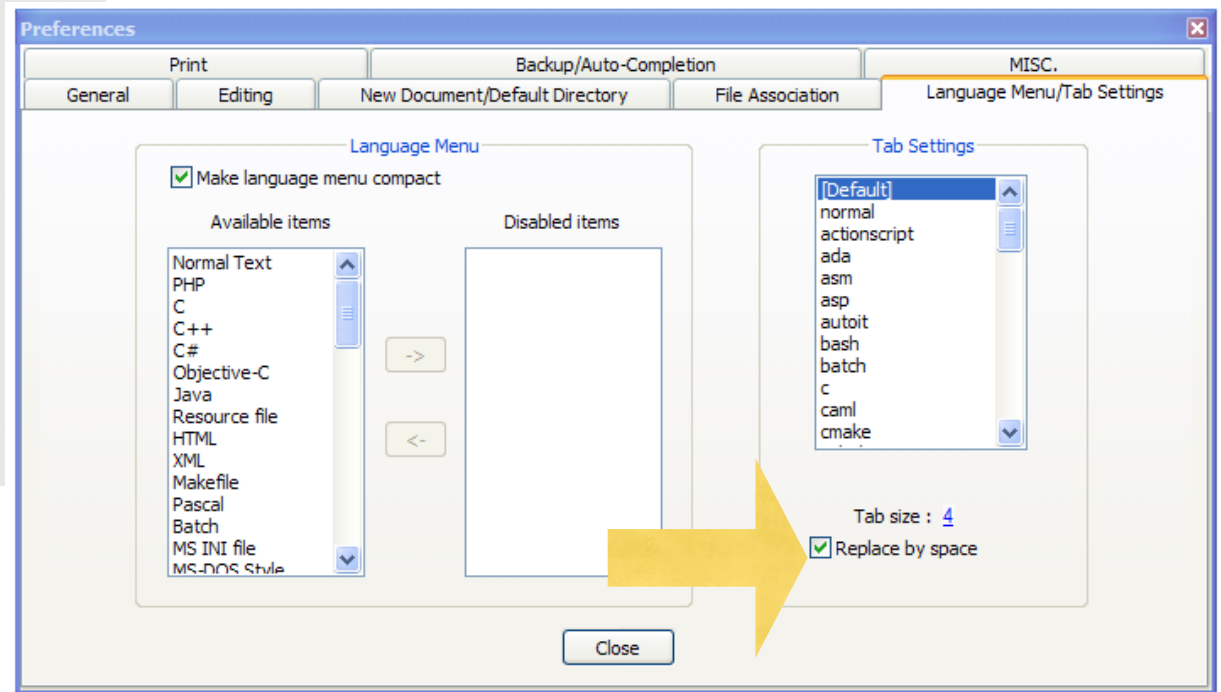
# Warning: Turn Off Tabs!!

---

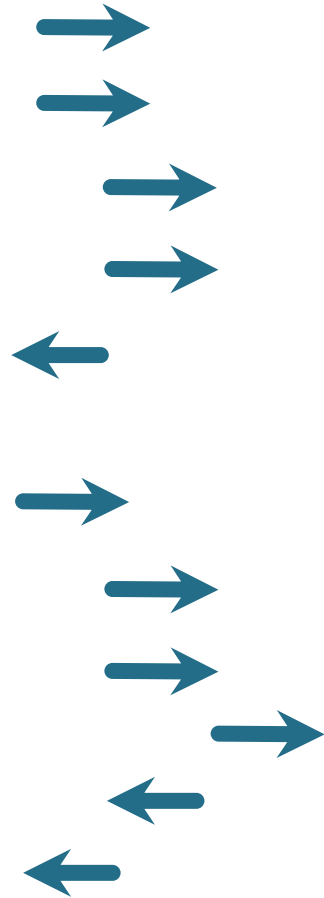
- Atom automatically uses spaces for files with ".py" extension (nice!)
- Most text editors can turn tabs into spaces - make sure to enable this feature
  - - Notepad++: Settings -> Preferences -> Language Menu/Tab Settings
  - - TextWrangler: TextWrangler -> Preferences -> Editor Defaults
- Python cares a \*lot\* about how far a line is indented. If you mix tabs and spaces, you may get "indentation errors" even if everything looks fine



This will save you  
much unnecessary  
pain.



increase / maintain after if or for  
decrease to indicate end of block



```
x = 5
if x > 2 :
    print('Bigger than 2')
    print('Still bigger')
print('Done with 2')

for i in range(5) :
    print(i)
    if i > 2 :
        print('Bigger than 2')
    print('Done with i', i)
print('All Done')
```

# Think About begin/end Blocks

```
x = 5
```

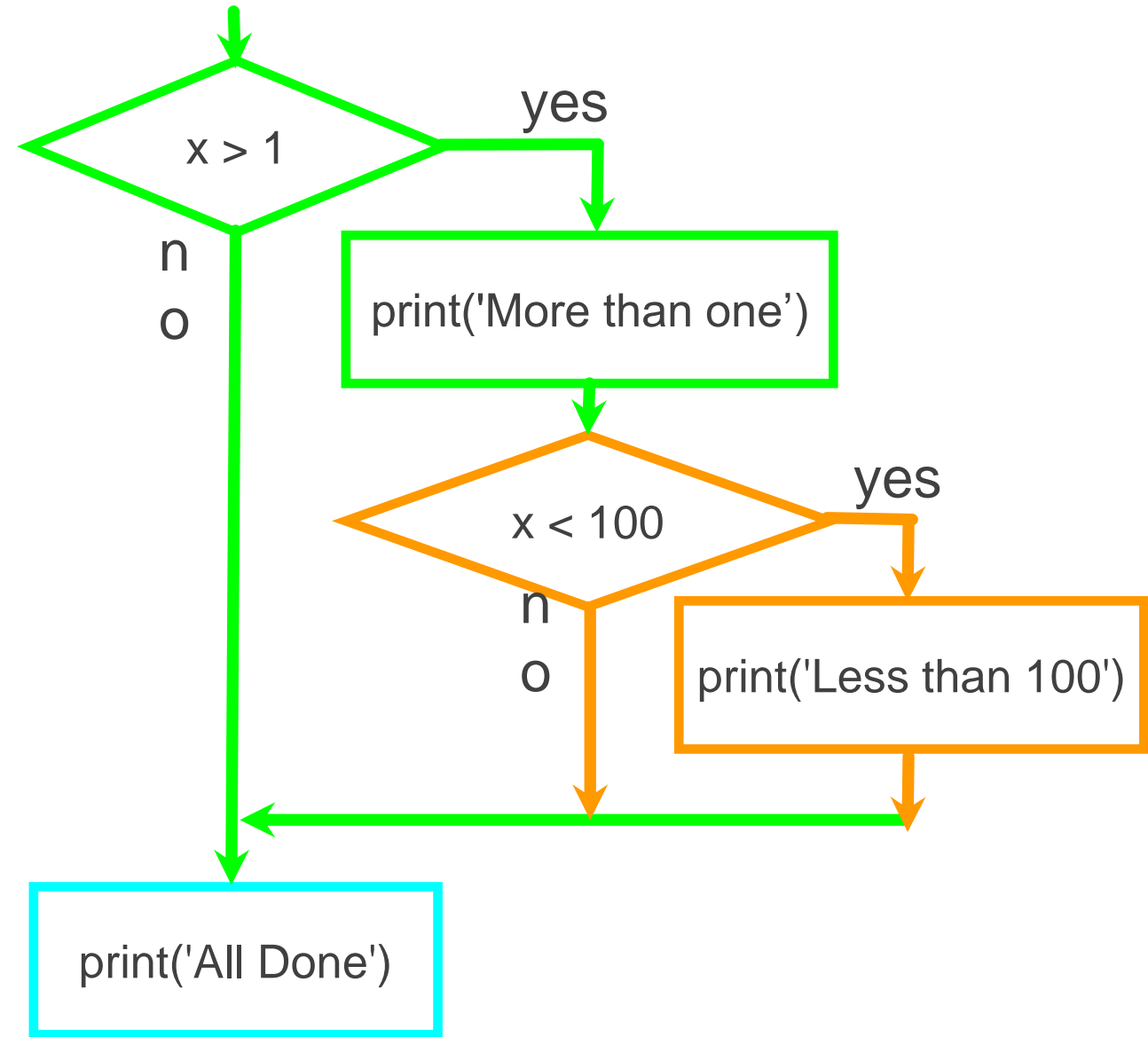
```
if x > 2 :  
    print('Bigger than 2')  
    print('Still bigger')  
print('Done with 2')
```

```
for i in range(5) :  
    print(i)  
    if i > 2 :  
        print('Bigger than 2')  
    print('Done with i', i)  
print('All Done')
```



# Nested Decisions

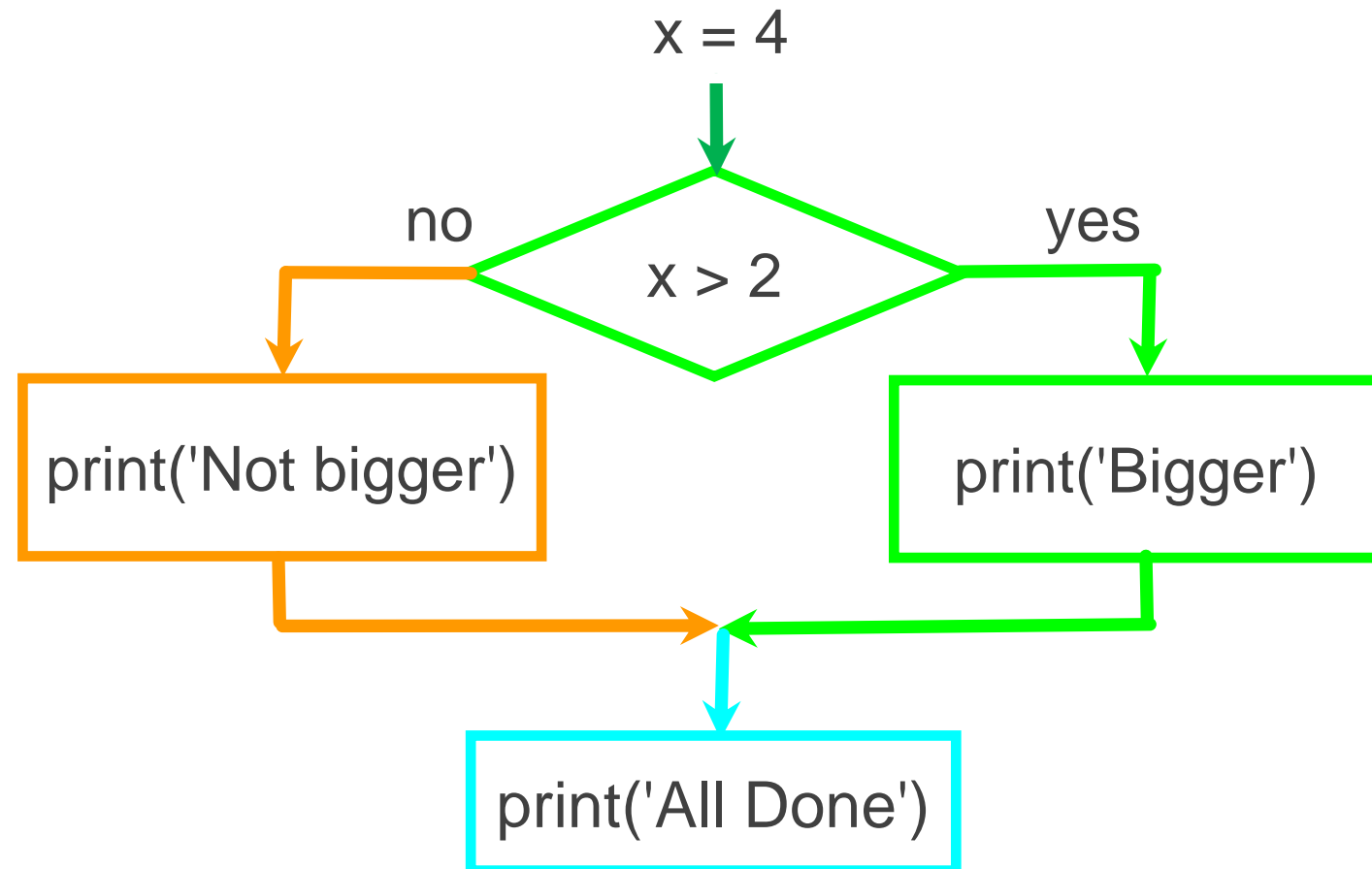
```
x = 42
if x > 1 :
    print('More than one')
    if x < 100 :
        print('Less than 100')
print('All done')
```





# Two-way Decisions

- Sometimes we want to do one thing if a logical expression is true and something else if the expression is false
- It is like a fork in the road - we must choose one or the other path but not both

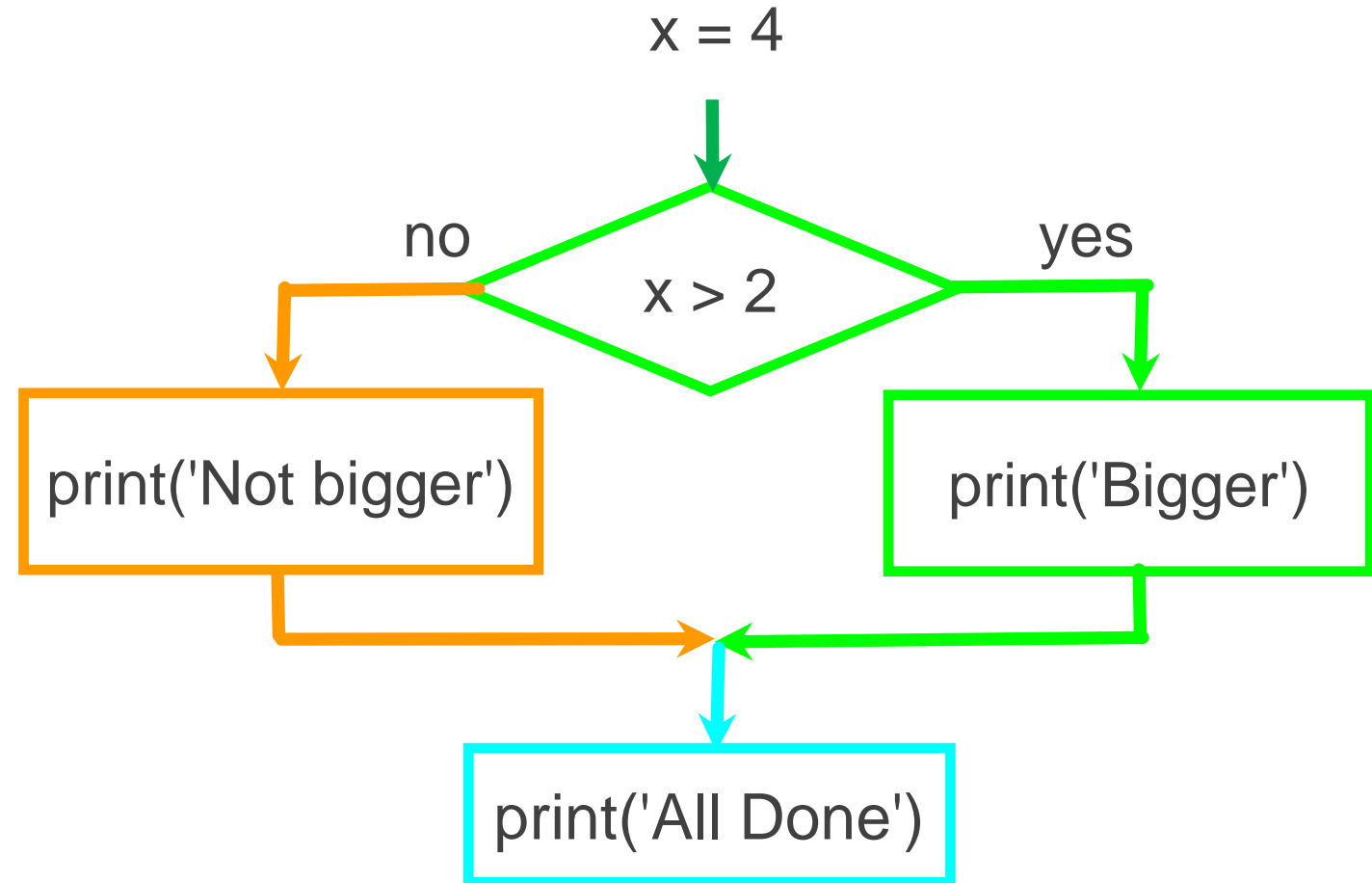




# Two-way Decisions with else:

```
x = 4
```

```
if x > 2 :  
    print('Bigger')  
else :  
    print('Smaller')  
  
print('All done')
```



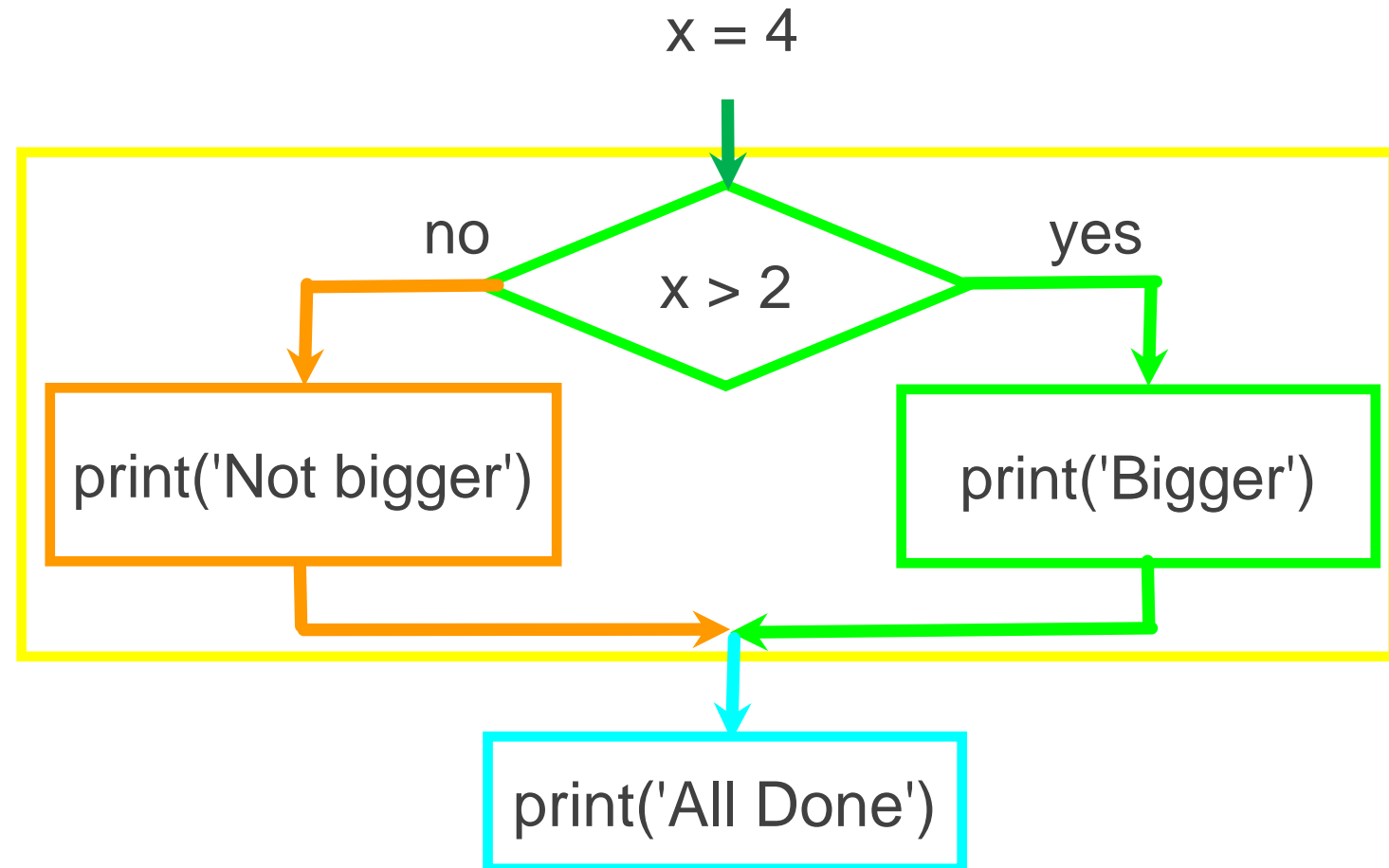


# Visualize Blocks

```
x = 4
```

```
if x > 2 :  
    print('Bigger')  
else :  
    print('Smaller')
```

```
print('All done')
```





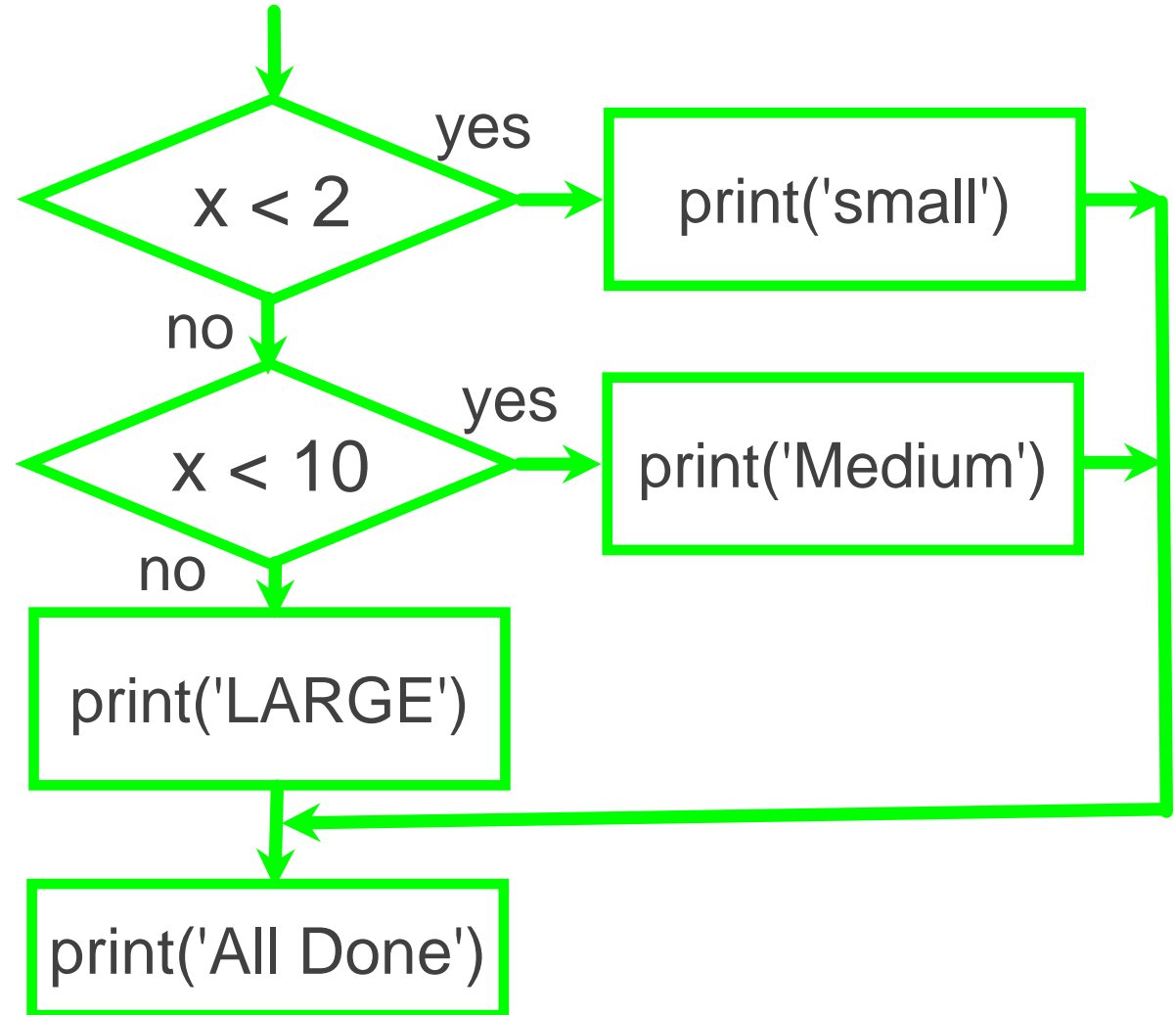
# Multi-way Selection

---

ACTIVITY

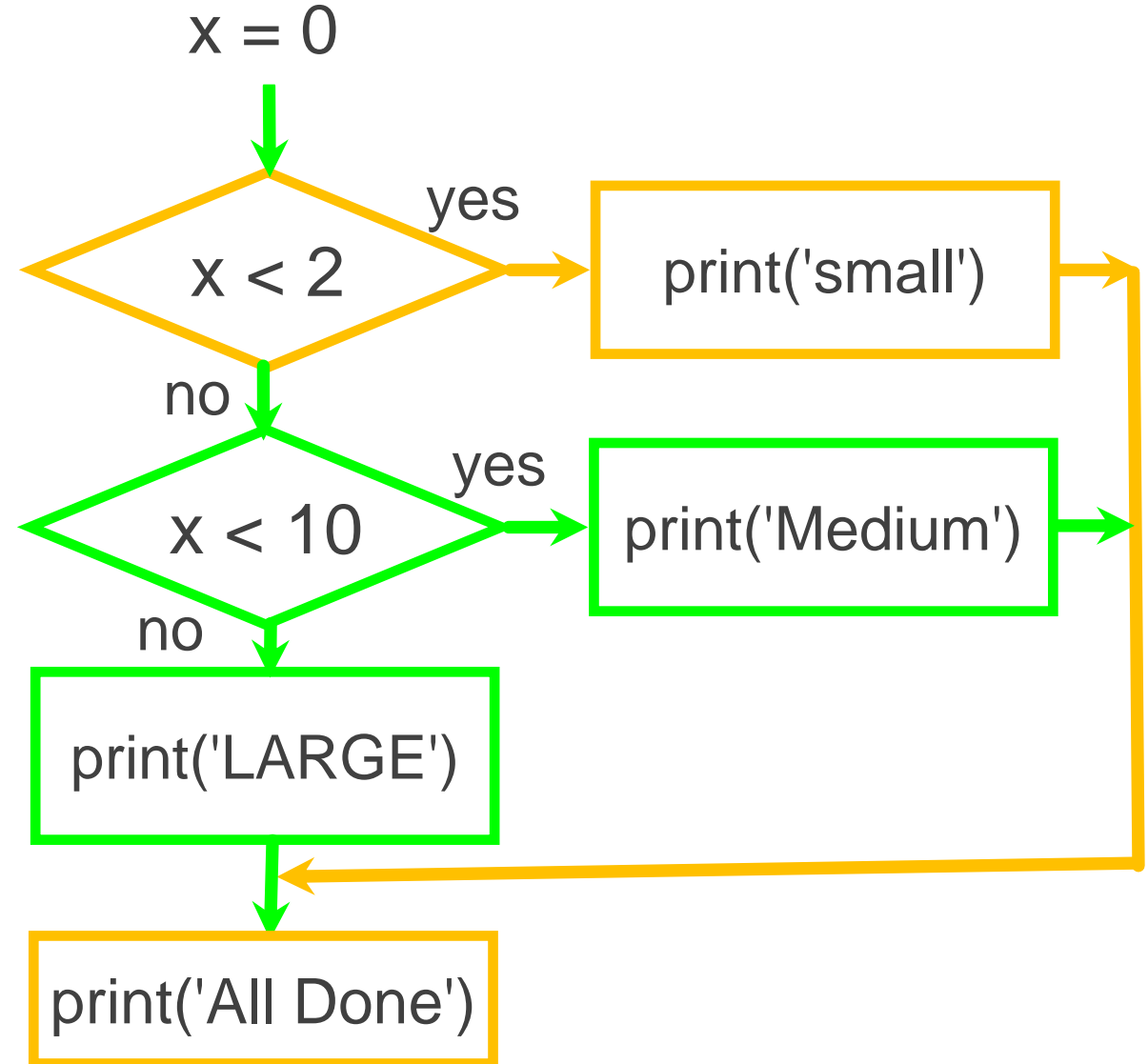
# Multi-way

```
if x < 2 :  
    print('small')  
elif x < 10 :  
    print('Medium')  
else :  
    print('LARGE')  
print('All done')
```



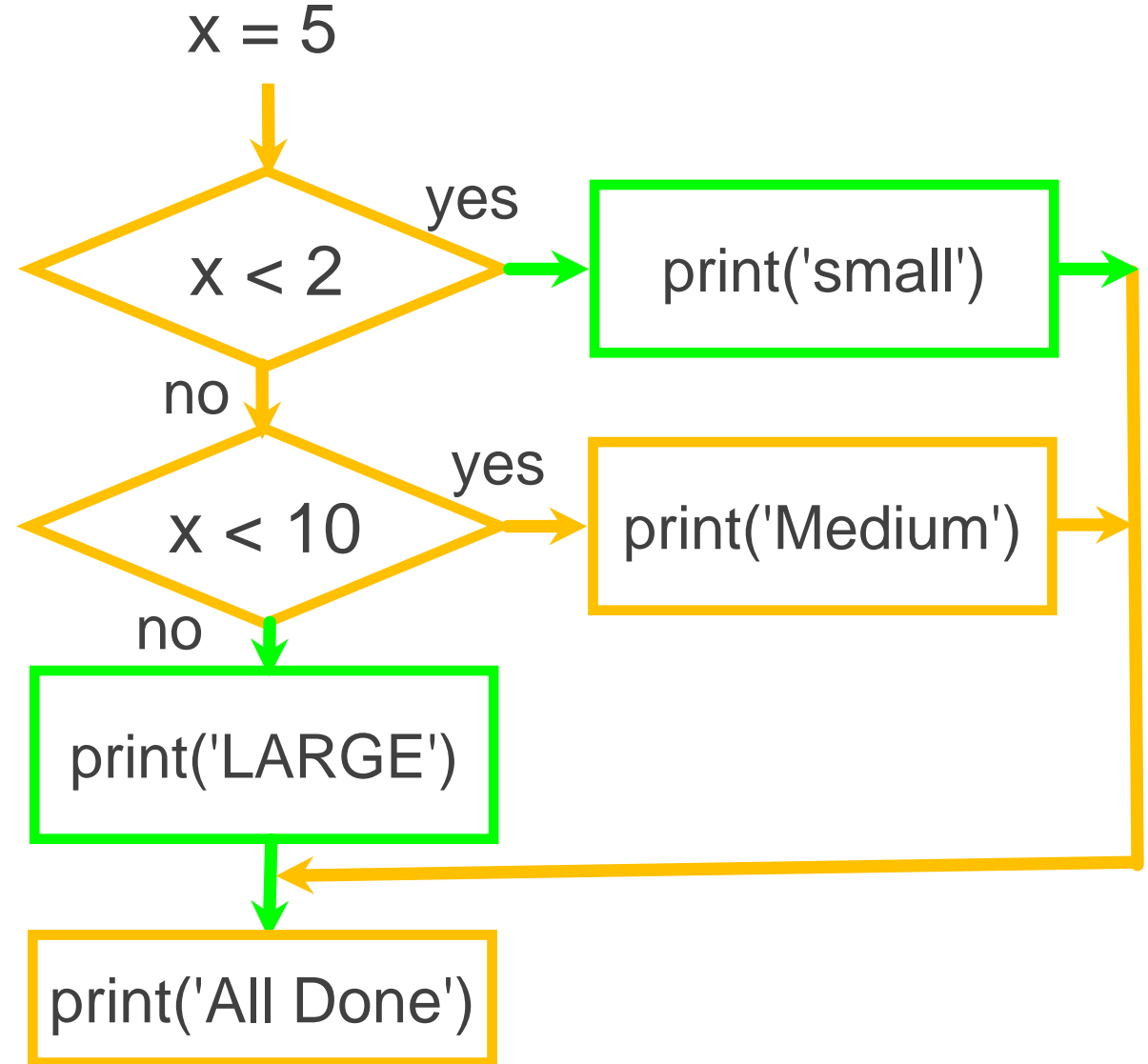
# Multi-way

```
x = 0
if x < 2 :
    print('small')
elif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')
```



# Multi-way

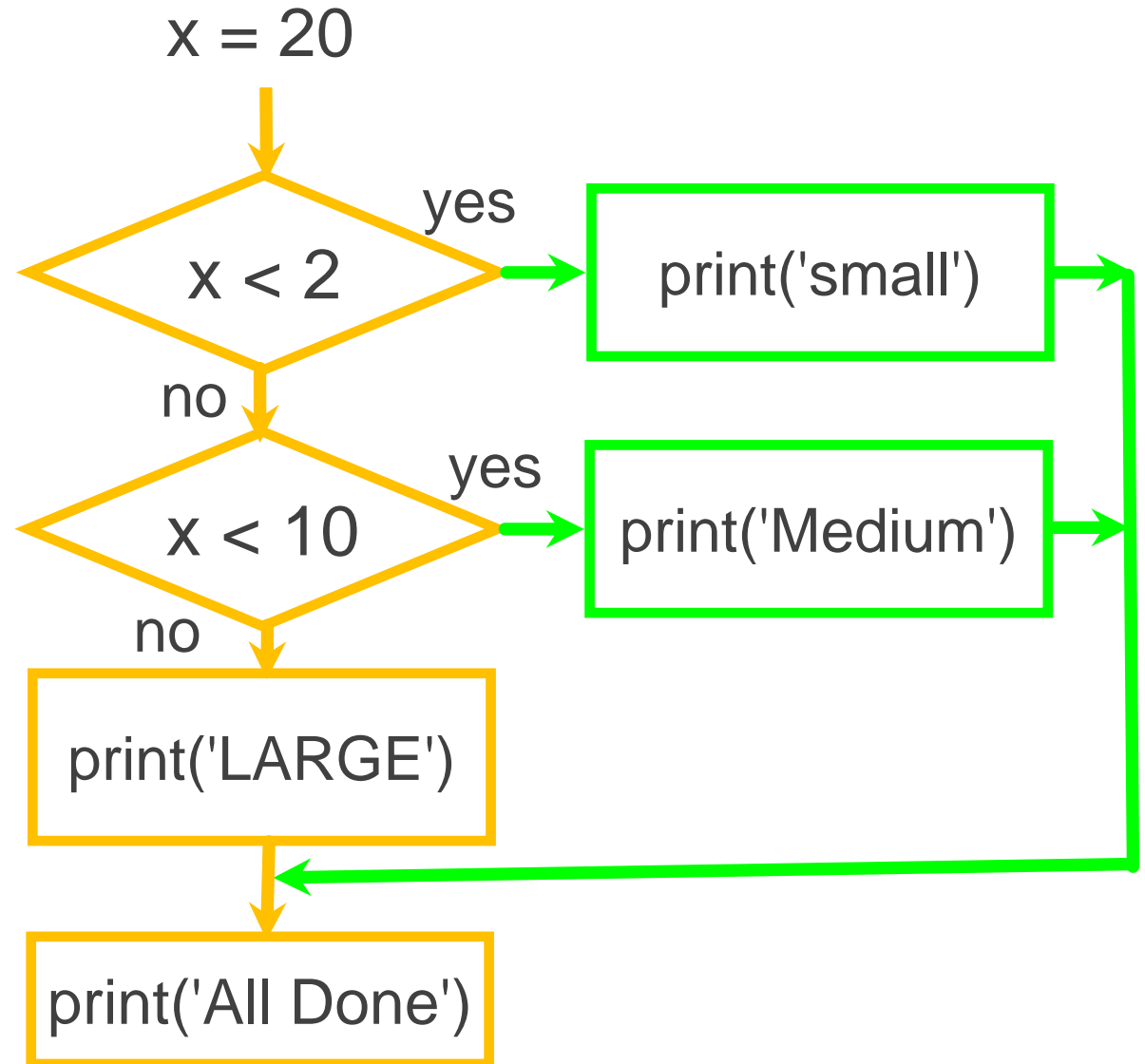
```
x = 5
if x < 2 :
    print('small')
elif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')
```





# Multi-way

```
x = 20
if x < 2 :
    print('small')
elif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')
```



# Multi-way

```
# No Else
x = 5
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')

print('All done')
```

```
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')
elif x < 20 :
    print('Big')
elif x < 40 :
    print('Large')
elif x < 100:
    print('Huge')
else :
    print('Ginormous')
```



# Multi-way Puzzles

---

Which will never print regardless of the value for x?

```
if x < 2 :  
    print('Below 2')  
elif x >= 2 :  
    print('Two or more')  
else :  
    print('Something else')
```

```
if x < 2 :  
    print('Below 2')  
elif x < 20 :  
    print('Below 20')  
elif x < 10 :  
    print('Below 10')  
else :  
    print('Something else')
```



# Match-Case

## switch statement

---

ACTIVITY



# Match Case

---

- With the introduction of **Python 3.10**, several new features were introduced, and one of them was – python match case. The Python match case is similar to the switch case statement, which is recognized as structural pattern matching in python.



# Switch Case Structure

---

- The switch control block has similar functionality as that of an if-else ladder. However, writing multiple if statements are not the most effective way of doing so. Instead, by using the switch case statement, we can simply combine them into a single structure.
- The switch case statement in a C/C++ program would look something like this:



# C++ Switch Statement

---

```
01  switch (variable to be evaluated):  
02  {  
03      case value1 : //statement 1  
04                  break;  
05  
06      case value2 : //statement 2  
07                  break;  
08  
09      case value_n : //statement n  
10                  break;  
11  
12      default:    //default statement  
13  }
```



# Match Case Statement

---

- To implement switch-case like characteristics and if-else functionalities, we use a match case in python. A match statement will compare a given variable's value to different shapes, also referred to as the pattern. The main idea is to keep on comparing the variable with all the present patterns until it fits into one.





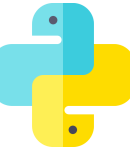
# Syntax of Match Case in Python

---

The match case consists of three main entities :

- The match keyword
- One or more case clauses
- Expression for each case

The case clause consists of a pattern to be matched to the variable, a condition to be evaluated if the pattern matches, and a set of statements to be executed if the pattern matches.



# Syntax of Match Case in Python

---

```
1 match variable_name:
2     case 'pattern1' : //statement1
3     case 'pattern2' : //statement2
4     ...
5     case 'pattern n' : //statement n
```



# Example of a Match Case Python Statement

---

```
1  quit_flag = False
2  match quit_flag:
3      case True:
4          print("Quitting")
5          exit()
6      case False:
7          print("System is on")
```



# Example of a Match Case Python Statement

---

- Here, we have a variable named `quit_flag` which we have assigned a Boolean value of `False` by default. Thus, we have a match case that will compare the patterns with the '`quit_flag`' variable.
- We have two cases for two possible values of `quit_flag` – `True` and `False`. For the first case, if the variable is `True`, then it will print 'Quittting' and execute the `exit()` function to end the program. In case if it is `false`, it will just print a statement saying that 'System is on.'



# Match Case Python for Function Overloading

---

- We can also use structural pattern matching for function overloading in python. With function overloading, we have multiple functions with the same name but with a different signature.
- Depending on the case condition, we can access the same function name with different signature values. With function overloading, we can improve the readability of the code.



# Match Case Python for Function Overloading

---

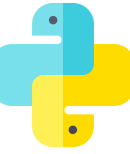
- As of now, python doesn't support function overloading. The following example can be used to verify it –
- Here, we shall be creating a user-defined function name `calc()` which accepts one argument. If the argument type is an integer, then we will return the square of the number. Else if the argument type is float, then we will return the cube of a number.



# Match Case Python for Function Overloading

---

```
01 def calc(n:int):
02     return (n**2)
03
04 def calc(n:float):
05     return (n**3)
06 n = 9.5
07 is_int = isinstance(n, int)
08
09 match is_int:
10     case True : print("Square is :",calc(n))
11     case False : print("Cube is:", calc(n))
```



# Pattern Values

---

- The match case in python allows a number, string, 'None' value, 'True' value, and 'False' value as the pattern. Thus, different types of patterns are possible, and we will be looking into some of the possible patterns.





# Wildcard Pattern

---

- We can have a wildcard pattern too. A wildcard pattern is executed when none of the other pattern values are matched.

**Following shows an example of a wildcard pattern.**

- We will take the same example of quit. But here, instead of passing a Boolean value to the variable, we will pass an integer value to the ***'quit\_flag'*** variable.



# Wildcard Pattern

---

```
01  quit_flag = 4
02
03  match quit_flag:
04      case True:
05          print("Quitting")
06          exit()
07      case False:
08          print("System is on")
09      case _:
10          print("Boolean Value was not passed")
```



# Wildcard Pattern

---

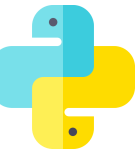
- Here the underscore has been used as a wildcard pattern. It will not bind the 'quit\_flag' variable's value to anything but match the variable's value to the statement.



# Multiple pattern values using OR operator

---

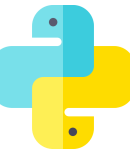
- We can also have optional values for a given case. Using the OR operator, we can execute a single expression for multiple possibilities of pattern for the given variable.
- In the below code, we have a variable name 'sample'. If the variable's value is of Boolean type, then it will execute a common statement for both the conditions. Else, for any other value, it will print 'Not a Boolean value'.



# Multiple pattern values using OR operator

---

```
1 sample = True
2
3 match sample:
4     case (True|False):
5         print("It is a boolean value")
6     case _:
7         print("Not a boolean value")
```

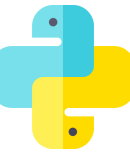


# Checking for a collection of values

---

The pattern can also be a collection of values. It will match the pattern against the entire collection. Let us take a list 'list1' as an example. We will take the entire list collection as the pattern.

```
1 list1 = ['a', 'b', 'c', 'd']
2
3 match list1:
4     case ['e', 'f'] : print("e,f present")
5     case ['a', 'b', 'c', 'd'] : print("a,b,c,d present")
```

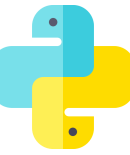


# Named constants as patterns

---

- We can have a named constant as a pattern for match case statements. The constant should be a qualified name addressed by a dot operator. It works like a literal, but it never minds.

```
01  class switch:
02      on = 1
03      off = 0
04
05  status = 0
06
07  match status:
08      case switch.on :
09          print('Switch is on')
10      case switch.off :
11          print('Switch is off')
```



# Inline if statement in match case format

---

- We can also add an if statement to a pattern for a match case in python. That if the condition is also known as 'guard'. The expression for a given pattern would be evaluated only if the guard is True. If the guard is False, it does not execute that pattern's statement.
- Let us take an example for the same. We have a variable 'n' will is assigned a numerical value. We have three cases here – n is negative, n is zero, and n is positive. The match case will check the guard and accordingly print the statement.





# Inline if statement in match case format

---

```
1  n = 0
2  match n:
3      case n if n < 0:
4          print("Number is negative")
5      case n if n == 0:
6          print("Number is zero")
7      case n if n > 0:
8          print("Number is positive")
```



# Exceptions

---

ACTIVITY



# The try / except Structure

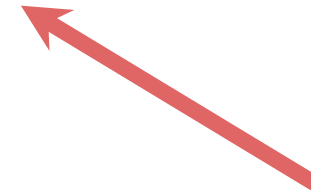
---

- You surround a dangerous section of code with try and except
- If the code in the try works - the except is skipped
- If the code in the try fails - it jumps to the except section

```
$ cat notry.py
astr = 'Hello Bob'
istr = int(astr)
print('First', istr)
astr = '123'
istr = int(astr)
print('Second', istr)
```

```
$ python3 notry.py
```

```
Traceback (most recent call last):
File "notry.py", line 2, in <module>
istr = int(astr)ValueError: invalid literal
for int() with base 10: 'Hello Bob'
```



All  
Done

The  
program  
stops  
here

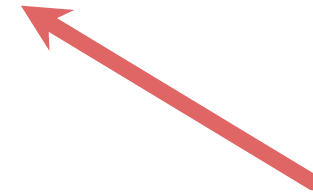


```
$ cat notry.py  
astr = 'Hello Bob'  
istr = int(astr)
```

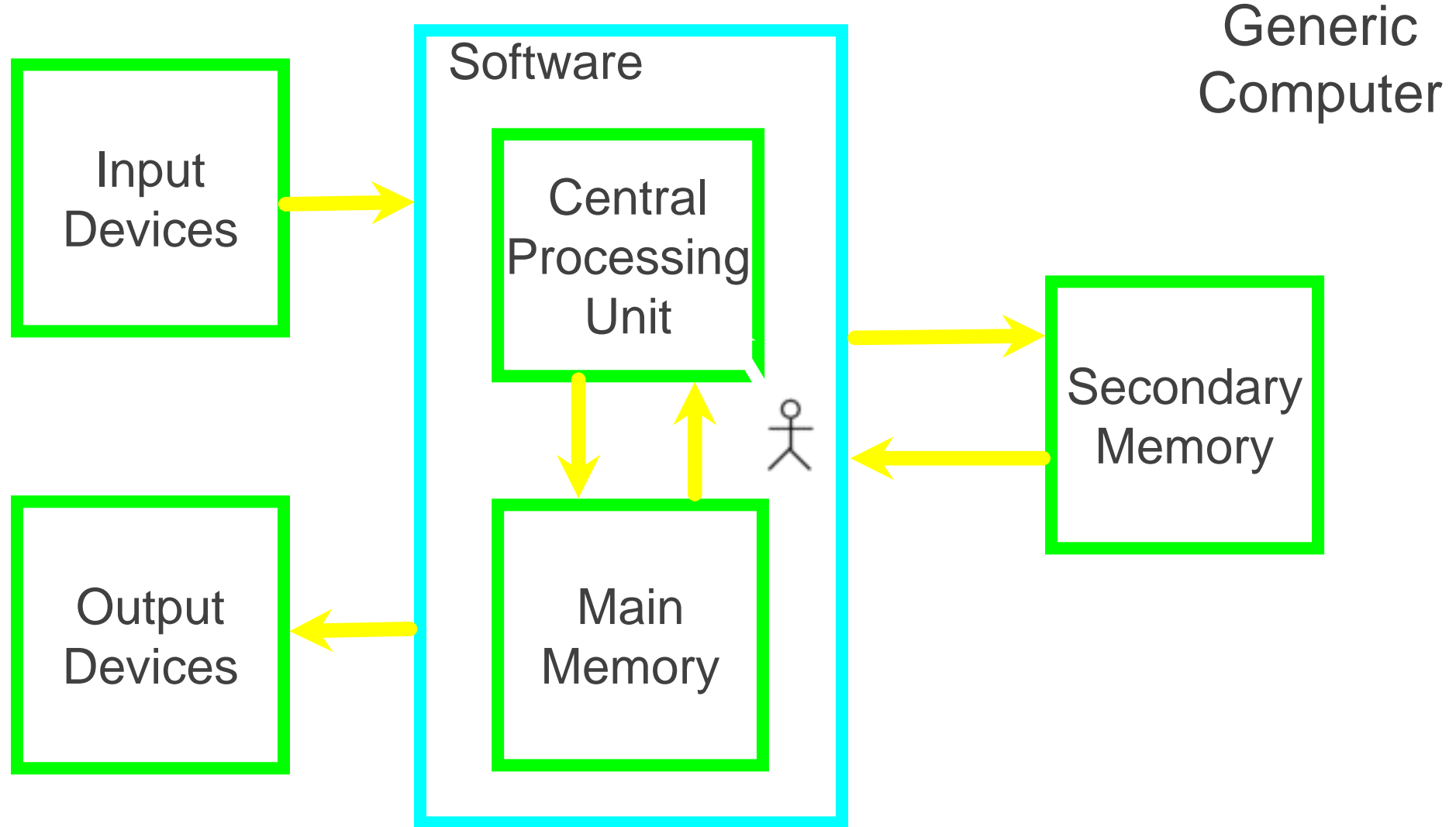


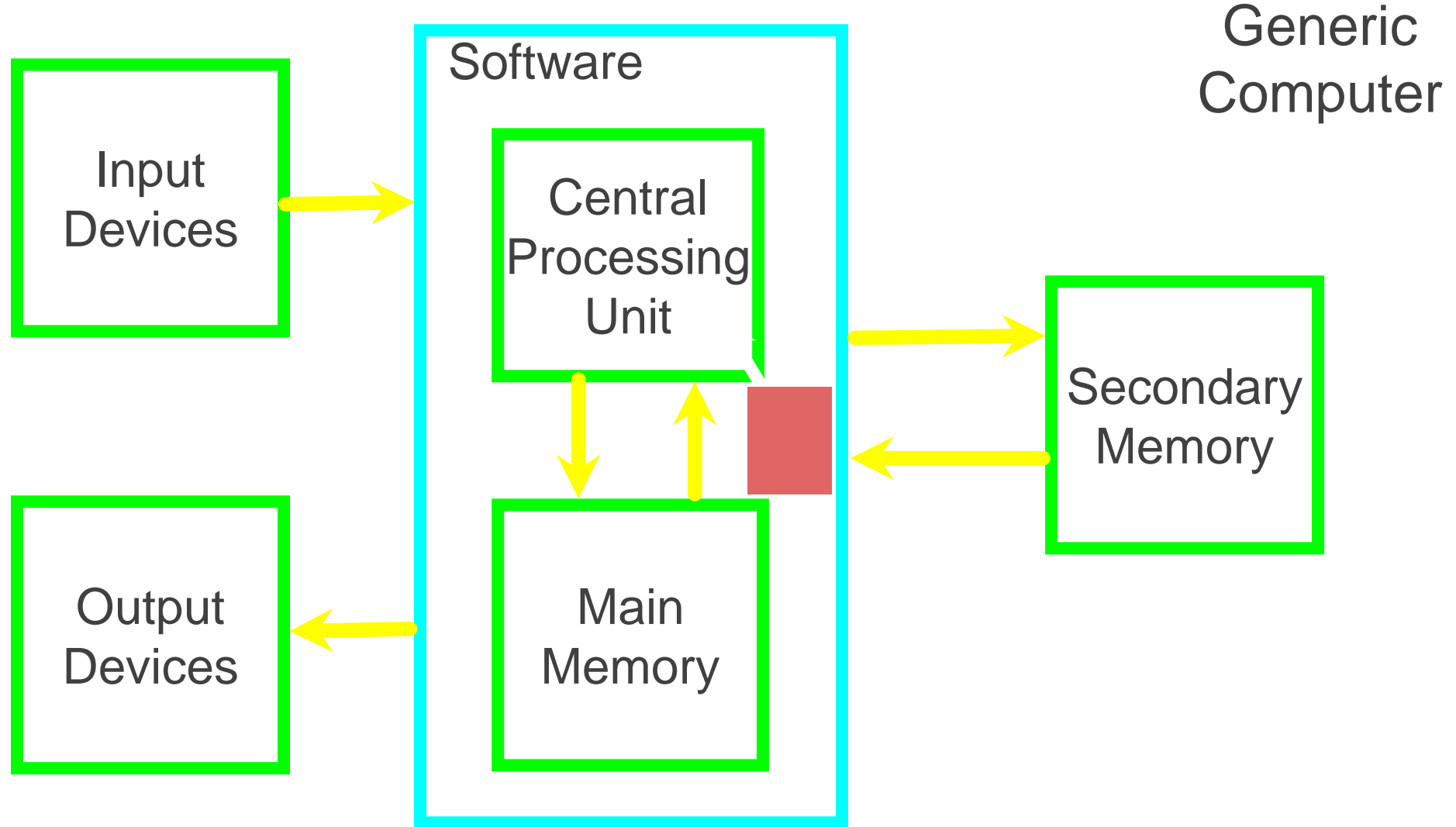
```
$ python3 notry.py
```

```
Traceback (most recent call last):  
File "notry.py", line 2, in <module>  
istr = int(astr)ValueError: invalid literal  
for int() with base 10: 'Hello Bob'
```



All  
Done





```
astr = 'Hello Bob'
try:
    istr = int(astr)
except:
    istr = -1

print('First', istr)
```

```
astr = '123'
try:
    istr = int(astr)
except:
    istr = -1

print('Second', istr)
```

When the first conversion fails - it just drops into the except: clause and the program continues.

```
$ python tryexcept.py
First -1
Second 123
```

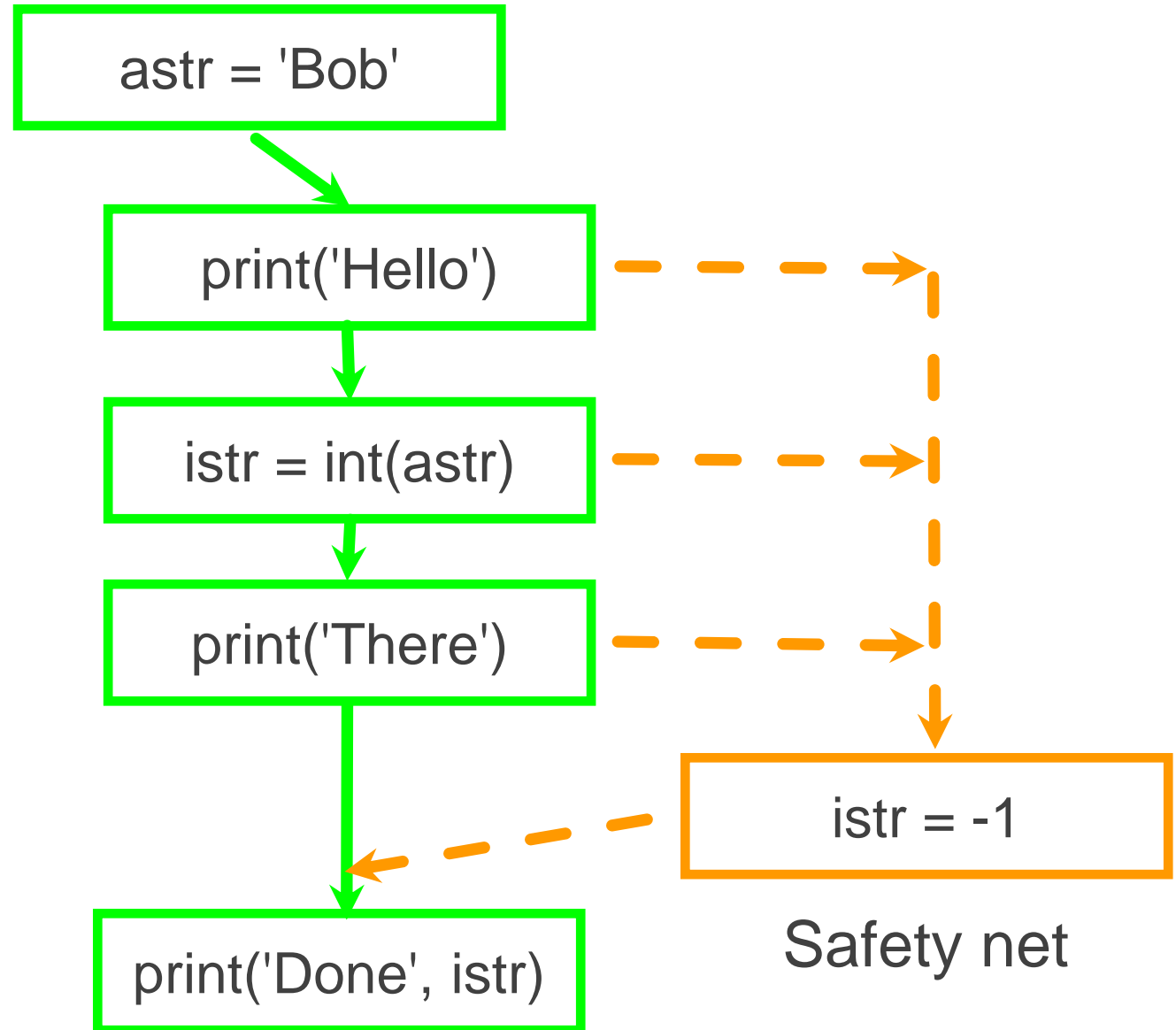
When the second conversion succeeds - it just skips the except: clause and the program continues.



# try / except

```
astr = 'Bob'
try:
    print('Hello')
    istr = int(astr)
    print('There')
except:
    istr = -1

print('Done', istr)
```





# Sample try / except

---

```
rawstr = input('Enter a number:')
try:
    ival = int(rawstr)
except:
    ival = -1

if ival > 0 :
    print('Nice work')
else:
    print('Not a number')
```

```
$ python3 trynum.py
Enter a number:42
Nice work
$ python3 trynum.py
Enter a number:forty-two
Not a number
$
```



# Summary

---

- Comparison operators  
== <= >= > < !=
- Indentation
- One-way Decisions
- Two-way decisions:  
if: and else:
- Nested Decisions
- Multi-way decisions using elif
- try / except to compensate for errors



# Lab

---

SALARY

## Exercise

Rewrite your pay computation to give the employee 1.5 times the hourly rate for hours worked above 40 hours.

Enter Hours: 45

Enter Rate: 10

Pay: 475.0

$$475 = 40 * 10 + 5 * 15$$

## Exercise

Rewrite your pay program using try and except so that your program handles non-numeric input gracefully.

```
Enter Hours: 20
```

```
Enter Rate: nine
```

```
Error, please enter numeric input
```

```
Enter Hours: forty
```

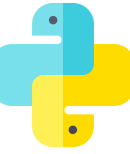
```
Error, please enter numeric input
```



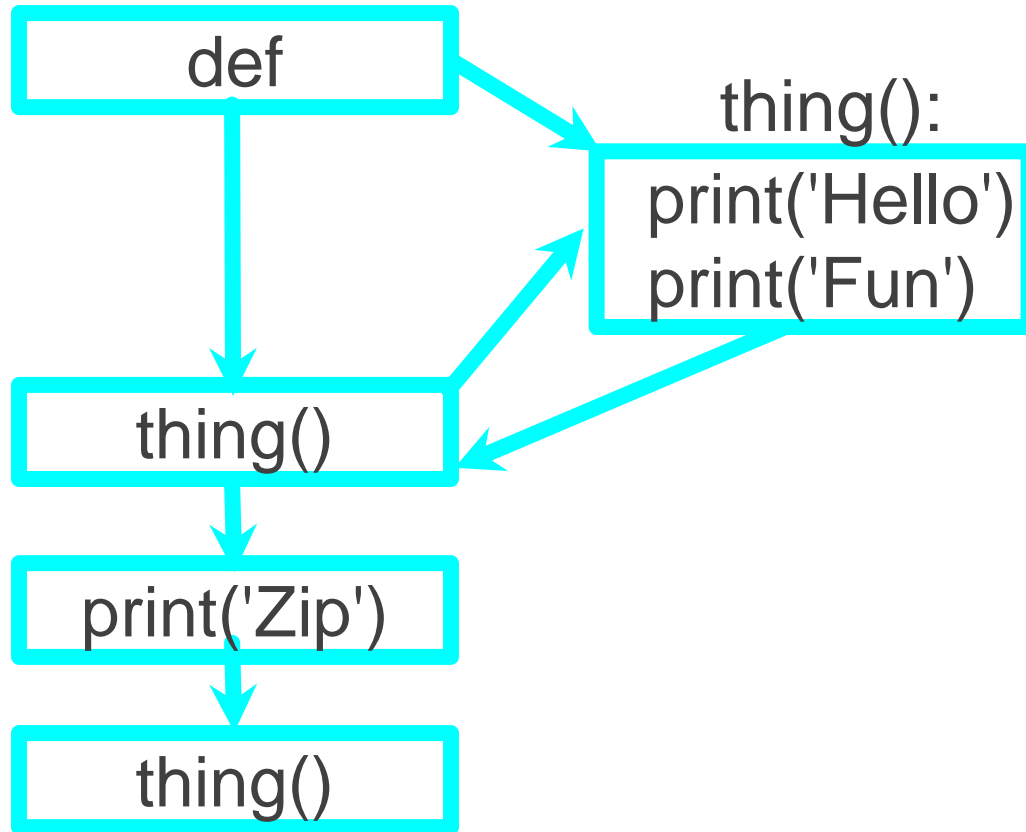
# Function

---

LECTURE 1



# Stored (and reused) Steps



Program:

```
def thing():  
    print('Hello')  
    print('Fun')
```

```
thing()  
print('Zip')  
thing()
```

Output:

```
Hello  
Fun  
Zip  
Hello  
Fun
```

We call these reusable pieces of code “functions”





# Python Functions

---

- There are two kinds of functions in Python.
  - - Built-in functions that are provided as part of Python - `print()`, `input()`, `type()`, `float()`, `int()` ...
  - - Functions that we define ourselves and then use
- We treat the built-in function names as “new” reserved words  
(i.e., we avoid them as variable names)



# Function Definition

---

- In Python a function is some reusable code that takes arguments(s) as input, does some computation, and then returns a result or results
- We define a function using the def reserved word
- We call/invoke the function by using the function name, parentheses, and arguments in an expression



# Built-in Functions

---

ACTIVITY

Argument

big = max('Hello world')

Assignment

'w'

Result

```
>>> big = max('Hello world')
>>> print(big)
w
>>> tiny = min('Hello world')
>>> print(tiny)

>>>
```

# Max Function

```
>>> big = max('Hello world')  
>>> print(big)  
w
```

'Hello world'  
(a string)

max()  
function

'w'  
(a string)

A function is some stored code that we use. A function takes some input and produces an output.

Guido wrote this code

# Max Function

```
>>> big = max('Hello world')  
>>> print(big)  
w
```

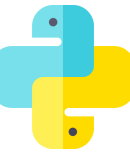
'Hello world'  
(a string)

```
def max(inp):  
    blah  
    blah  
    for x in inp:  
        blah  
        blah
```

A function is some stored code that we use. A function takes some input and produces an output.

'w'  
(a string)

Guido wrote this code



# Type Conversions

---

- When you put an integer and floating point in an expression, the integer is implicitly converted to a float
- You can control this with the built-in functions `int()` and `float()`

```
>>> print(float(99) / 100)
0.99
>>> i = 42
>>> type(i)
<class 'int'>
>>> f = float(i)
>>> print(f)
42.0
>>> type(f)
<class 'float'>
>>> print(1 + 2 * float(3) / 4 - 5)
-2.5
>>>
```

# String Conversions

- You can also use `int()` and `float()` to convert between strings and integers
- You will get an error if the string does not contain numeric characters

```
>>> sval = '123'
>>> type(sval)
<class 'str'>
>>> print(sval + 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str'
and 'int'
>>> ival = int(sval)
>>> type(ival)
<class 'int'>
>>> print(ival + 1)
124
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
```





# Custom Functions

---

ACTIVITY



# Building our Own Functions

---

- We create a new function using the `def` keyword followed by optional parameters in parentheses
- We indent the body of the function
- This defines the function but does not execute the body of the function

```
def print_lyrics():  
    print("I'm a lumberjack, and I'm okay.")  
    print('I sleep all night and I work all  
day.')
```

print\_lyrics():

```
print("I'm a lumberjack, and I'm okay.")  
print('I sleep all night and I work all day.')
```

```
x = 5  
print('Hello')
```

```
def print_lyrics():  
    print("I'm a lumberjack, and I'm okay.")  
    print('I sleep all night and I work all day.')
```

```
print('Yo')  
x = x + 2  
print(x)
```

Hello  
Yo  
7



# Definitions and Uses

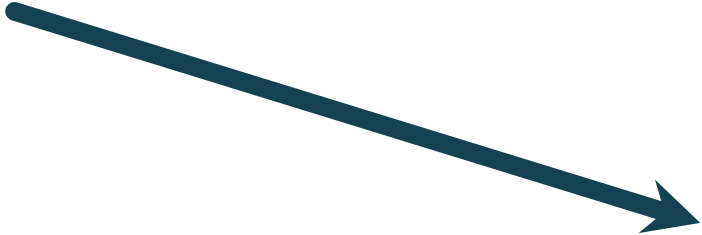
---

- Once we have defined a function, we can call (or invoke) it as many times as we like
- This is the store and reuse pattern

```
x = 5  
print('Hello')
```

```
def print_lyrics():  
    print("I'm a lumberjack, and I'm okay.")  
    print('I sleep all night and I work all day.')
```

```
print('Yo')  
print_lyrics()  
x = x + 2  
print(x)
```



Hello  
Yo  
I'm a lumberjack, and I'm okay.  
I sleep all night and I work all day.  
7



# Parameters

---

ACTIVITY



# Arguments

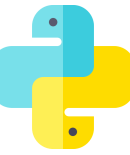
---

- An argument is a value we pass into the function as its input when we call the function
- We use arguments so we can direct the function to do different kinds of work when we call it at different times
- We put the arguments in parentheses after the name of the function

```
big = max('Hello world')
```



Argument



# Parameters

---

- A parameter is a variable which we use in the function definition. It is a “handle” that allows the code in the function to access the arguments for a particular function invocation.

```
>>> def greet(lang):  
...     if lang == 'es':  
...         print('Hola')  
...     elif lang == 'fr':  
...         print('Bonjour')  
...     else:  
...         print('Hello')  
...  
>>> greet('en')  
Hello  
>>> greet('es')  
Hola  
>>> greet('fr')  
Bonjour  
>>>
```





# Return Values

---

ACTIVITY



# Return Values

---

- Often a function will take its arguments, do some computation, and return a value to be used as the value of the function call in the calling expression. The return keyword is used for this.

```
def greet():  
    return "Hello"  
  
print(greet(), "Glenn")  
print(greet(), "Sally")
```

Hello Glenn  
Hello Sally



# Return Value

---

- A “fruitful” function is one that produces a result (or return value)
- The return statement ends the function execution and “sends back” the result of the function

```
>>> def greet(lang):  
...     if lang == 'es':  
...         return 'Hola'  
...     elif lang == 'fr':  
...         return 'Bonjour'  
...     else:  
...         return 'Hello'  
...  
>>> print(greet('en'), 'Glenn')  
Hello Glenn  
>>> print(greet('es'), 'Sally')  
Hola Sally  
>>> print(greet('fr'), 'Michael')  
Bonjour Michael  
>>>
```



# Putting Things Together

---

ACTIVITY

# Arguments, Parameters, and Results

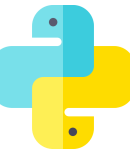
```
>>> big = max('Hello world')  
>>> print(big)  
w
```

Argument → 'Hello world'

```
def max(inp):  
    blah  
    blah  
    for x in inp:  
        blah  
        blah  
    return 'w'
```

Parameter

'w'  
↑  
Result



# Multiple Parameters / Arguments

---

- We can define more than one parameter in the function definition
- We simply add more arguments when we call the function
- We match the number and order of arguments and parameters

```
def addtwo(a, b):  
    added = a + b  
    return added
```

```
x = addtwo(3, 5)  
print(x)
```

8



# Void (non-fruitful) Functions

---

- When a function does not return a value, we call it a “void” function
- Functions that return values are “fruitful” functions

Void functions are “not fruitful”



# To function or not to function...

---

- Organize your code into “paragraphs” - capture a complete thought and “name it”
- Don’t repeat yourself - make it work once and then reuse it
- If something gets too long or complex, break it up into logical chunks and put those chunks in functions
- Make a library of common stuff that you do over and over - perhaps share this with your friends...





# Loops

---

LECTURE 1

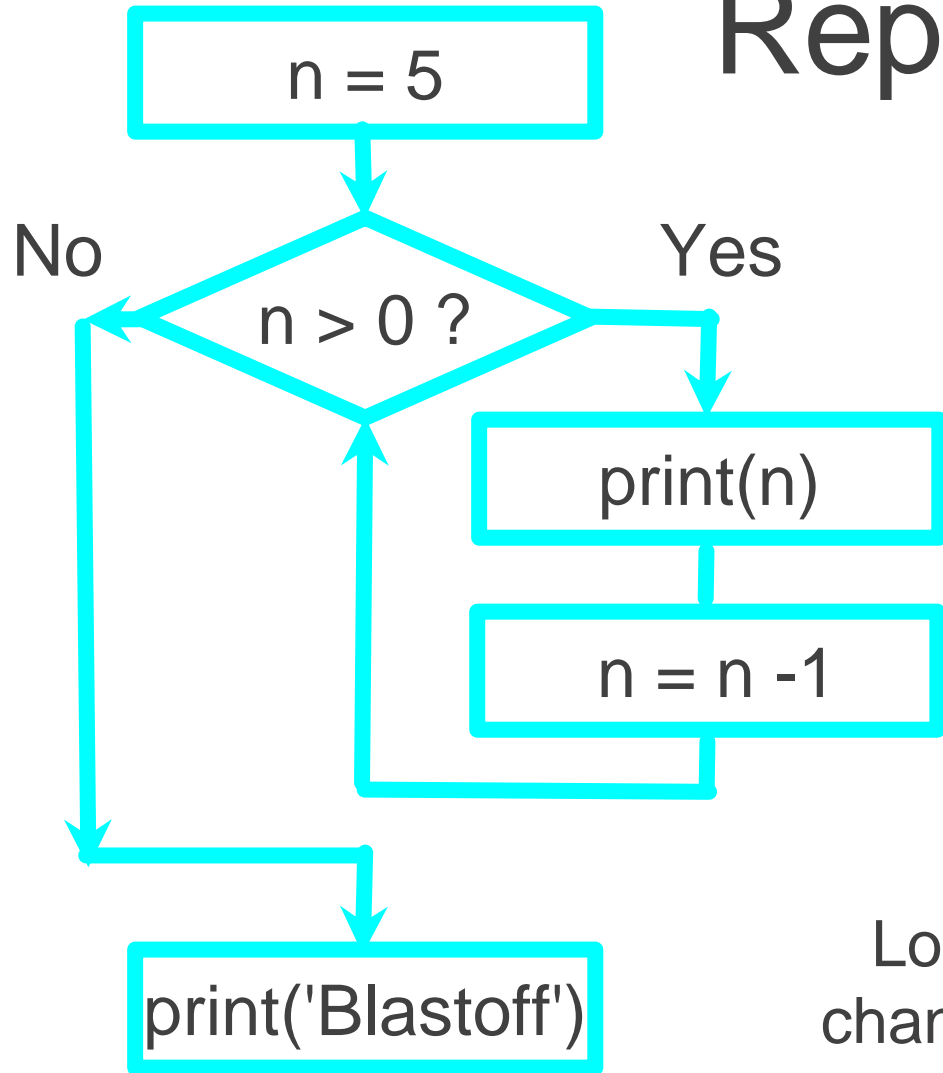


# Repetition

---

ACTIVITY

# Repeated Steps



Program:

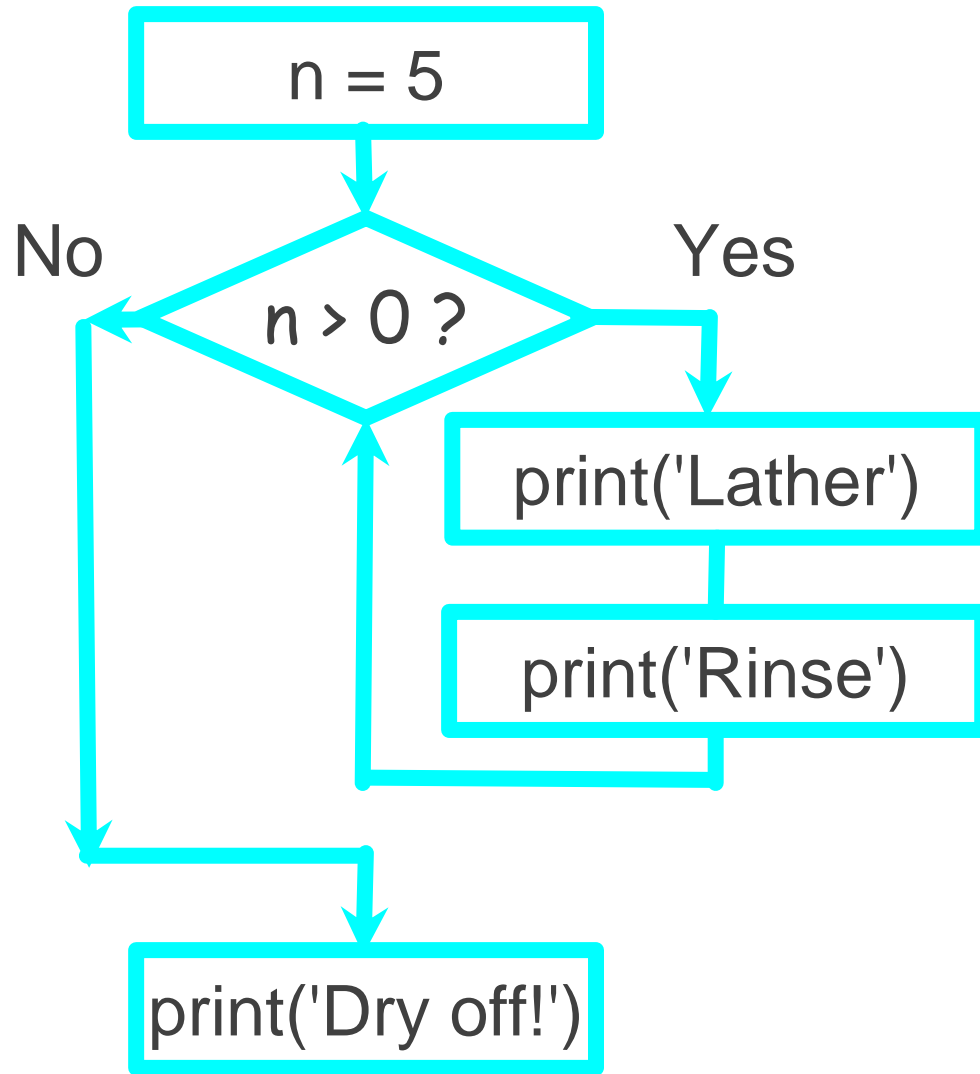
```
n = 5
while n > 0 :
    print(n)
    n = n - 1
print('Blastoff!')
print(n)
```

Output:

5  
4  
3  
2  
1  
Blastoff!  
0

Loops (repeated steps) have iteration variables that change each time through a loop. Often these iteration variables go through a sequence of numbers.

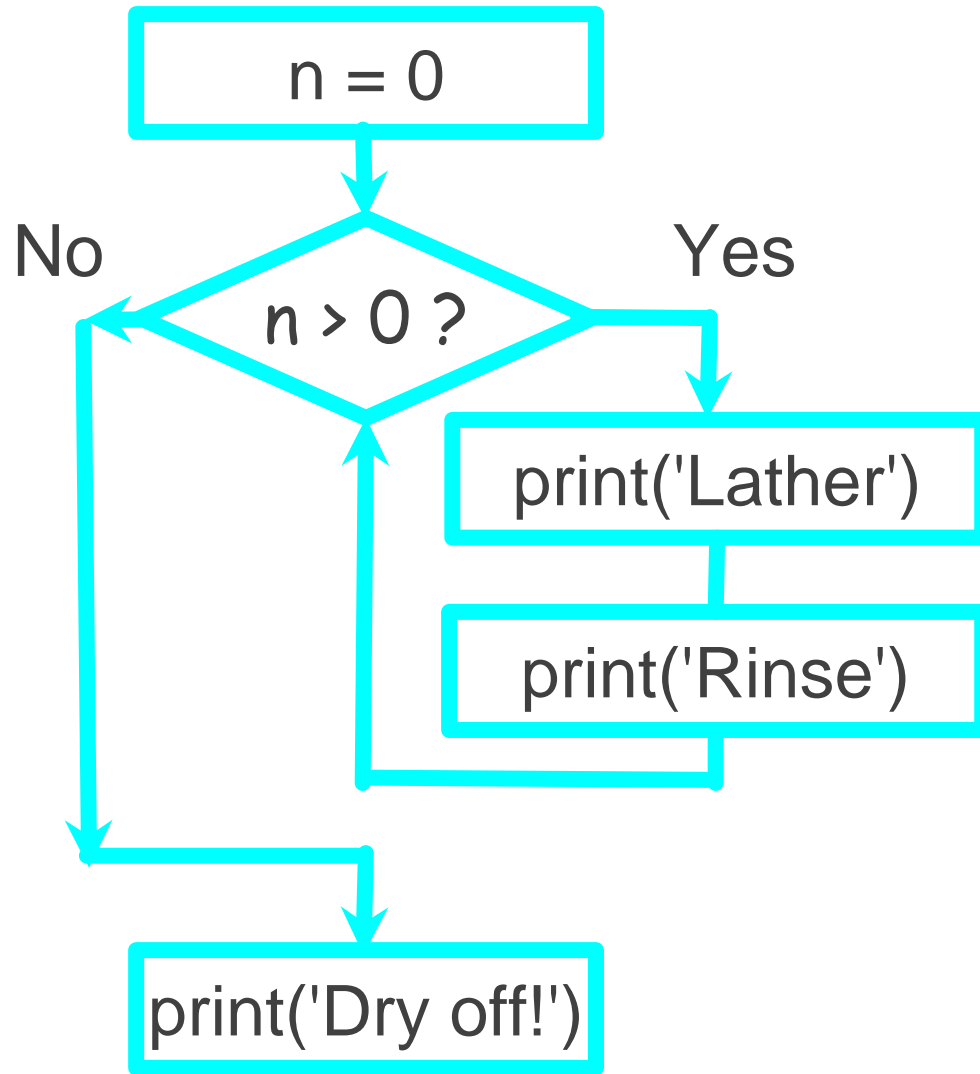
# An Infinite Loop



```
n = 5
while n > 0 :
    print('Lather')
    print('Rinse')
print('Dry off!')
```

What is wrong with this loop?

# Another Loop



```
n = 0
while n > 0 :
    print('Lather')
    print('Rinse')
print('Dry off!')
```

What is this loop doing?



# Input Loops

---

ACTIVITY



# Breaking Out of a Loop

---

- The break statement ends the current loop and jumps to the statement immediately following the loop
- It is like a loop test that can happen anywhere in the body of the loop

```
while True:
    line = input('> ')
    if line == 'done' :
        break
    print(line)
print('Done!')
```

```
> hello there
hello there
> finished
finished
> done
Done!
```




# Breaking Out of a Loop

---

- The break statement ends the current loop and jumps to the statement immediately following the loop
- It is like a loop test that can happen anywhere in the body of the loop

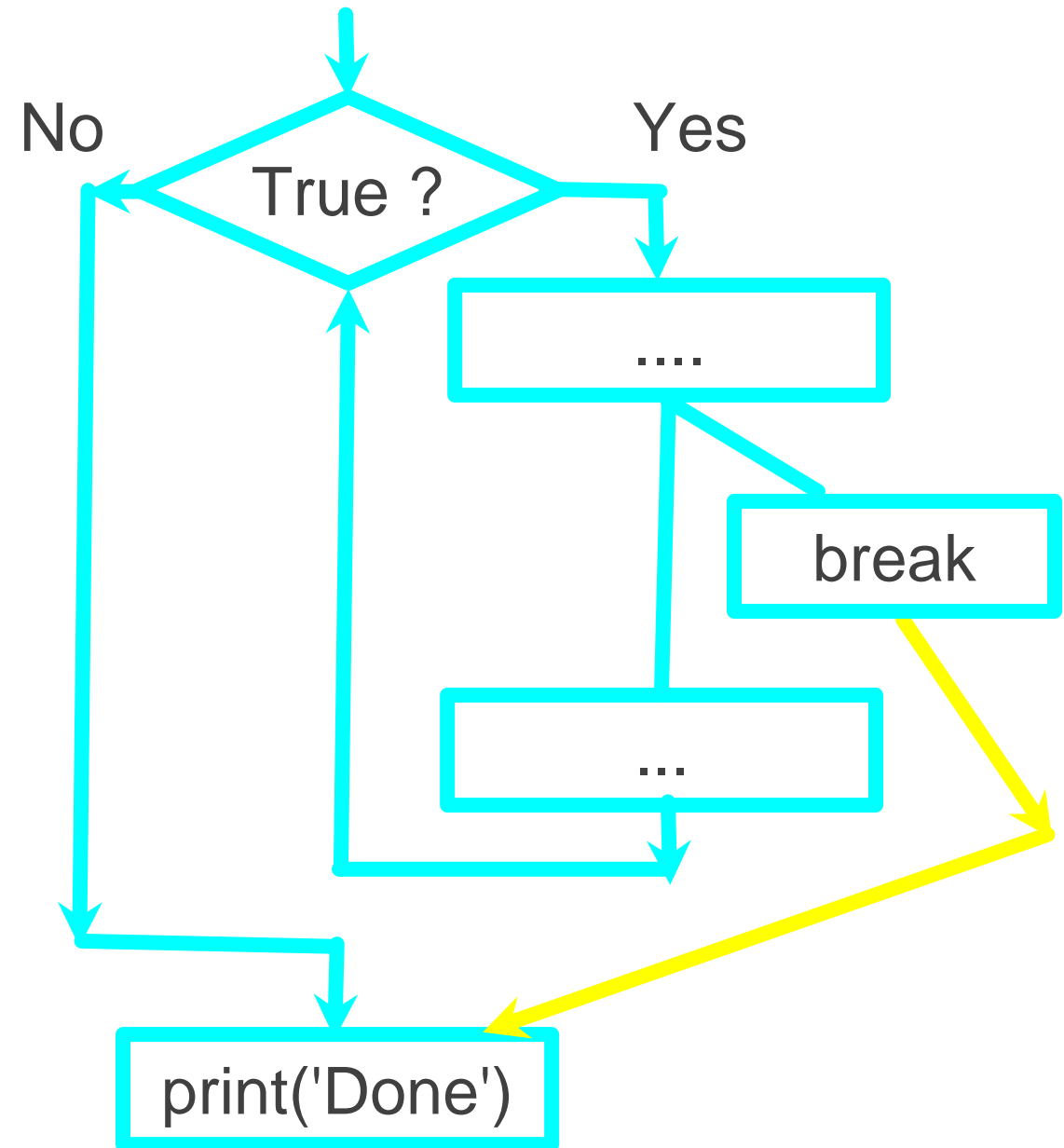
```
while True:
    line = input('> ')
    if line == 'done' :
        break
    print(line)
print('Done!')
```



```
> hello there
hello there
> finished
finished
> done
Done!
```



```
while True:
    line = input('> ')
    if line == 'done' :
        break
    print(line)
print('Done!')
```

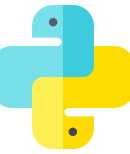




# Break levels

---

ACTIVITY



# Finishing an Iteration with continue

---

The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration

```
while True:
    line = input('> ')
    if line[0] == '#' :
        continue
    if line == 'done' :
        break
    print(line)
print('Done!')
```

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```



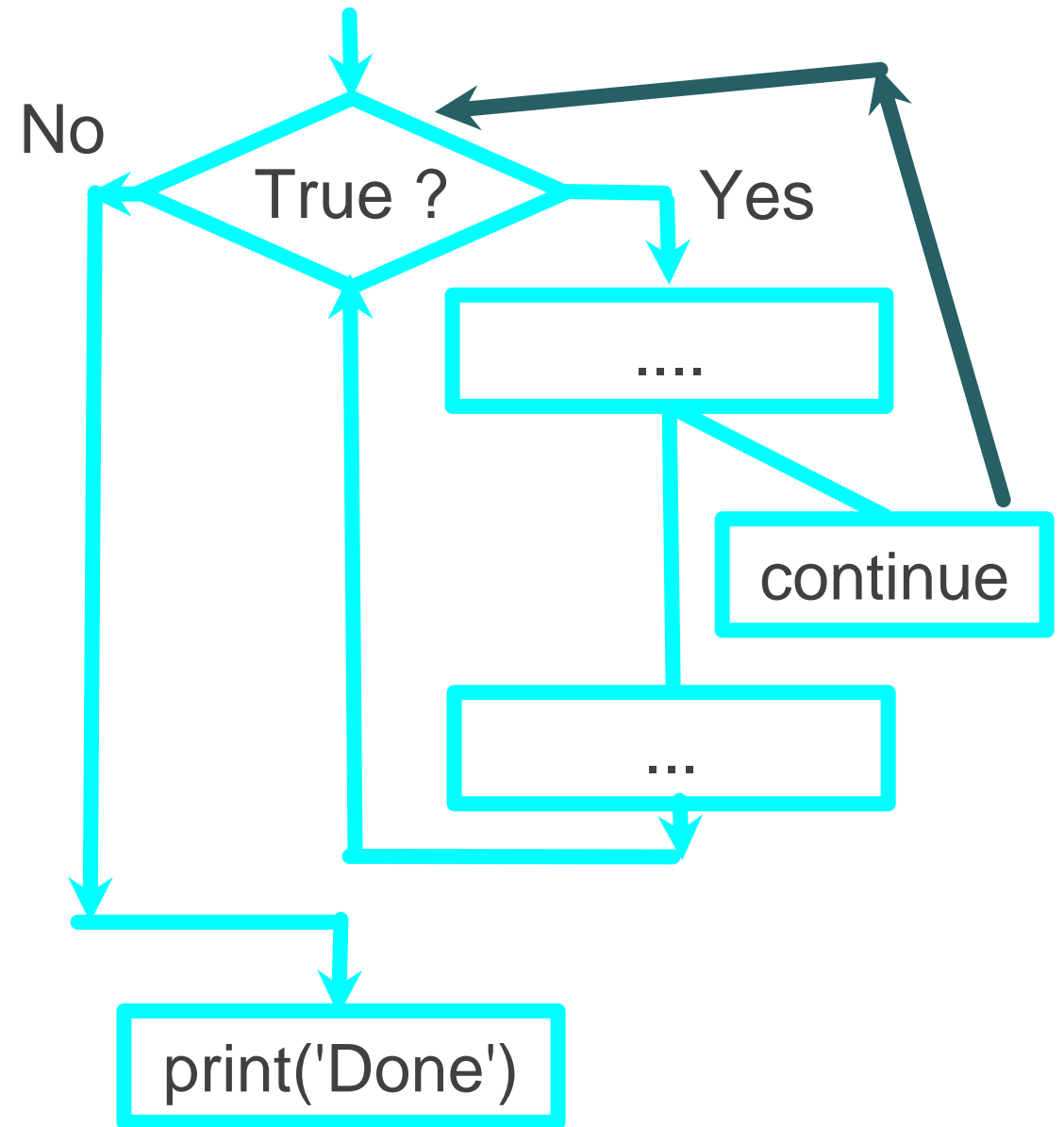
# Finishing an Iteration with continue

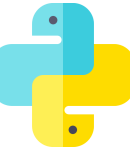
- The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')
```

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done' :
        break
    print(line)
print('Done!')
```





# Indefinite Loops

---

- While loops are called “indefinite loops” because they keep going until a logical condition becomes False
- The loops we have seen so far are pretty easy to examine to see if they will terminate or if they will be “infinite loops”
- Sometimes it is a little harder to be sure if a loop will terminate



# Definite Loops

---

ACTIVITY



# Definite Loops

---

- Quite often we have a list of items of the lines in a file - effectively a finite set of things
- We can write a loop to run the loop once for each of the items in a set using the Python for construct
- These loops are called “definite loops” because they execute an exact number of times
- We say that “definite loops iterate through the members of a set”





# A Simple Definite Loop

---

```
for i in [5, 4, 3, 2, 1] :  
    print(i)  
print('Blastoff!')
```

5  
4  
3  
2  
1  
Blastoff!



# A Definite Loop with Strings

---

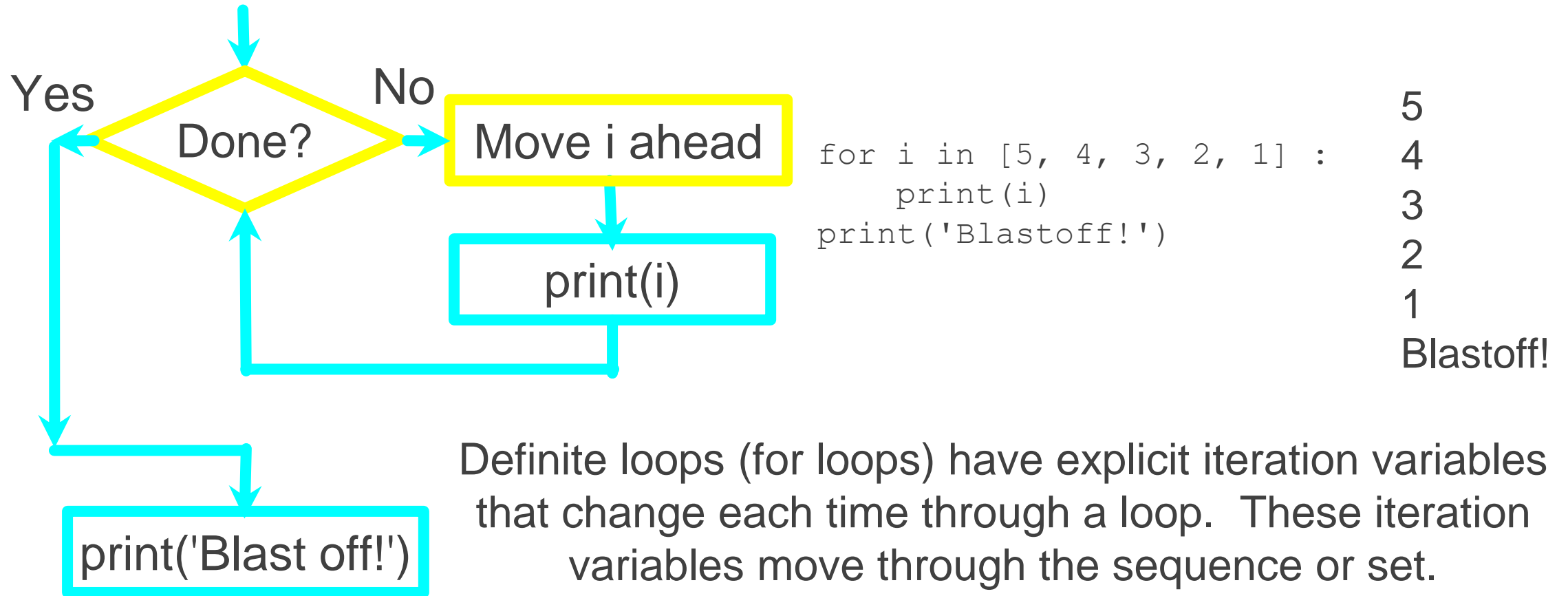
```
friends = ['Joseph', 'Glenn', 'Sally']  
for friend in friends :  
    print('Happy New Year:', friend)  
print('Done!')
```

Happy New Year: Joseph  
Happy New Year: Glenn  
Happy New Year: Sally

Done!



# A Simple Definite Loop





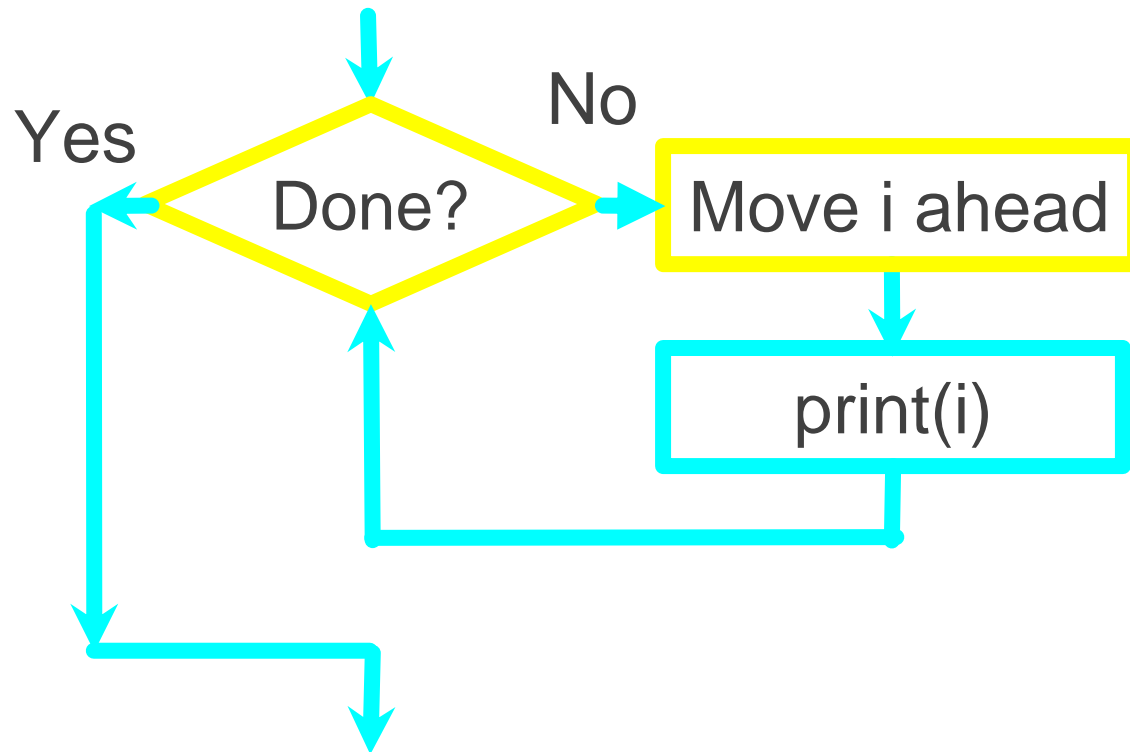
# Looking at in...

- The iteration variable “iterates” through the sequence (ordered set)
- The block (body) of code is executed once for each value in the sequence
- The iteration variable moves through all of the values in the sequence

Iteration variable

Five-element  
sequence

```
for i in [5, 4, 3, 2, 1] :  
    print(i)
```

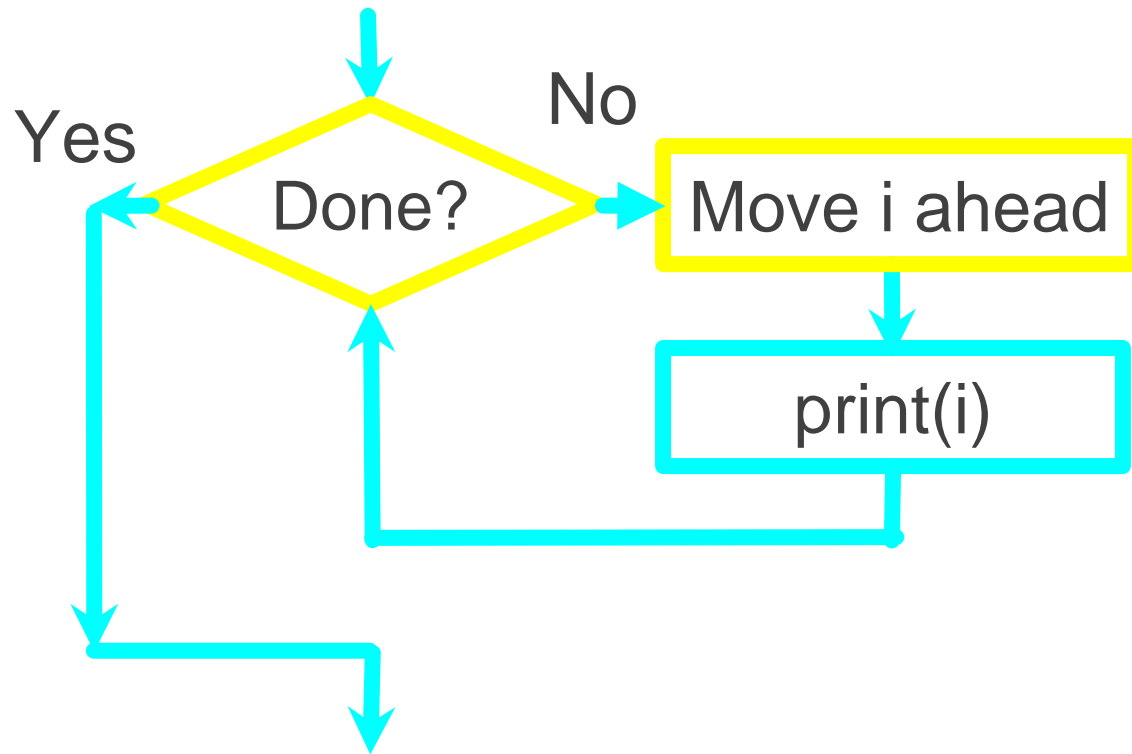


```
for i in [5, 4, 3, 2, 1] :  
    print(i)
```

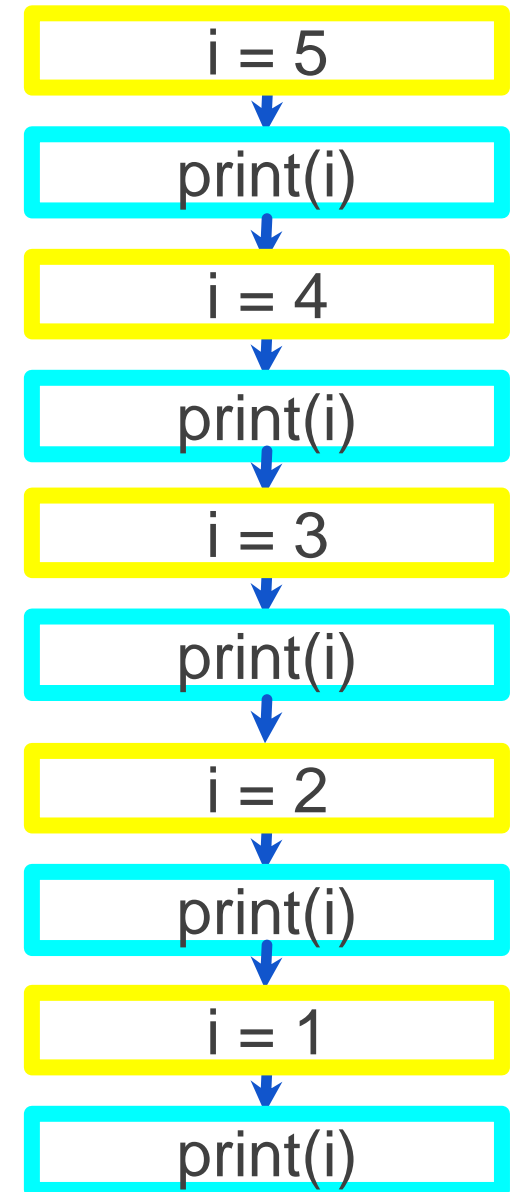
The iteration variable “iterates” through the sequence (ordered set)

The block (body) of code is executed once for each value in the sequence

The iteration variable moves through all of the values in the sequence



```
for i in [5, 4, 3, 2, 1] :  
    print(i)
```





# Loop Applications

---

ACTIVITY



# Loop Applications

---

- Control loop (Control variable/state variable)
- Indexed loop
- Counting loop
- Sentinel Loop
- Statistics
- Input Validation Loop
- 2-D index space
- Histogram





# Making “smart” loops

---

- The trick is “knowing” something about the whole loop when you are stuck writing code that only sees one entry at a time

Set some variables to initial values

for thing in data:

Look for something or do something to each entry separately, updating a variable

Look at the variables



# Looping Through a Set

---

```
print('Before')
for thing in [9, 41, 12, 3, 74, 15] :
    print(thing)
print('After')
```

\$ python basicloop.py

Before

9

41

12

3

74

15

After



# Finding the Largest Value

---

```
largest_so_far = -1
print('Before', largest_so_far)
for the_num in [9, 41, 12, 3, 74, 15] :
    if the_num > largest_so_far :
        largest_so_far = the_num
        print(largest_so_far, the_num)

print('After', largest_so_far)
```

\$ python largest.py

Before -1

9 9

41 41

41 12

41 3

74 74

74 15

After 74

We make a variable that contains the largest value we have seen so far. If the current number we are looking at is larger, it is the new largest value we have seen so far.



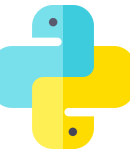
# Counting in a Loop

---

```
zork = 0
print('Before', zork)
for thing in [9, 41, 12, 3, 74, 15] :
    zork = zork + 1
    print(zork, thing)
print('After', zork)
```

```
$ python countloop.py
Before 0
1 9
2 41
3 12
4 3
5 74
6 15
After 6
```

To count how many times we execute a loop, we introduce a counter variable that starts at 0 and we add one to it each time through the loop.



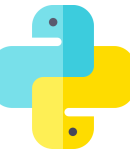
# Summing in a Loop

---

```
zork = 0
print('Before', zork)
for thing in [9, 41, 12, 3, 74, 15] :
    zork = zork + thing
    print(zork, thing)
print('After', zork)
```

```
$ python countloop.py
Before 0
9 9
50 41
62 12
65 3
139 74
154 15
After 154
```

To add up a value we encounter in a loop, we introduce a sum variable that starts at 0 and we add the value to the sum each time through the loop.



# Finding the Average in a Loop

---

```
count = 0
sum = 0
print('Before', count, sum)
for value in [9, 41, 12, 3, 74, 15] :
    count = count + 1
    sum = sum + value
    print(count, sum, value)
print('After', count, sum, sum / count)
```

```
$ python averageloop.py
```

```
Before 0 0
```

```
1 9 9
```

```
2 50 41
```

```
3 62 12
```

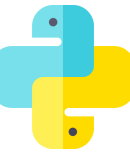
```
4 65 3
```

```
5 139 74
```

```
6 154 15
```

```
After 6 154 25.666
```

An average just combines the counting and sum patterns and divides when the loop is done.



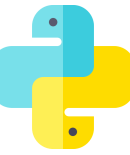
# Filtering in a Loop

---

```
print('Before')
for value in [9, 41, 12, 3, 74, 15] :
    if value > 20:
        print('Large number',value)
print('After')
```

```
$ python search1.py
Before
Large number 41
Large number 74
After
```

We use an if statement in the loop to catch / filter the values we are looking for.



# Search Using a Boolean Variable

---

```
found = False
print('Before', found)
for value in [9, 41, 12, 3, 74, 15] :
    if value == 3 :
        found = True
    print(found, value)
print('After', found)
```

```
$ python search1.py
Before False
False 9
False 41
False 12
True 3
True 74
True 15
After True
```

If we just want to search and know if a value was found, we use a variable that starts at False and is set to True as soon as we find what we are looking for.





# How to Find the Smallest Value

---

```
largest_so_far = -1
print('Before', largest_so_far)
for the_num in [9, 41, 12, 3, 74, 15] :
    if the_num > largest_so_far :
        largest_so_far = the_num
        print(largest_so_far, the_num)

print('After', largest_so_far)
```

\$ python largest.py

Before -1

9 9

41 41

41 12

41 3

74 74

74 15

After 74

How would we change this to make it find the smallest value in the list?



# Finding the Smallest Value

---

```
smallest_so_far = -1
print('Before', smallest_so_far)
for the_num in [9, 41, 12, 3, 74, 15] :
    if the_num < smallest_so_far :
        smallest_so_far = the_num
    print(smallest_so_far, the_num)

print('After', smallest_so_far)
```

We switched the variable name to `smallest_so_far` and switched the `>` to `<`



# Finding the Smallest Value

---

```
smallest_so_far = -1
print('Before', smallest_so_far)
for the_num in [9, 41, 12, 3, 74, 15] :
    if the_num < smallest_so_far :
        smallest_so_far = the_num
    print(smallest_so_far, the_num)

print('After', smallest_so_far)
```

\$ python smallbad.py

Before -1

-1 9

-1 41

-1 12

-1 3

-1 74

-1 15

After -1

We switched the variable name to `smallest_so_far` and switched the `>` to `<`



# Finding the Smallest Value

---

```
smallest = None
print('Before')
for value in [9, 41, 12, 3, 74, 15] :
    if smallest is None :
        smallest = value
    elif value < smallest :
        smallest = value
    print(smallest, value)
print('After', smallest)
```

\$ python smallest.py

Before

9 9

9 41

9 12

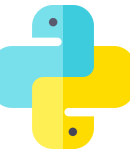
3 3

3 74

3 15

After 3

We still have a variable that is the smallest so far. The first time through the loop smallest is None, so we take the first value to be the smallest.



# The **is** and **is not** Operators

---

```
smallest = None
print('Before')
for value in [3, 41, 12, 9, 74, 15] :
    if smallest is None :
        smallest = value
    elif value < smallest :
        smallest = value
    print(smallest, value)

print('After', smallest)
```

Python has an **is** operator that can be used in logical expressions

Implies “is the same as”

Similar to, but stronger than ==

**is not** also is a logical operator