Computing (ES 112)

Yogesh K. Meena
Shouvick Mondal

August 2024

Computer Science & Engineering
IIT Gandhinagar

Variables, Expressions, and Statements

Operator Precedence in Python

-13 Decimal

01101 5-bit binary (absolute) for +13

10010 1's complement (flip)

10011 2's complement (1's complement + 1)

In 2's complement notation, a non-negative number is represented by its absolute binary representation; in this case, the MSB is 0. Negative numbers are in 2's complement form, where the MSB is 1

This 1 is in decimal (base 10)

$= -1*(1101)_2$

$= -1*13$

$= -13$

To get back original value from 2's complement representation, apply 2's complement operation on the 2's complement itself

Operator Precedence in Python

Operator	Description
`+x, -x, ~x`	Positive, negative, bitwise NOT

```
>>> x=13 #this is 13 in decimal (base 10)
>>> ~x  #bitwise NOT of x equals to -(x+1) in base 10
-14
```

01101 5-bit binary (absolute) for 13 (in base 10)

10010 Invert each bit including the sign bit (MSB)

This is viewed as 2's complement even it was not obtained by regular 2's complement

Final value

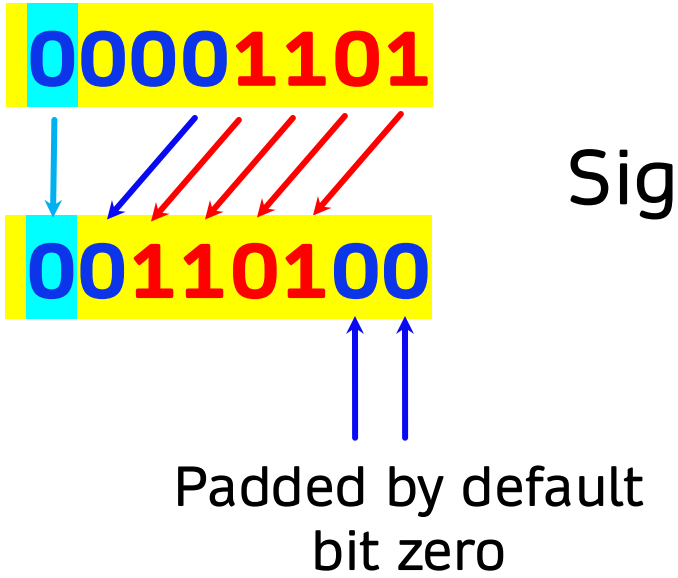
$=-1*(1110)_2$

$=-1*14$

Operator Precedence in Python

Operator	Description
<<, >>	Left shift, right shift

```
>>> 13<<2 #shift all bits by 2 positions to the left
>>> 52
```



8-bit binary (absolute) for 13

Sign bit (MSB) is copied to preserve signedness

Final value

$=+1*(110100)_2$

$=+1*52$

Each left shit by one position means multiplication by 2

Operator Precedence in Python

Operator	Description
<<, >>	Left shift, right shift

```
>>> -13<<2 #shift all bits by 2 positions to the left
>>> 52
```

What if we use 5-bit for data storage?

11110011

11001100

Padded by default bit zero

8-bit 2's complement of -13

Sign bit (MSB) is copied to preserve signedness

Final value

$=-1*(110100)_2$

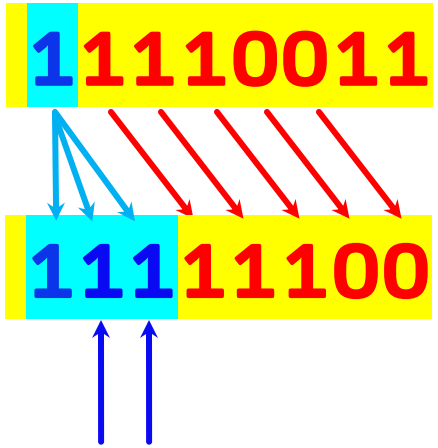
$=-1*52$

Each left shit by one position means multiplication by 2

Operator Precedence in Python

Operator	Description
<<, >>	Left shift, right shift

```
>>> -13>>2 #shift all bits by 2 positions to the right
>>> -4
```



Padded by copies of the sign bit

8-bit 2's complement of -13

Sign bit (MSB) is copied to preserve signedness

Final value

$=-1*(100)_2$

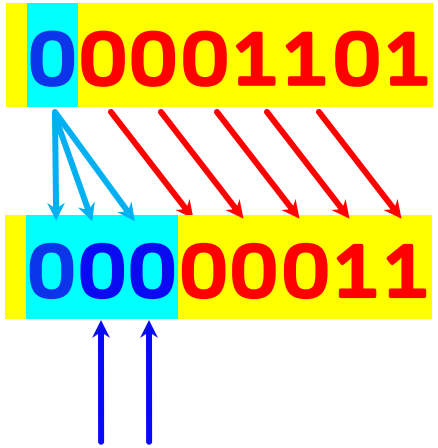
$=-1*4$

Each right shit by one position means division by 2

Operator Precedence in Python

Operator	Description
<<, >>	Left shift, right shift

```
>>> 13>>2 #shift all bits by 2 positions to the right
>>> 3
```



Padded by copies of the sign bit

8-bit binary (absolute) for 13

Sign bit (MSB) is copied to preserve signedness

Final value

$=+1*(11)_2$

$=+1*3$

Each right shit by one position means division by 2

Operator Precedence in Python

Operator	Description
&	Bitwise AND
^	Bitwise XOR (eXclusive OR)
|	Bitwise OR (inclusive OR)

priority

Truth table AND

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Truth table XOR

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

Truth table OR

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	1

For numbers represented in multiple bits (>1), each bit position is operated individually, hence the name 'bitwise'. There is no notion of carry here...

Operator Precedence in Python

Operator	Description
&	Bitwise AND
^	Bitwise XOR (eXclusive OR)
|	Bitwise OR (inclusive OR)

priority

Truth table AND

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

```
>>> 6&4
4
>>> 6&-4
4
>>> -6&-4
-8
```

```
>>> -6&4
0
>>> 6&4
4
```

Why & how does this happen?

Think about how numbers represented in memory. For example, in 8-bit representation (2's complement), both operands are aligned to same number of bits and operated bitwise.

Operator Precedence in Python

Operator	Description
`in, not in, is, is not, <, <=, >, >=, !=, ==`	Comparisons, including membership tests and identity tests

```
>>> 5<3
False
>>> 0>-7
True
>>> 0==7
False
>>> 0==0
True
>>> 0!=1
True
```

```
>>> 1!=0
True
>>> 1>=0
True
>>> 1<=0
False
>>> (0==0) is True
True
>>> (0==7) is not True
True
```

```
>>> 5 in [1,2]
False
>>> 5 in [1,5,8,3]
True
>>> 5 not in [1,5,8,3]
False
```

L=[1,5,8,3] is a list of integers in the stored in the order of their appearance. L[0] (zeroth item of L) means 1

What is L[2], L[6]?

Operator Precedence in Python

Operator	Description
not	Boolean NOT
and	Boolean AND
or	Boolean OR

priority

Truth table AND

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

Truth table OR

a	b	a and b
False	False	False
False	True	True
True	False	True
True	True	True

Bitwise operates at the bit level (numeric), whereas Boolean operators at the Truth level (logic).

```
>>> 5 & True
1
>>> 5 & False
0
>>> 5 and True
True
>>> 5 and False
False
```

Why & how does this happen?

Operator Precedence in Python

Operator	Description
not	Boolean NOT
and	Boolean AND
or	Boolean OR

priority

Truth table AND

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

Truth table OR

a	b	a and b
False	False	False
False	True	True
True	False	True
True	True	True

Bitwise operators **should be** used at the bit level (numeric), whereas Boolean operators at the Truth level (boolean).

```
>>> (5%2!=0) and (0==0)
True
>>> (5%2!=0) or (0!=0)
True
>>> not False
True
```

Operator precedence: best practices

- Remember the rules top to bottom
- When writing code - use parentheses
- When writing code - keep mathematical expressions simple enough that they are easy to understand
- Break long series of mathematical operations up to make them more clear

priority

Parenthesis
Power
Multiplication
Addition
Left to Right

These are <u>not the only</u> operators in Python. Identify the operators in the following?

- `print("hello world")`
- `array[i]`
- `a <= b`
- `y = -b`
- `c = a & b`

Anything that can operate upon / manipulate data (operand) is essentially an operator.

The notion of "type" in Python

```
>>> a=5
>>> b=6
>>> c=a+b
>>> print(c)
11
>>> b='6'
>>> c=a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Type can be perceived as a constraint (semantic) as to what kind of data should be expected by an operator to carry out its fundamental operation.

In lay terms, we cannot compare apples and oranges in Python...

Type matters in Python

- In Python, *variables*, *literals*, and *constants* have a "type"
- Python *knows the difference* between an integer number and a string
- Some operations are prohibited
- For example "**+**" means "**addition**" if something is a **number** and "**concatenate**" if something is a **string**

```
>>> a=5
>>> b=6
>>> c=a+b
>>> print(c)
11
>>> a='5'
>>> b='6'
>>> c=a+b
>>> print(c)
56
```

concatenate = put together

Querying the type of data

```
>>> eee = 'hello ' + 'there'
>>> eee = eee + 1
Traceback (most recent call last):  File "<stdin>", line 1, in
<module>TypeError: Can't convert 'int' object to str implicitly
>>> type(eee)
<class'str'>
>>> type('hello')
<class'str'>
>>> type(1)
<class'int'>
>>>
```

Implicitly means?

We can ask Python what type something is by using the type() function

Several types of numbers

- Numbers have two main types
- - Integers are whole numbers:

  -14, -2, 0, 1, 100, 401233
- - Floating Point Numbers

  have decimal parts: -2.5 , 0.0, 98.6, 14.0
- There are other number types - they are variations on float and integer

```
>>> xx = 1
>>> type (xx)
<class 'int'>
>>> temp = 98.6
>>> type(temp)
<class'float'>
>>> type(1)
<class 'int'>
>>> type(1.0)
<class'float'>
>>>
```

Type Conversions

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

```
>>> print(float(99) + 100)
199.0
>>> i = 42
>>> type(i)
<class'int'>
>>> f = float(i)
>>> print(f)
42.0
>>> type(f)
<class'float'>
>>>
```

Integer Division

- Integer division produces a floating point result
- This was different in Python 2.x
- To ignore the part after the decimal point (including itself), you should use '//' insead of '/'

```
>>> print(10 / 2)
5.0
>>> print(9 / 2)
4.5
>>> print(99 / 100)
0.99
>>> print(10.0 / 2.0)
5.0
>>> print(99.0 / 100.0)
0.99
>>> print(10 // 4)
2
```

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: Can't convert 'int' object to str implicitly
>>> 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() with base 10: 'x'
```

User Input: Recap

ES112Test

Run

Search

Files

instruction.md

main.py

Packager files

poetry.lock

pyproject.toml

Tools

CPU RAM Storage

Help

main.py × instruction.md ×

instruction.md

```md
# Instructions
Write a program that takes the age of an individual and determines whether they are eligible to cast their vote.

## Input
1. Age

## Output
1. The output of the program should be YES/NO

## Constraints
1. Age should be positive

## Sample examples
Sample Input 1:
```
31
```
Sample Output 1:
```
YES
```

Sample Input 2:
```
15
```
Sample Output 2:
```
NO
```
```

Ln 30, Col 4 History

Input constraints
(age>0)

semantic constraints
(age >=18)

Program

```
age=int(input())
if age >= 18:
    print("YES")
else:
    print('NO')
```

Indentation (spaces, tabs) and Colon ':' MUST be respected syntactically

We can instruct Python to pause and read data from the user using the **input()** function

The input() function returns a string which needs a type conversion to int

Comments in Python

- Anything after a **#** is ignored by Python
- Why comment?
  - Describe what is going to happen in a sequence of code
  - Document who wrote the code or other ancillary information
  - Turn off a line of code - perhaps temporarily

```
# basic eligibility testing
age=int(input())
if age >= 18:
   print("YES") # eligible to cast your vote
else:
   print('NO') # minors are not allowed
```

Conditional Execution

Conditional Steps: Recap

Program

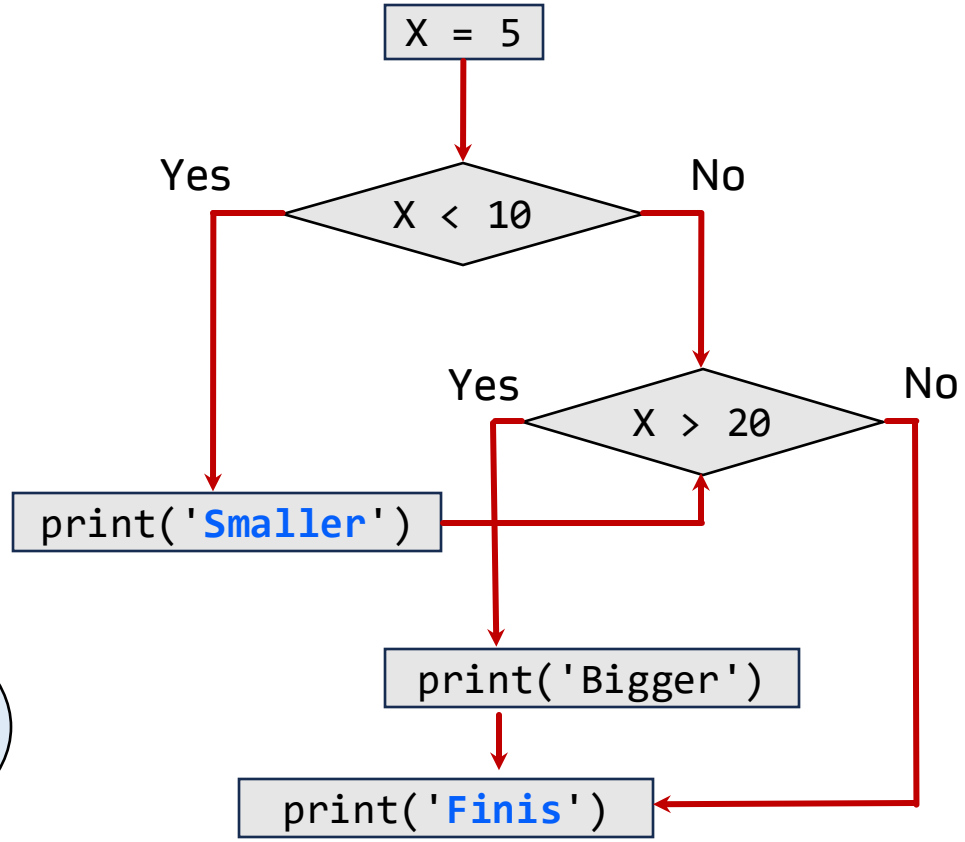
```
x = 5
if x < 10:
    print('Smaller')
if x > 20:
    print('Bigger')

print('Finis')
```

Indentation (spaces, tabs) MUST be consistent.

Colon ':' is a MUST part of the syntax.

Control flow



Output

```
Smaller
Finis
```

When a program is running, **the execution path is not unique**. The non-uniqueness comes from **decision making** on which path to take!

Repeated Steps: Recap

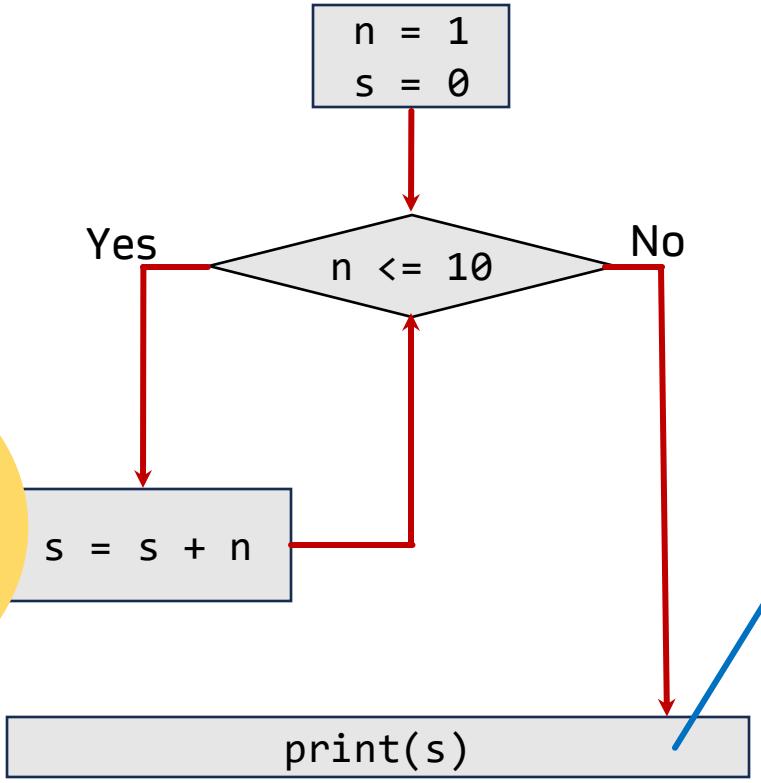
Program

```
n = 1
s = 0
while n <= 10:
    s = s + n
print(s)
```

What will be the output of the program?

Indentation (spaces, tabs) and Colon ':' MUST be respected syntactically

Control flow



Output

?

When a program is running, **the execution of some instruction can be repeated**. This called looping. Here the **iteration variable** is **n**

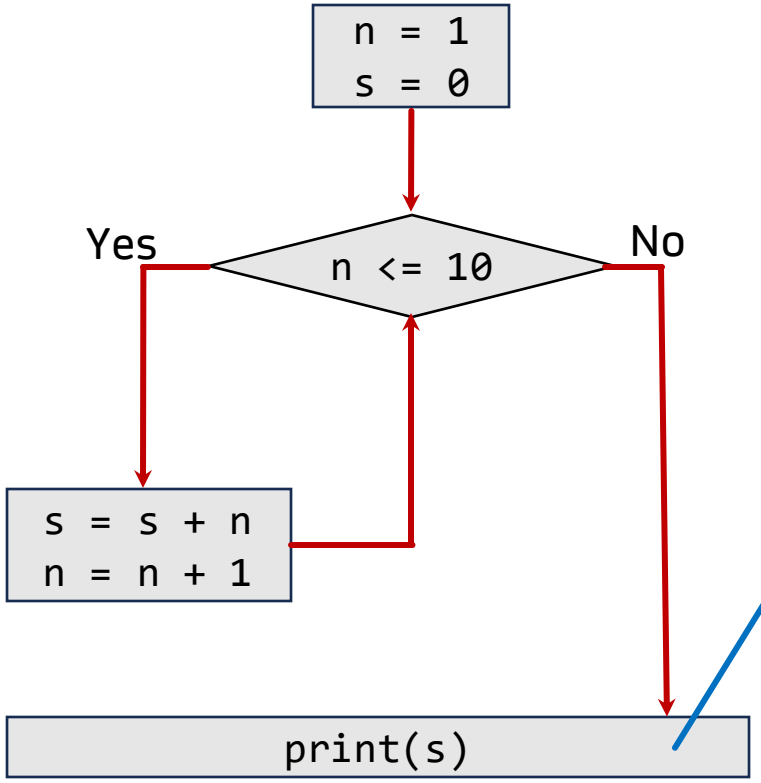
Repeated Steps: Recap

Program

```
n = 1
s = 0
while n <= 10:
    s = s + n
    n = n + 1
print(s)
```

Indentation (spaces, tabs) and Colon ':' MUST be respected syntactically

Control flow



Output

```
55
```

When a program is running, **the execution of some instruction can be repeated**. This called looping. Here the **iteration variable** is **n**

```
i = 0
while i < 6:
  print(i)
  i += 1
```

```
i = 0
for x in
range (0, 6):
  print(x)
```

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

Operator	Operation
<	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

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

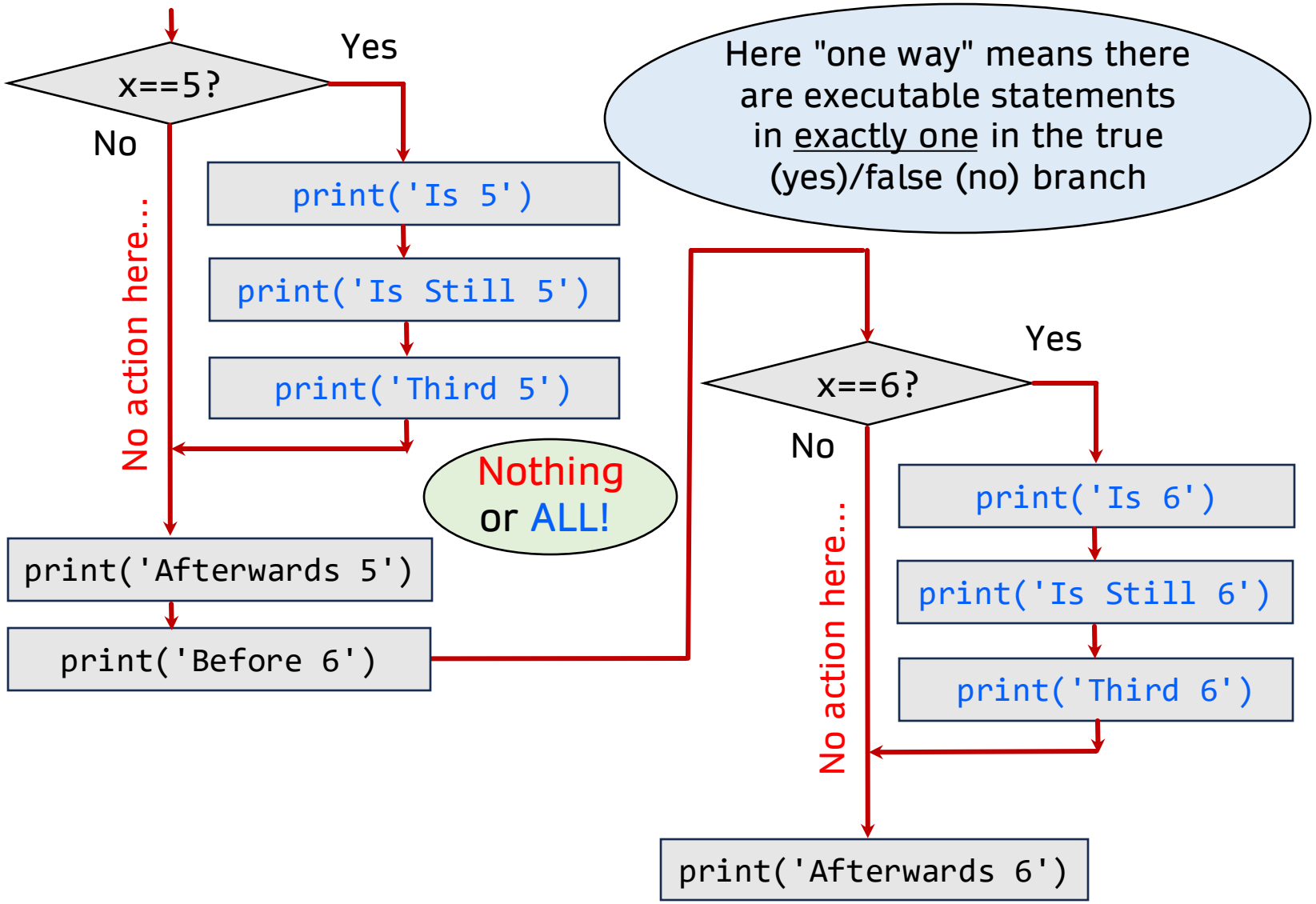
Less than 6

Less than or Equals 5

Not equal 6

One-Way 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')
```



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

Indentation (spaces, tabs) and Colon ':' MUST be respected syntactically

Warning: Turn Off Tabs!

- **Replit** automatically uses spaces for files with "`.py`" extension (`main.py`)
- 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...

Indentation (spaces, tabs)
and Colon ':' MUST be
respected syntactically

Indentation Level = Column Level

Indentation (spaces, tabs) and Colon ':'
MUST be respected syntactically.

increase / maintain after if or for

decrease to indicate end of block

Column#

1 2 3 4

Row#

```
1 x = 5
2 if x > 2:
3     print('Bigger than 2')
4     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')
```

Begin/End blocks

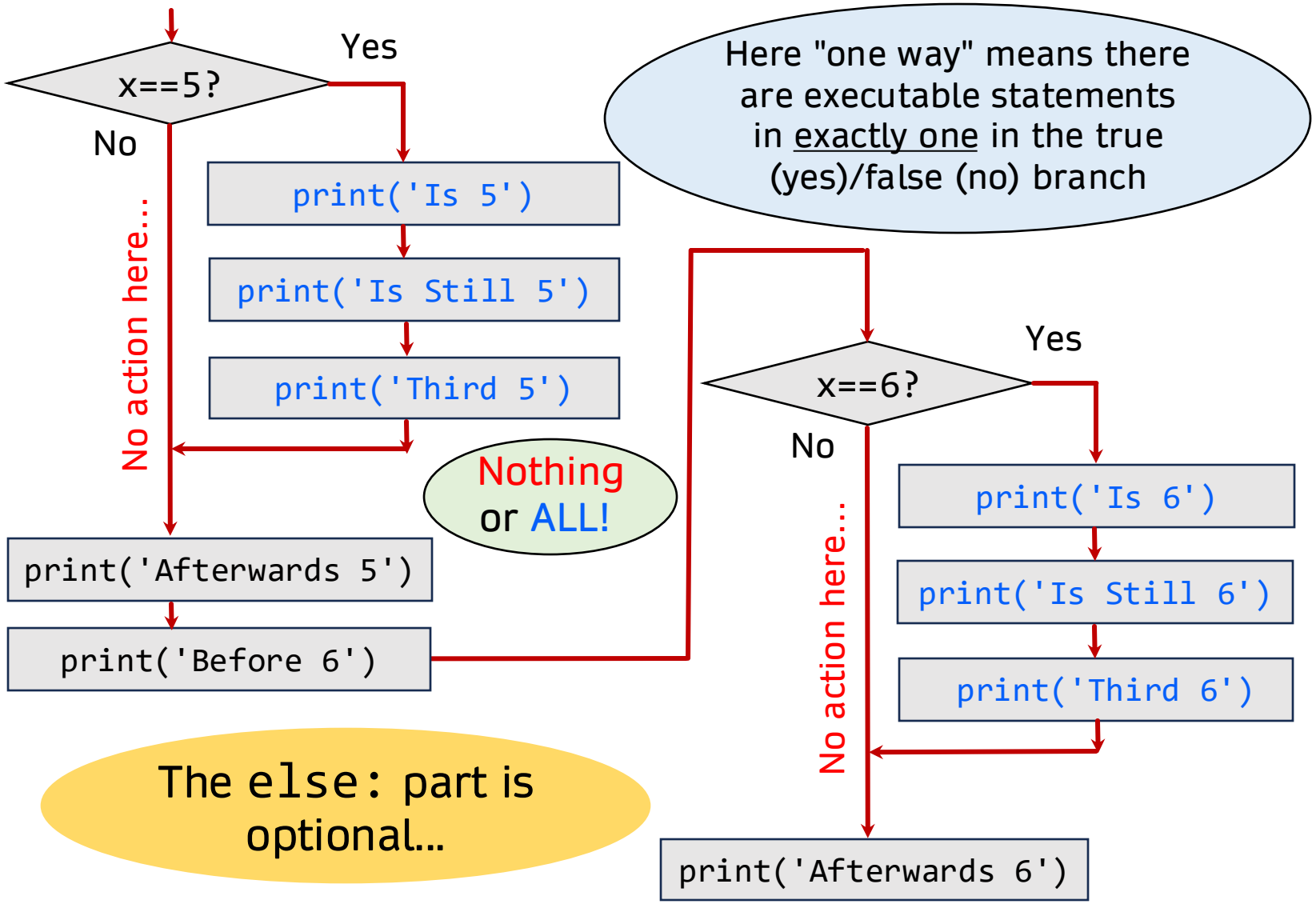
```
x = 5
if x > 2:
  print('Bigger than 2')
  print('Still bigger')
print('Done with 2')
```

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

Blocks are marked in **red**/**blue**. Appropriate indentation at the programmers' view helps in achieving this explicitly.

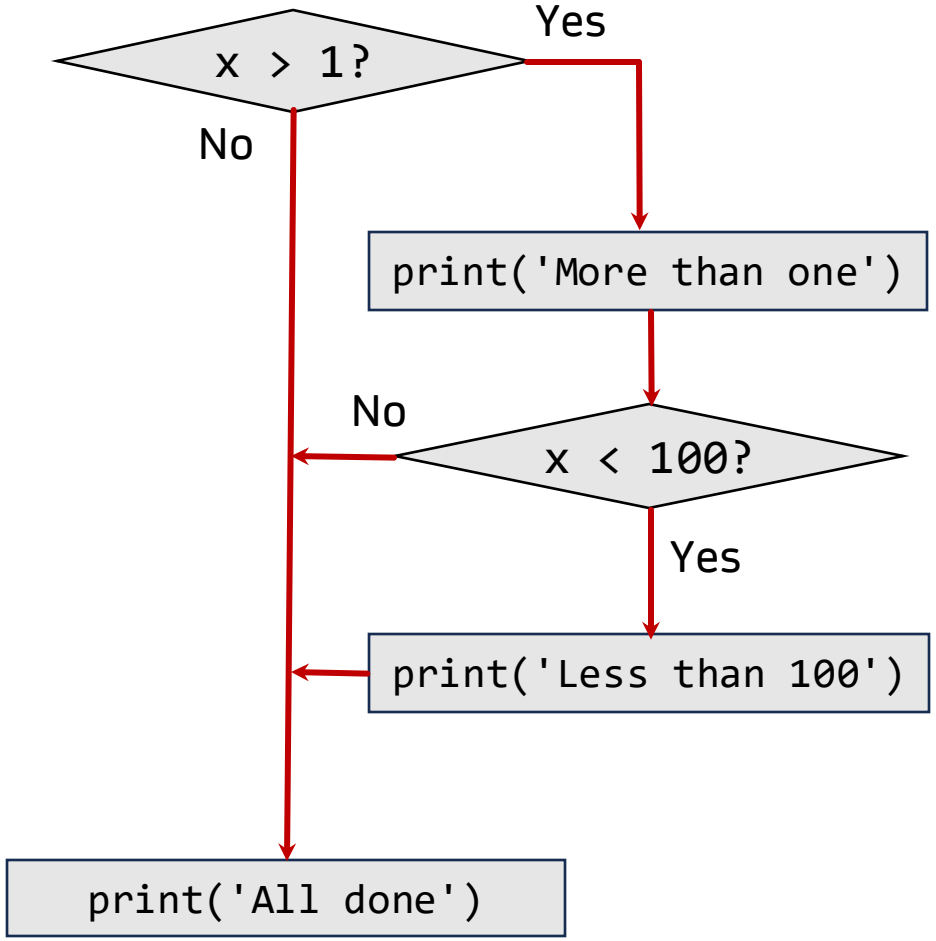
Decision Making: One-Way

```
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')
```



Decision Making: Nested ("one within other")

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



Here the inner if statement is nested within the outer if statement

Note how the corresponding blocks become nested too because of the if nesting

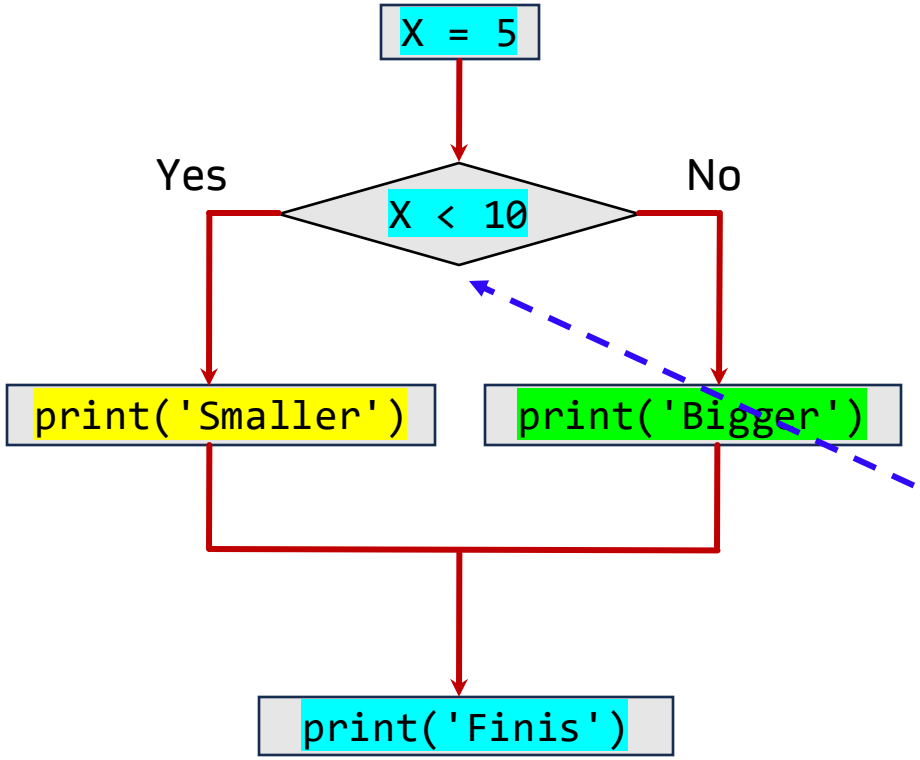
Decision Making: Two Way

Program

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

print('Finis')
```

Control flow



Output

```
Smaller
Finis
```

- 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

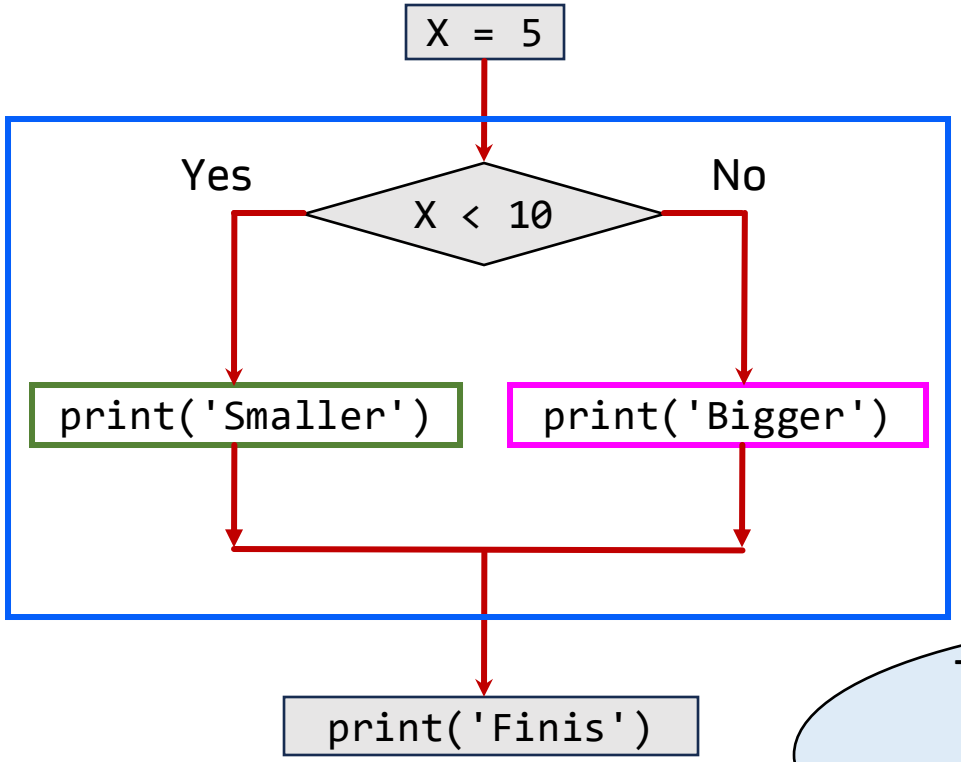
Visualizing Block Composition

Program

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

print('Finis')
```

Control flow



Output

```
Smaller
Finis
```

This can reduce the cognitive load on the programmer (software developer) in a large-scale setting

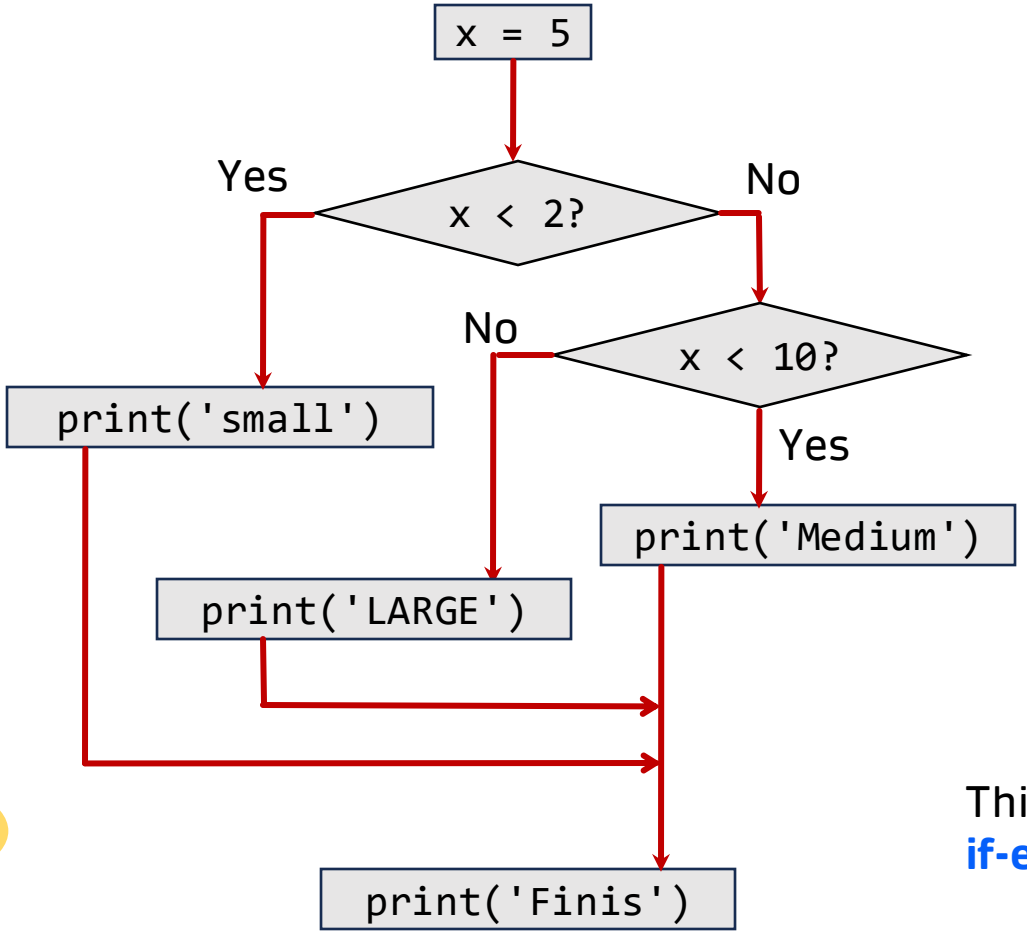
Decision Making: Multi Way

Program

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

The `else:` part is optional...

Control flow



Output

```
Medium
Finis
```

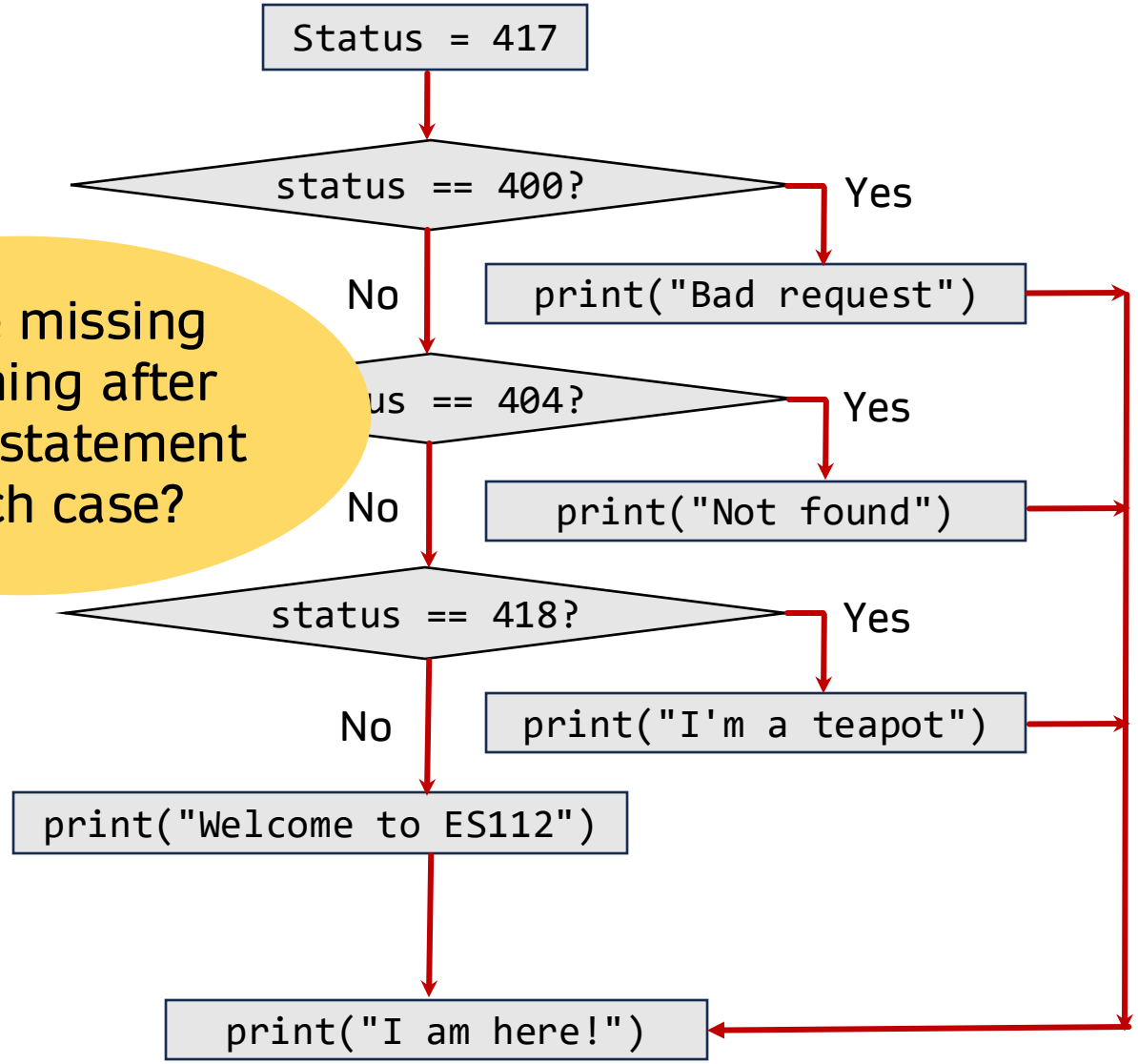
How will the control flow look like with `if-elif-if-elif...`(100 times)?

This is called **if-else ladder**

Decision Making: match-case

```
status = 417
match status:
  case 400:
    print("Bad request")
  case 404:
    print("Not found")
  case 418:
    print("I'm a teapot")
  case _:
    print("Welcome to ES112")
print("I am here!")
```

Are we missing something after the last statement in each case?



The **underscore** '_' is a wildcard.
If nothing else matches, this is the **last resort (default)** action to be performed

Decision Making: match-case

```
status = 417
match status:
   case 400:
      print("Bad request")
      break
   case 404:
      print("Not found")
      break
   case 418:
      print("I'm a teapot")
      break
   case _:
      print("Welcome to ES112")
print("I am here!")
```

In Python, (unlike C/C++/Java) we do not need a break because fall-through does not happen

On execution

```
   File "main.py", line 5
      break
      ^^^^^
SyntaxError: 'break' outside loop
```

Exception Handling: try-except

- 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

Can you give some example of such exceptional situations?

Exception Handling: try-except

```
a = 10
b = 0
c = 0 #initialization
c = a/b
print(c)
```

On execution

```
Traceback (most recent call last):    File "main.py", line 4, in
    c = a/b
ZeroDivisionError: division by zero
```

The program crashes at this point (fault site), anything after is not executed at all

```
a = 10
b = 0
c = 0 #initialization
try:
  c = a/b
except:
  print("Exception!")
print(c)
```

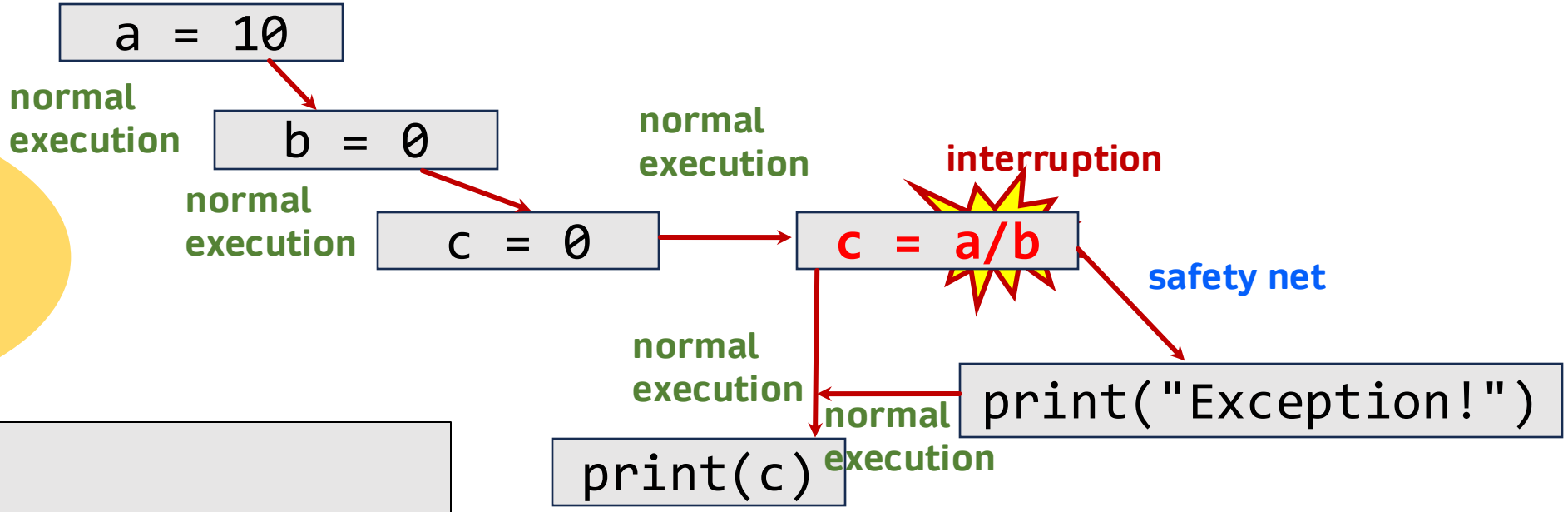
On execution

```
Exception!
0
```

The program **gracefully handles** the **situation**, executes `print(c)`, and exits

Exception Handling: try-except

Where are the decision boxes in the flowchart?




```
a = 10
b = 0
c = 0 #initialization
try:
  c = a/b
except:
  print("Exception!")
print(c)
```

On execution

```
Exception!
0
```

The program **gracefully handles** the **situation**, executes print(c), and exits...

Acknowledgements / Contributions

Initial Development: Charles Severance, University of Michigan School of Information

Contributors 2024 - Yogesh K. Meena and Shouvick Mondal, IIT Gandhinagar