Conditional Steps and Operators

Learning Objects

1. **Identify conditional operators** used in programming and explain their functions in controlling program flow.

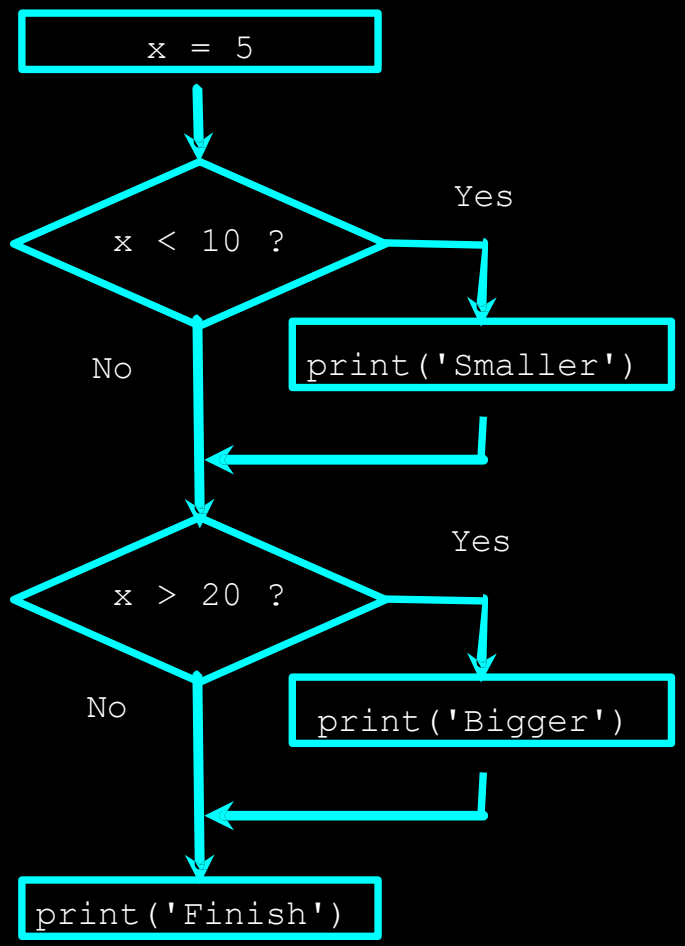
2. **Explain how conditional statements** (like `if`, `else`, and `elif`) are structured and applied to make decisions in a program.

3. **Implement conditional steps in code** to execute specific actions based on different conditions.

4. **Design simple programs** that incorporate conditional operators to solve basic decision-making problems.

5. **Define and implement functions** to organize code into reusable blocks, improving readability and efficiency.

6. **Apply parameters and return values** in functions to handle data dynamically and produce meaningful outputs



Conditional Steps

Program:

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

print('Finish')
```

Output:

```
Smaller
Finish
```

Notice the

```
if
if
```

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

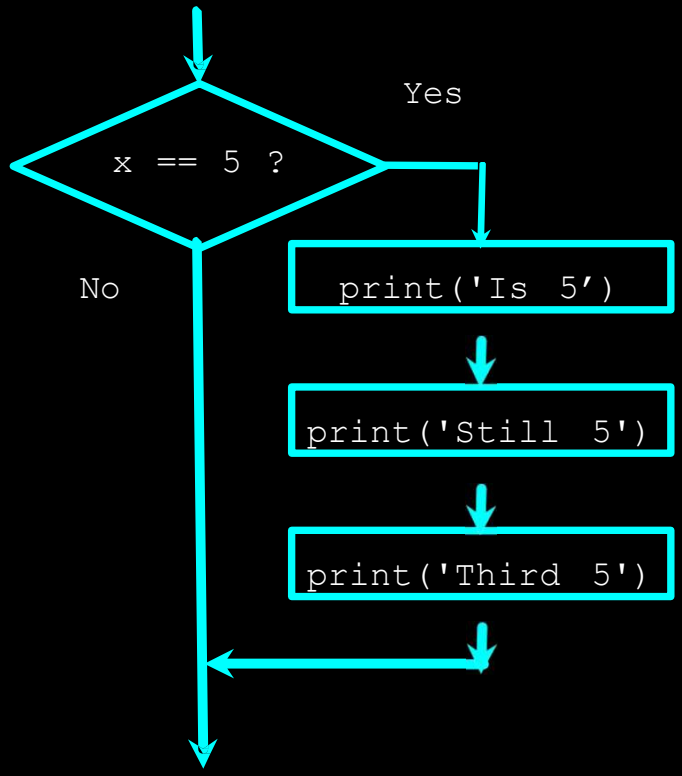
Remember: "=" is used for assignment.

http://en.wikipedia.org/wiki/George_Boole

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')
```

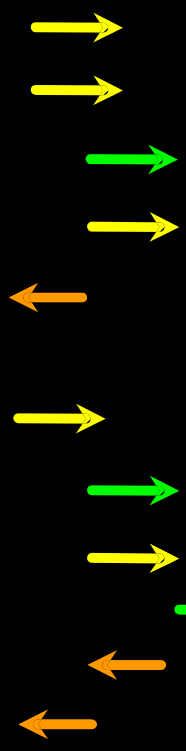
```
Before 5
Is 5
Is Still 5
Third 5
Afterwards 5
Before 6
Afterwards 6
```



Note, in these **`if`** statements, the **`else`** clause is omitted entirely

increase / maintain after if or for

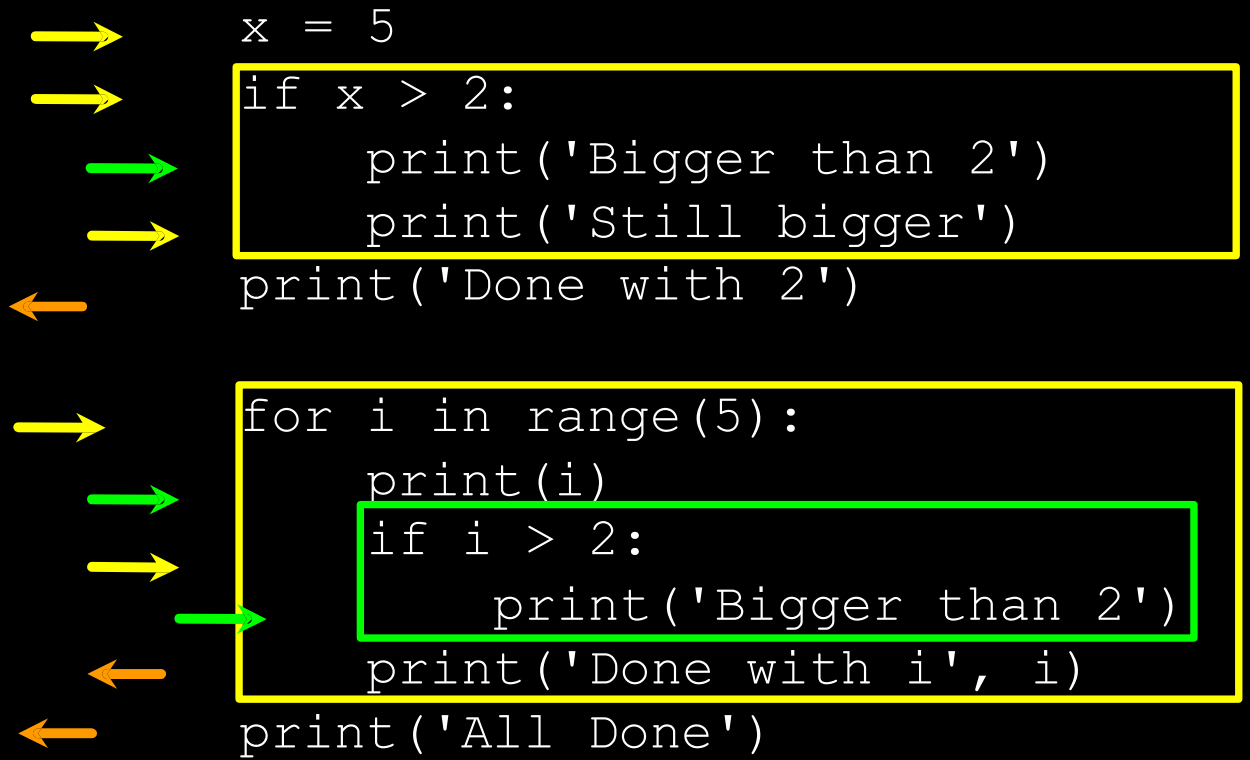
decrease to indicate end of block



```
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

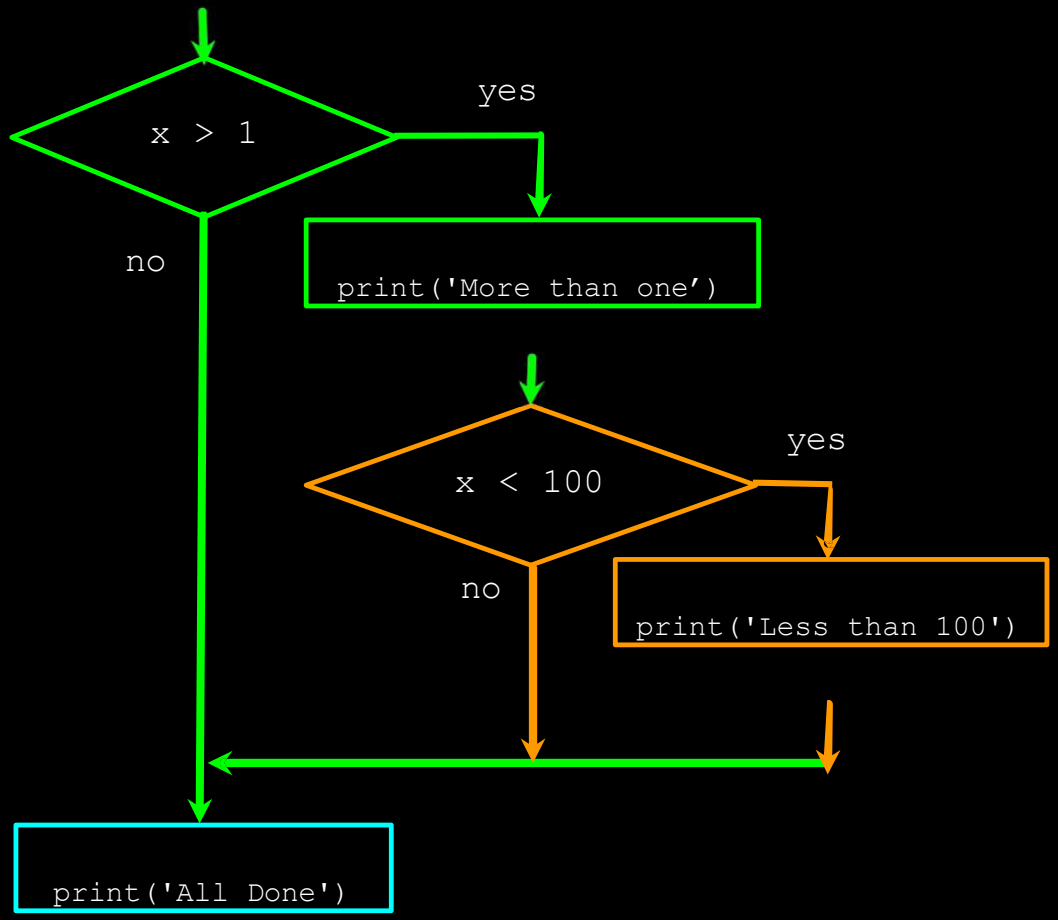


```
x = 5
if x > 2:
    print('Bigger than 2')
    print('Still bigger')
print('Done with 2')

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

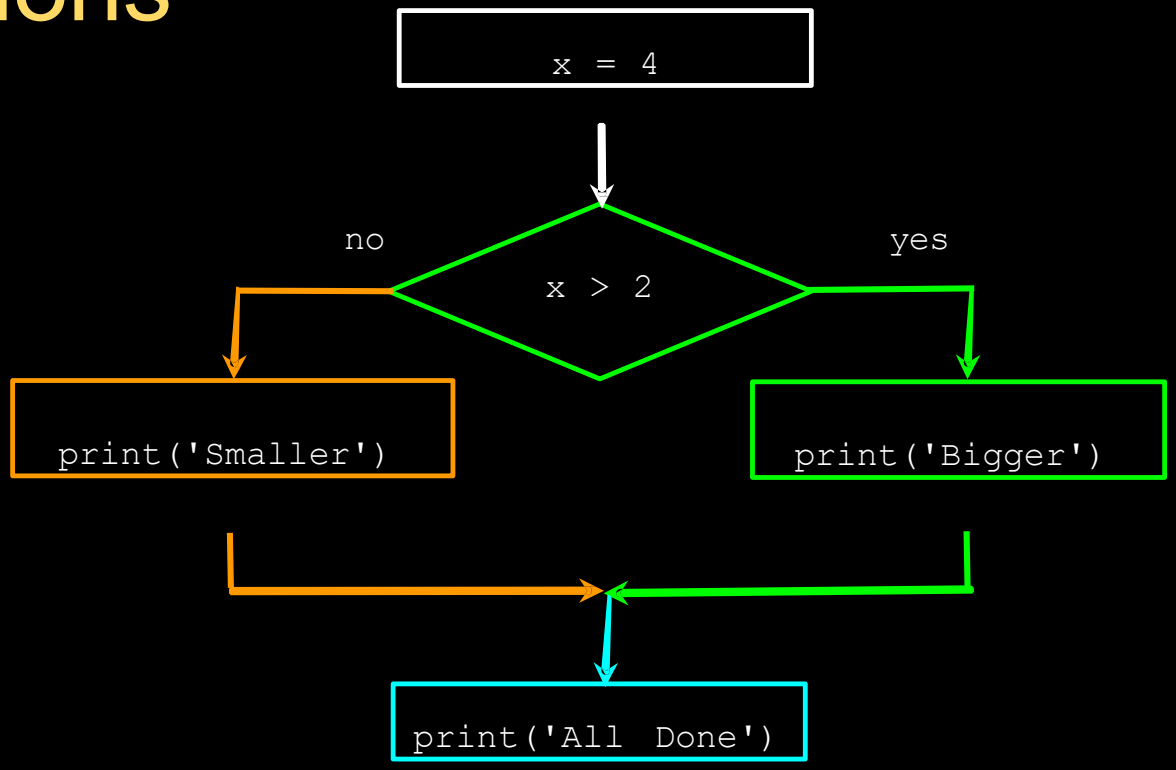
Nested Decisions

```
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

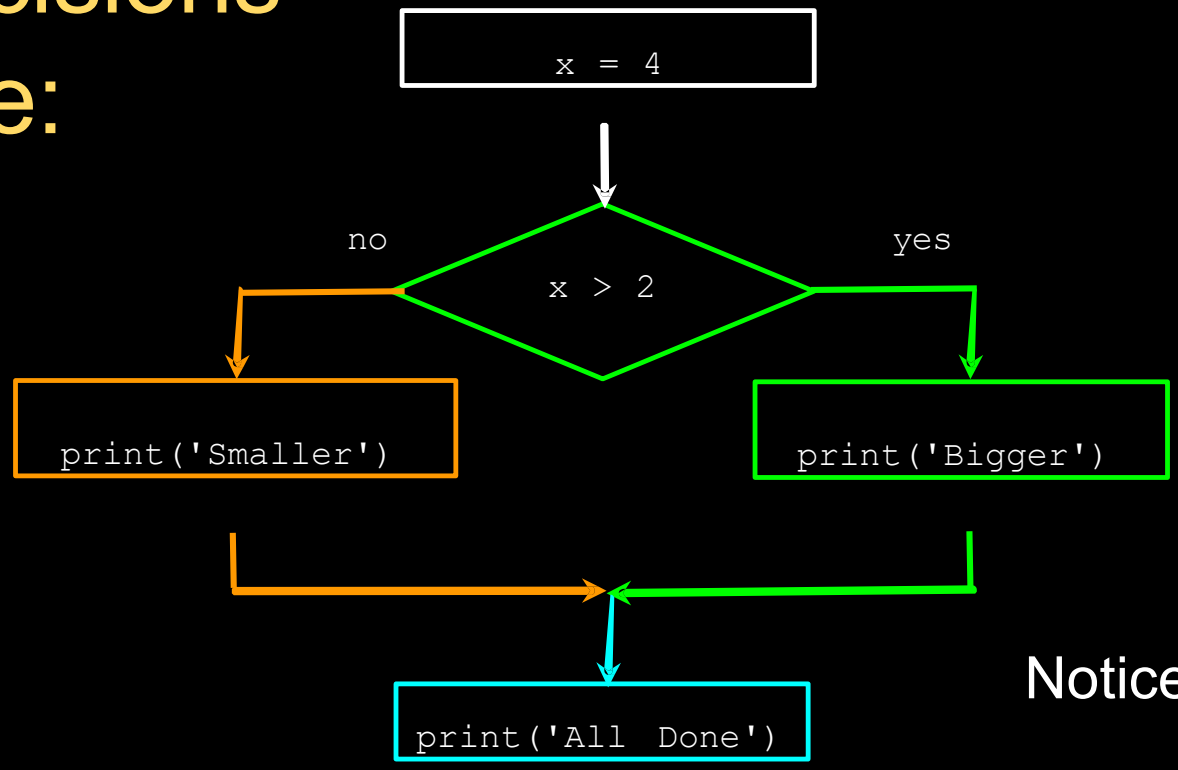


Two-way Decisions with else:

```
x = 4

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

print('All done')
```



Notice the

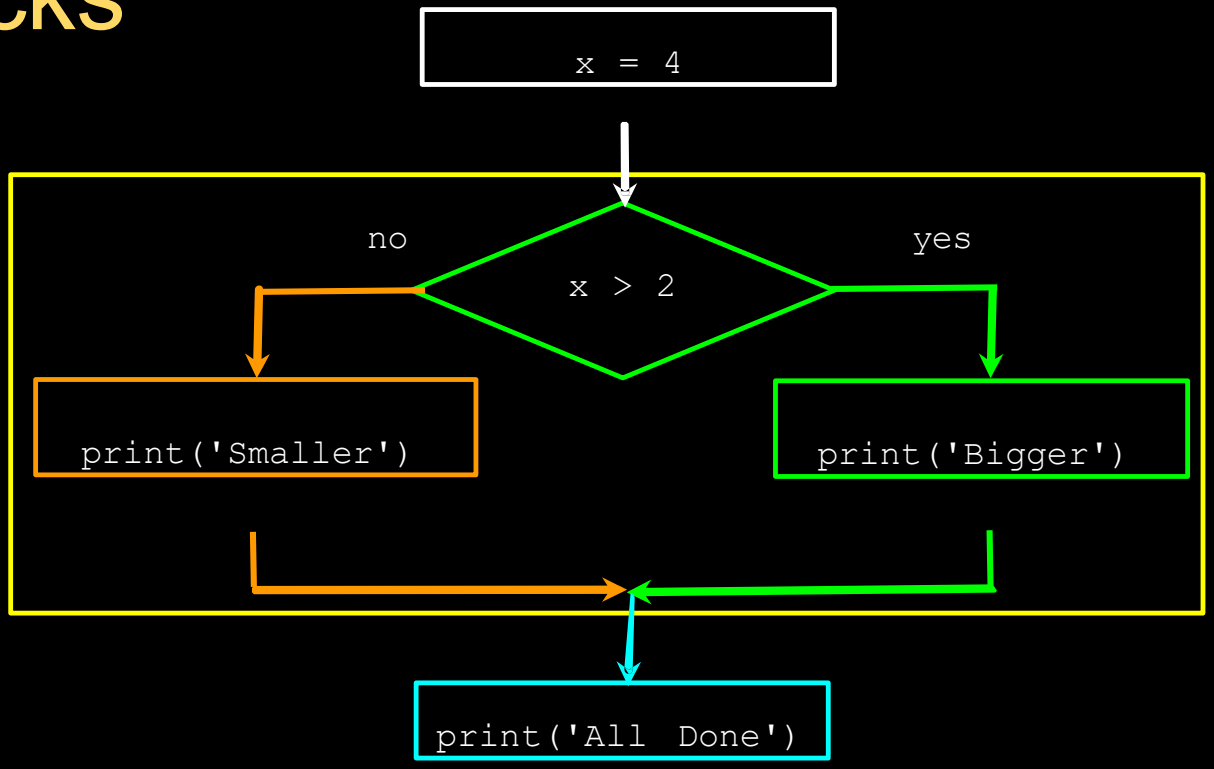
`if`
`else`

Visualize Blocks

```
x = 4

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

print('All done')
```



Important to only use tw—way decision when you know all current conditions and future conditions– Defensive programming

e.g., of Defensive Programming

```
if x % 2 == 0:
    print('even')
else:
    print('odd')
```

- Binary Decision
- No need for defensive programming

```
# future color
color = 'green'

if color == 'red':
    print('red')
else:
    print('blue')
```

- Didn't know all future colors
- And a new color is introduced. (green)
- Non-Defensive

```
color = 'green'

if color == 'red':
    print('red')
elif color == 'blue':
    print('blue')
else:
    raise ValueError(f"unknown color '{color}'")
```

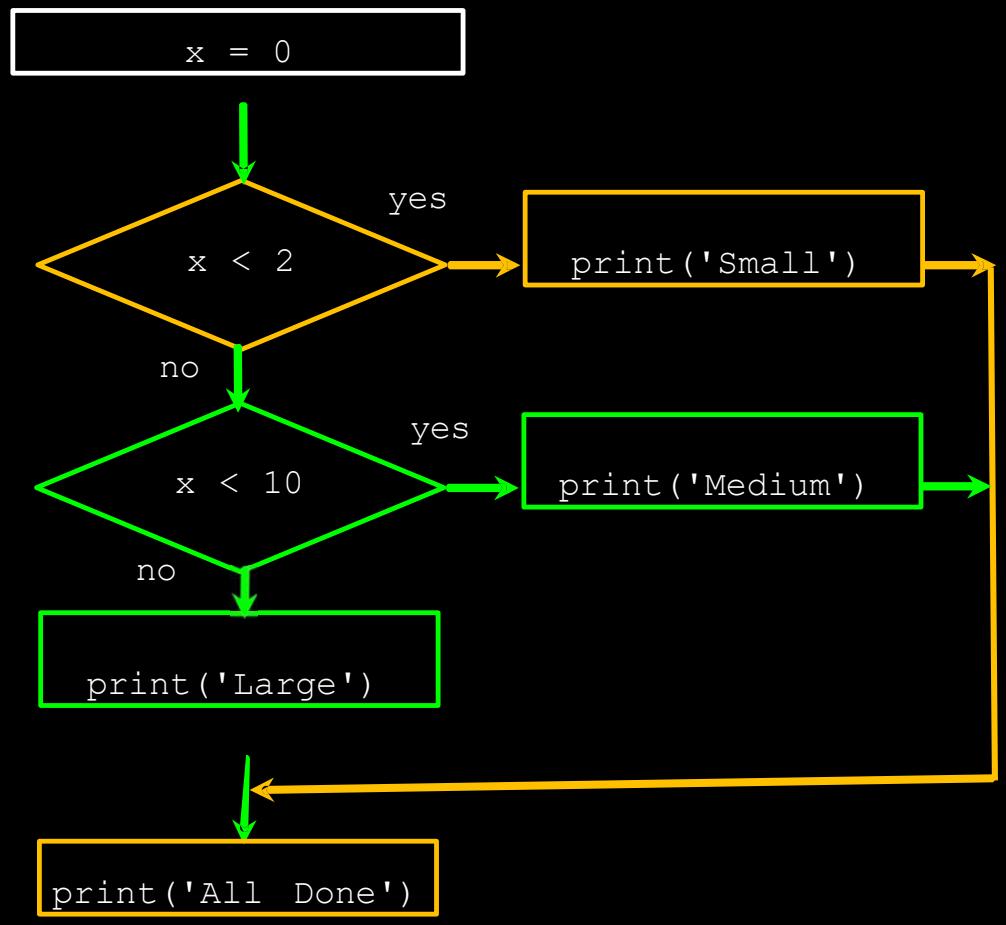
- Defensive programming
- Now will be alerted when a new color is introduced

raise Feel free to read more on raise: https://realpython.com/python-raise-exception/

More Conditional Structures…

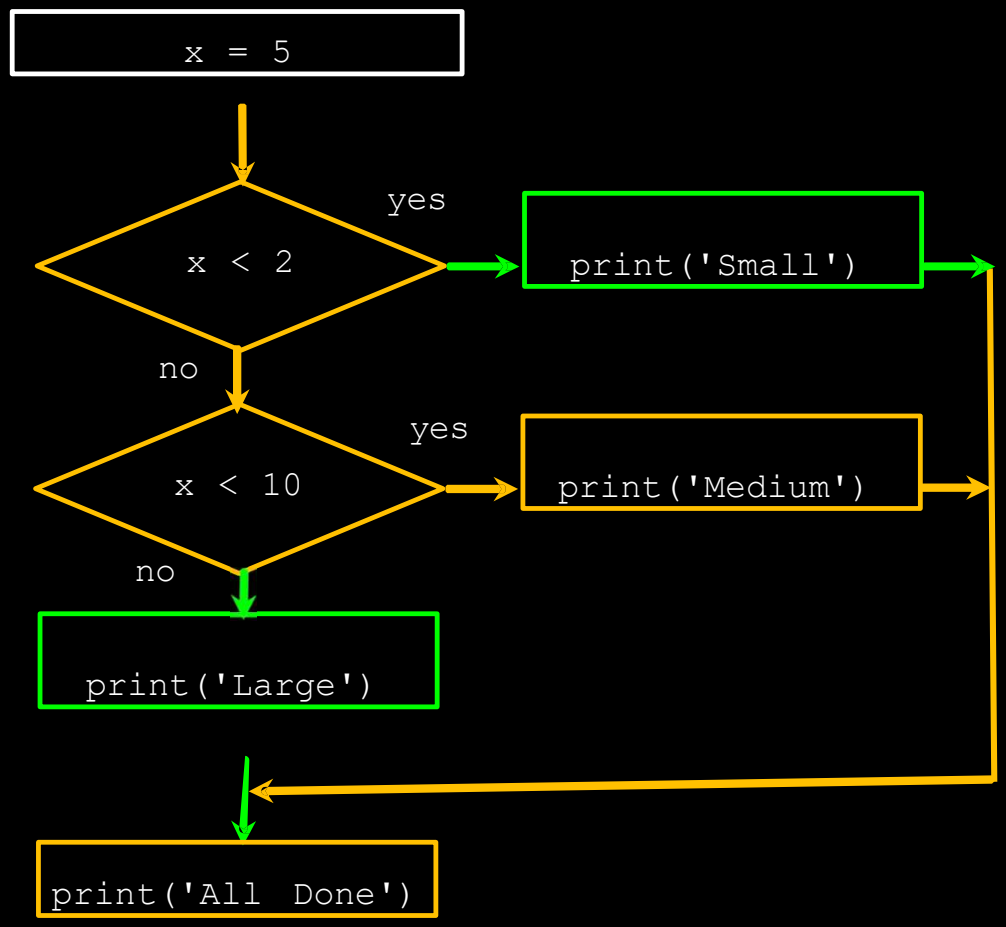
Multi-way (Chained Conditional)

```
x = 0
if x < 2:
    print('Small')
elif x < 10:
    print('Medium')
else:
    print('Large')
print('All Done')
```



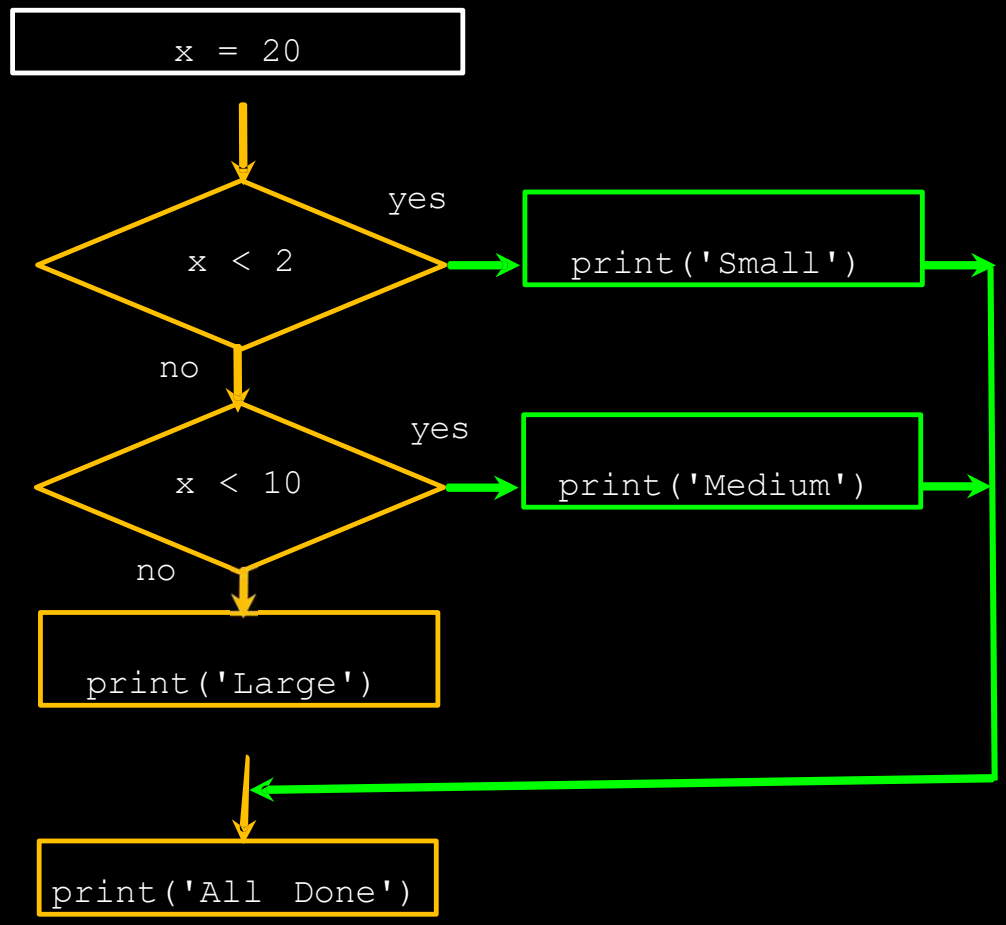
Multi-way (Chained Conditional)

```
x = 5
if x < 2:
    print('Small')
elif x < 10:
    print('Medium')
else:
    print('Large')
print('All Done')
```



Multi-way (Chained Conditional)

```
x = 20
if x < 2:
    print('Small')
elif x < 10:
    print('Medium')
else:
    print('Large')
print('All Done')
```



Multi-way (Chained Conditional)

```
# 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')
```

Example:

An example which uses if and else to split up a list of accession names into two different files – accessions that start with "a" go into the first file, and all other accessions go into the second file.

```
file1 = open("one.txt", "w")
file2 = open("two.txt", "w")
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for accession in accs:
    if accession.startswith('a'):
        file1.write(accession + "\n")
    else:
        file2.write(accession + "\n")
```

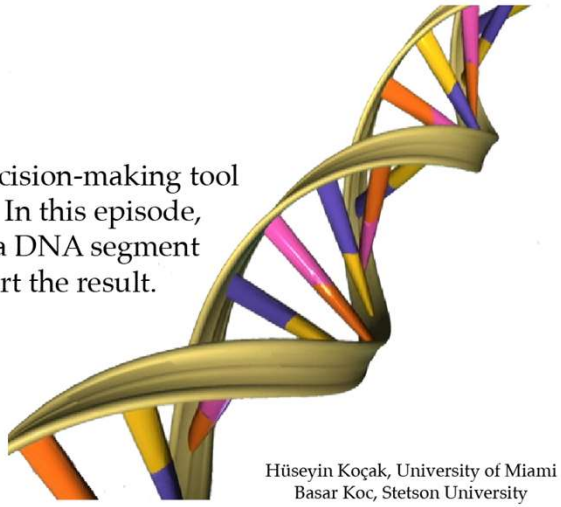
What if we have more than two possible branches? For example, we want three files of accession names: ones that start with "a", ones that start with "b", and all others.

```
file1 = open("one.txt", "w")
file2 = open("two.txt", "w")
file3 = open("three.txt", "w")
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for accession in accs:
    if accession.startswith('a'):
        file1.write(accession + "\n")
    elif accession.startswith('b'):
        file2.write(accession + "\n")
    else:
        file3.write(accession + "\n")
```

Example: Codon1

```
if (codon1 == 'ATG'):
```

The `if-else` statement is the basic decision-making tool for choosing between two alternatives. In this episode, we will determine if the first codon in a DNA segment is the start codon ATG or not, and report the result.



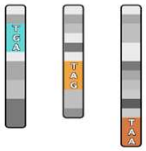
Hüseyin Koçak, University of Miami
Basar Koc, Stetson University

We will determine if the first codon in a DNA segment is the start codon ATG or not, and report the result.

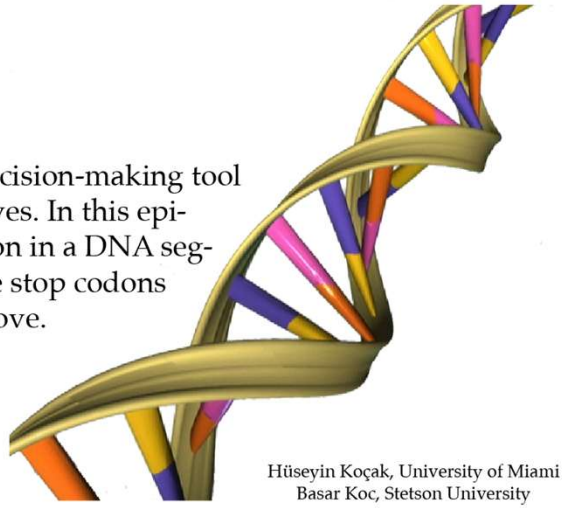
if_else.py

Example: Codon1

```
elif ((codon1 == 'TAA') or
      (codon1 == 'TAG') or
```



The `if-elif` statement is the basic decision-making tool for choosing among multiple alternatives. In this episode, we will determine if the last codon in a DNA segment is the start codon ATG, one of the stop codons (TAA, TAG, TGA), or neither of the above.



Hüseyin Koçak, University of Miami
Basar Koc, Stetson University

We will determine if the last codon in a DNA segment is the start codon ATG, one of the stop codons (TAA, TAG, TGA), or neither of the above.

if_elif.py

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 whenever we want
    - These are the building blocks of Python Modules, Packages, and OPP (methods)
- We treat the built-in function names as "new" reserved words (i.e., we avoid them as variable names)

Built-in Functions (1)

```
>>> any([False, False, False])
False
>>> any([False, True, False])
True

>>> any(['bar' == 'baz', len('foo') == 4, 'qux' in {'foo', 'bar', 'baz'}])
False
>>> any(['bar' == 'baz', len('foo') == 3, 'qux' in {'foo', 'bar', 'baz'}])
True
```

- Each of these built-in functions performs a specific task. The code that accomplishes the task is defined somewhere, but you don't need to know where or even how the code works. All you need to know about is the function's **interface**:
  - What arguments (if any) it takes
  - What values (if any) it returns

https://realpython.com/defining-your-own-python-function/

Built-in Functions (2)

- Then you call the function and pass the appropriate arguments. Program execution goes off to the designated body of code and does its useful thing
- When the function is finished, execution returns to your code where it left off. The function may or may not return data for your code to use, as the examples on the previous slide

- When you define your own Python function, it works just the same
- From somewhere in your code, you'll call your Python function and program execution will transfer to the body of code that makes up the function
- When the function is finished, execution returns to the location where the function was called
- Depending on how you designed the function's ***interface***, data may be passed in when the function is called, and return values may be passed back when it finishes

https://realpython.com/defining-your-own-python-function/

Why Functions Are Great!

- A program can use the function's code ***by just calling its name***
- Referred to as "Invocation" of the function

- Think of a function as a smaller program **within a program**
  - Just as you run programs to get results**, your programs call functions to get results**
  - You can use it in a program by simply passing the values in and knowing what values to expect to be passed back
- *IMPORTANTLY: Functions can call other functions*
- By writing a set of functions
  - Each which does **one** thing well
  - Each that can be tested independently
  - You can combine them in various ways to make new functions

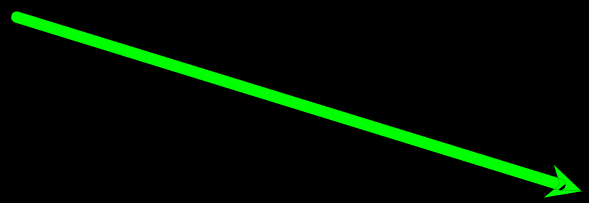
Implementing Our Own Functions

- 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.')
```

```
def 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')
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
```

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

```
depth = meter_to_feet(20)
```

Parameters

A parameter is a variable which we use in the function definition

It's a "handle" that allows the code in the function to access the arguments for a **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
>>>
```

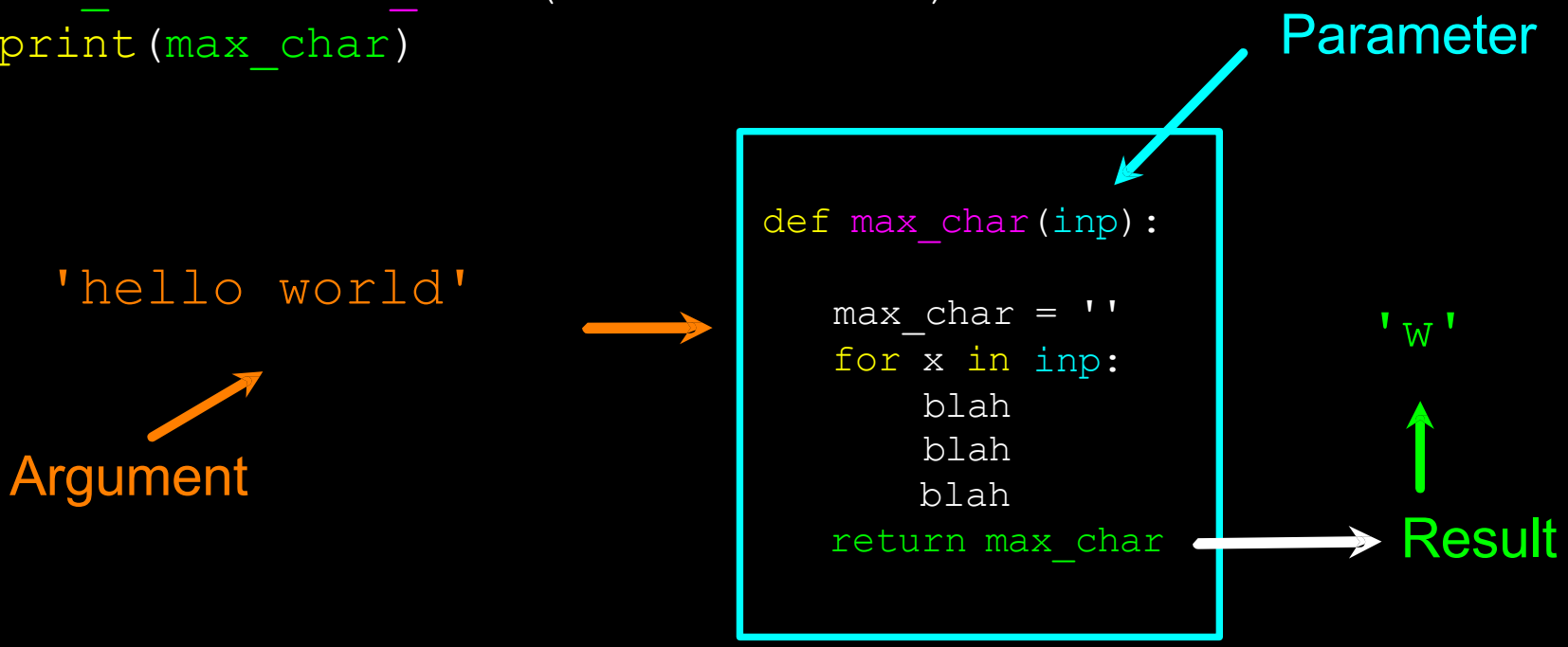
Return Value

- A function that produces a `result` (or `return value`) is a "fruitful" function
- **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
>>>
```

Arguments, Parameters, and Results

```
>>> max_char = max_char('hello world')
>>> print(max_char)
w
```



Multiple Parameters / Arguments

- We can define more than one parameter in the function definition
- Simply add more arguments when we call the function
- In Python, *using this style* we must match the number and order of arguments and parameters
- So, a function is a piece of code that has a specific use (or algorithm) but may be ***parameterized***

```
def add_two(val1, val2):
    added = val1 + val2
    return added

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

```
8
```

Void (non-fruitful) Functions

- When a function does not return a value, we call it a "void" function
- Void functions are "not fruitful"
- One “gotcha” – all Python functions return a value, whether they contain a `return` statement or not
- Functions without a return hand back a special object, denoted `None` which is a `NoneType`
- A common problem is writing a *value-returning function* and ***forgetting the `return`***!
- **TIP:** If your value-returning functions is not working, check to make sure you remembered to include the `return`!

Your Turn:

Let's create our `get_at_content` function. For this function, the input is going to be a single DNA sequence, and the output is going to be a decimal number. To translate these into Python terms: the function will take a single argument of type *string*, and will return a value of type *number*1