



HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Chapter 2: Control flow

Contents

- Branching and conditionals
- Iteration and loops

Branching and conditionals

Comparison operators on int, float, string

- `i` and `j` are variable names
- comparisons below evaluate to a Boolean

`i > j`

`i >= j`

`i < j`

`i <= j`

`i == j` → **equality** test, `True` if `i` is the same as `j`

`i != j` → **inequality** test, `True` if `i` not the same as `j`

Logic operators on bools

- a and b are variable names (with Boolean values)

not a \rightarrow True if a is False
 False if a is True

a and b \rightarrow True if both are True

a or b \rightarrow True if either or both are True

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Comparison example

```
pset_time = 15  
sleep_time = 8  
print(sleep_time > pset_time)  
derive = True  
drink = False  
both = drink and derive  
print(both)
```

CONTROL FLOW - BRANCHING

```
if <condition>:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

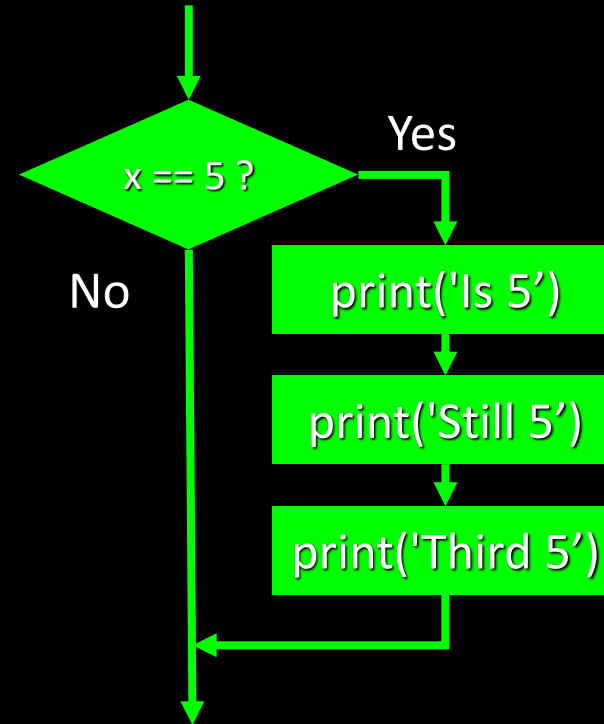
```
if <condition>:  
    <expression>  
    <expression>  
    ...  
elif <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

- <condition> has a value True or False
- evaluate expressions in that block if <condition> is True

The IF Statement

```
x = 5
```

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



Indentation Rules

- 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 to 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 w.r.t. indentation

Indentation Rules

increase / maintain after if or for

decrease to indicate end of block

blank lines and comment lines ignored

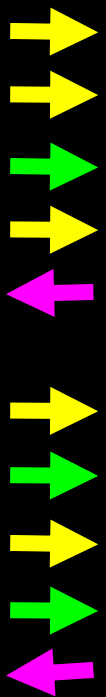


Diagram illustrating indentation rules for the first code block. Arrows indicate the indentation level for each line: yellow for 'x = 5', 'if x > 2:', and 'for i in range(5):'; green for the first two lines of the 'if' block and the first two lines of the 'for' block; and magenta for the final line of each block ('print('Done with 2\')' and 'print('Done with i', i)').

```
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)
```

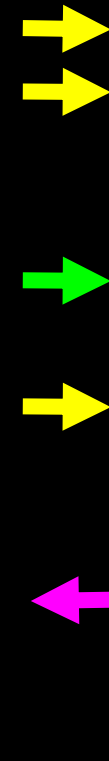


Diagram illustrating indentation rules for the second code block. Arrows indicate the indentation level for each line: yellow for 'x = 5', 'if x > 2:', and the first two lines of the 'for' block; green for the first two lines of the 'if' block and the first two lines of the 'for' block; and magenta for the final line of each block ('print('Done with 2\')' and 'print('Done with i', i)').

```
x = 5
if x > 2 :
    # comments

    print('Bigger than 2')
    # don't matter
    print('Still bigger')
    # but can confuse you

print('Done with 2')
# if you don't line
# them up
```

INDENTATION

- matters in Python
- how you denote blocks of code

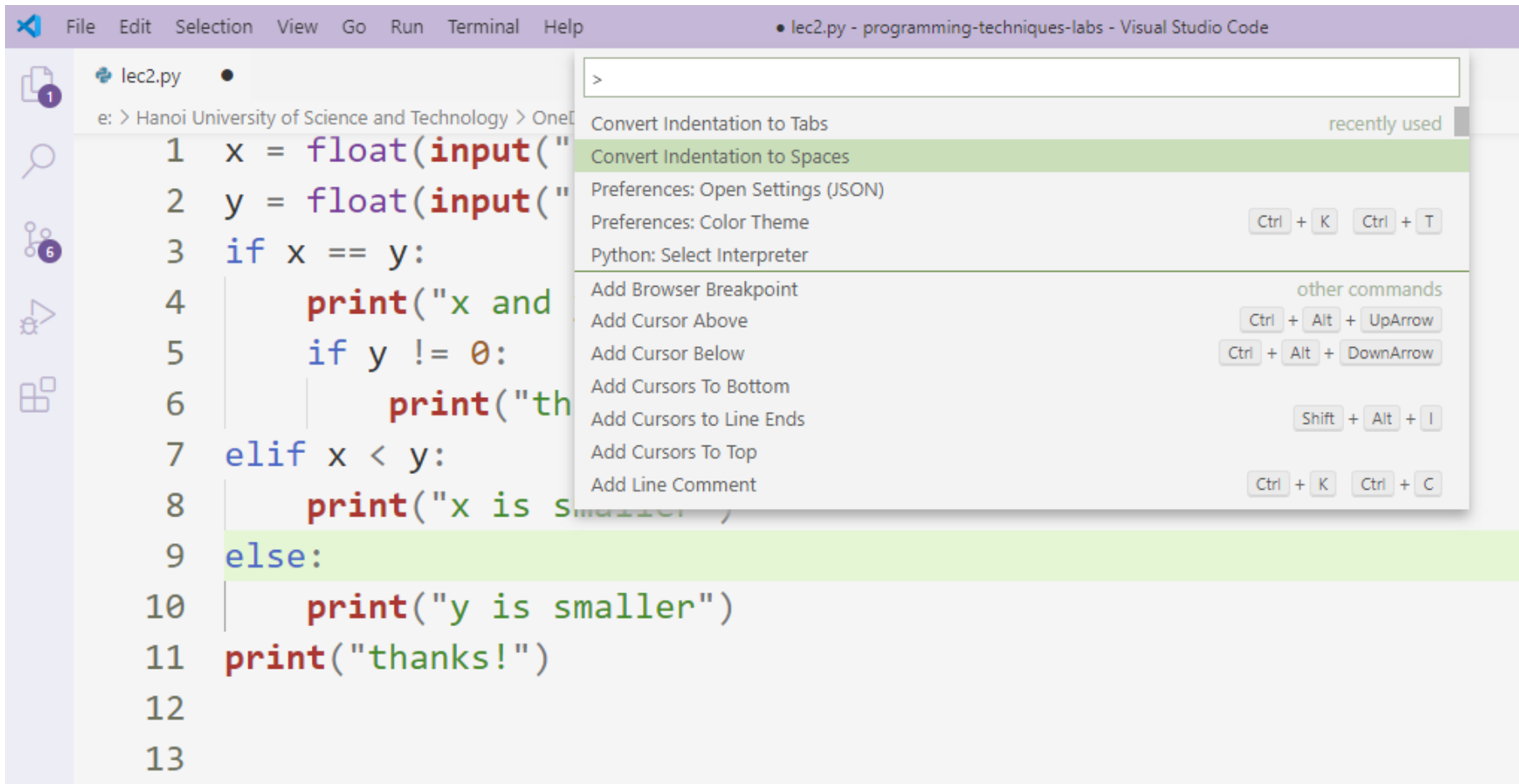
```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))
if x == y:
    print("x and y are equal")
    if y != 0:
        print("therefore, x / y is", x/y)
elif x < y:
    print("x is smaller")
else:
    print("y is smaller")
print("thanks!")
```

Warning: Turn Off Tabs

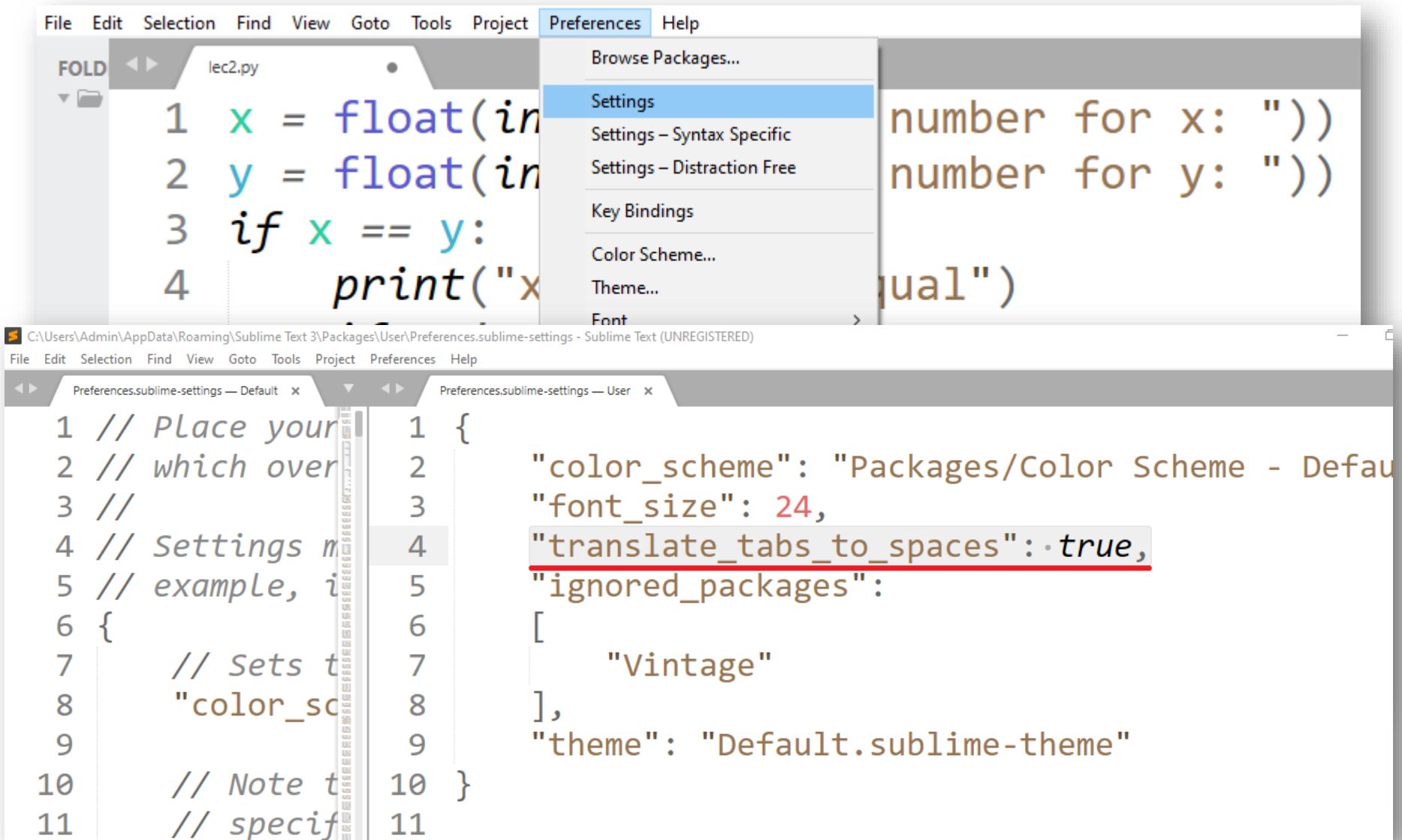
- Python cares a **lot** about how far line is indented. If you mix tabs and spaces, you may get “indentation errors” even if everything looks fine
- Most text editors can turn tabs into spaces - make sure to enable this feature

Turn Off Tabs in Visual Studio Code

- **Ctrl + Shift + P** and choose “Convert Indentation to Spaces”



Turn Off Tabs in Sublimes



The image shows two overlapping windows from the Sublime Text editor. The top window displays the 'Preferences' menu, with 'Settings' highlighted. The background code in this window is a Python snippet:

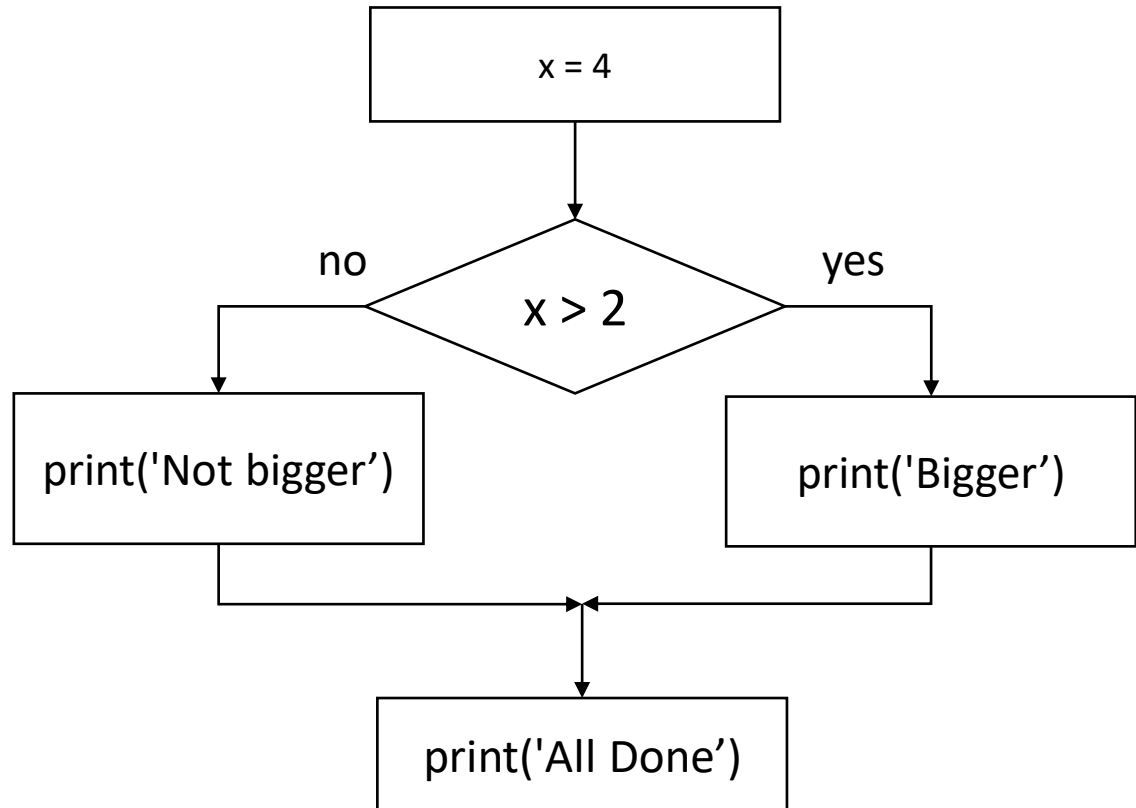
```
1 x = float(input("Enter a number for x: "))
2 y = float(input("Enter a number for y: "))
3 if x == y:
4     print("x and y are equal")
```

The bottom window shows the 'Preferences.sublime-settings' file, split into 'Default' and 'User' panes. The 'User' pane contains a JSON configuration where the line `"translate_tabs_to_spaces": true,` is highlighted with a red underline.

```
1 {
2     "color_scheme": "Packages/Color Scheme - Default",
3     "font_size": 24,
4     "translate_tabs_to_spaces": true,
5     "ignored_packages":
6     [
7         "Vintage"
8     ],
9     "theme": "Default.sublime-theme"
10 }
11
```

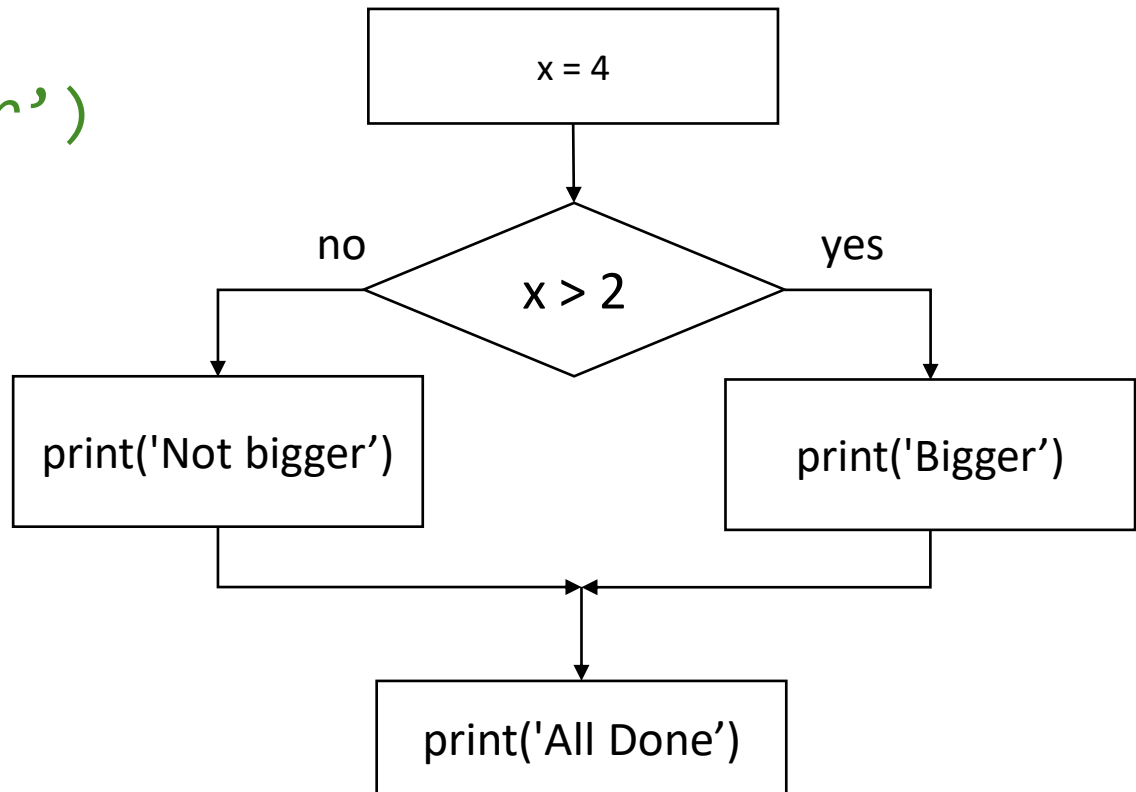
Two Way Decisions

- 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 branch using else :

```
x = 4
if x > 2:
    print('Bigger')
else:
    print('Smaller')
print('All done')
```

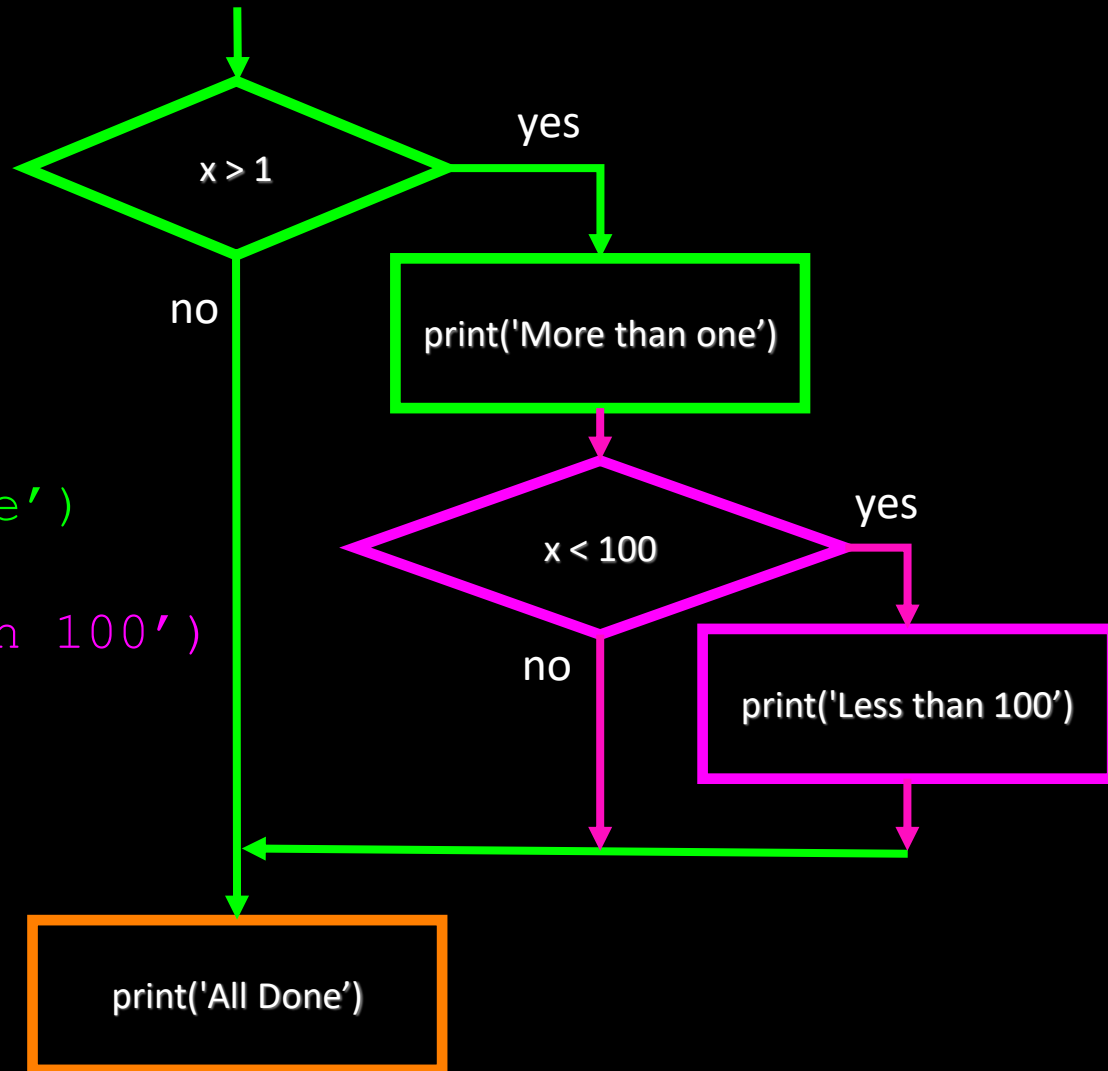


Nested Decisions

x = 42

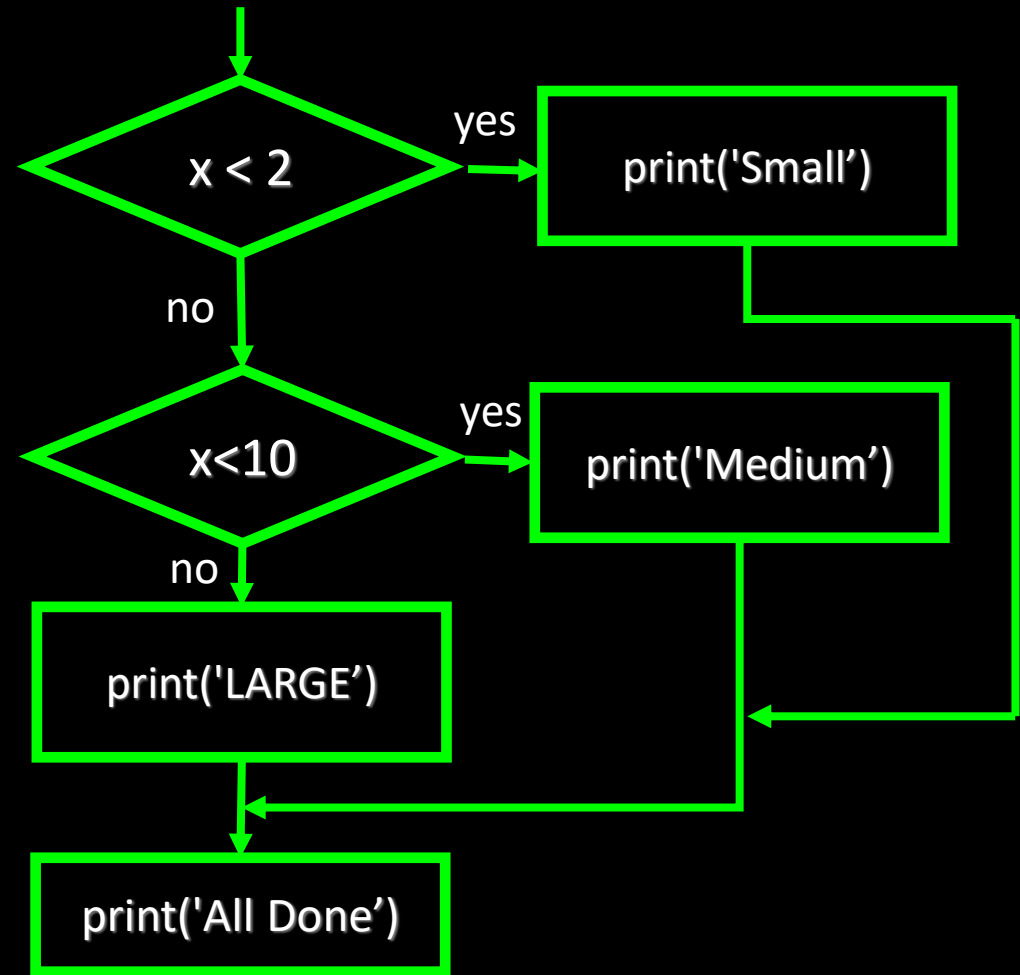
```
if x > 1:  
    print('More than one')  
    if x < 100:  
        print('Less than 100')
```

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



Chained Conditionals

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



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')
```

Multi-way Puzzles

- Which will never print?

```
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')
```

Exercise

Write a pay computation program that gives the employee 1.5 times the hourly rate for hours worked above 40 hours (and regular 1.0 rate for less than 40 hours)

Enter Hours: 45

Enter Rate: 10

Pay: 475.0

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

Exercise

- Write a program that prompts the user to input three stick lengths, converts them to integers, and check whether sticks with the given lengths can form a triangle.

- For example:

Input: 3 8 6

Output: YES

Input: 2 9 4

Output: NO

Exercise

- Return the number of days in the month of the Gregorian calendar.
- The program ask the user to type the number of a month and a year

Exercise

- Write a program to find all the roots of a quadratic equation $ax^2+bx+c=0$

Exercise

- Check whether the year you enter is a leap year or not
- In the Gregorian calendar, A leap year is a year containing one extra day (366 days)
- Most years that are evenly divisible by 4 are leap years
- Years are evenly divisible by 100 are not leap year unless they are evenly divisible by 400

Iteration and loops

CONTROL FLOW: while LOOPS

```
while <condition>:  
    <expression>  
    <expression>  
    ...
```

- <condition> evaluates to a Boolean
- if <condition> is True, do all the steps inside the while code block
- check <condition> again
- repeat until <condition> is False

CONTROL FLOW: while and for LOOPS

- iterate through numbers in a sequence

```
# more complicated with while loop
```

```
n = 0
```

```
while n < 5:
```

```
    print(n)
```

```
    n = n+1
```

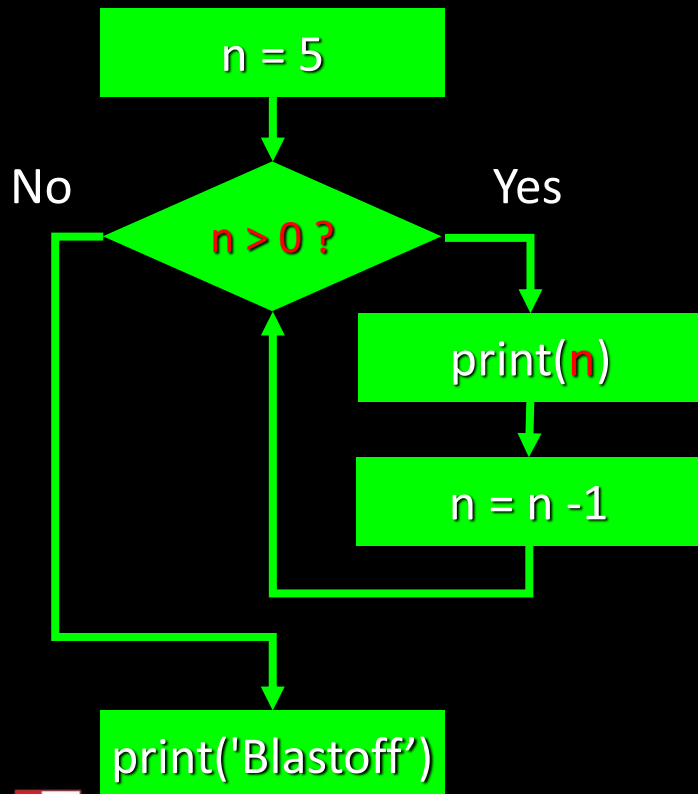
```
# shortcut with for loop
```

```
for n in range(5):
```

```
    print(n)
```

Repeated Steps

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



Program:

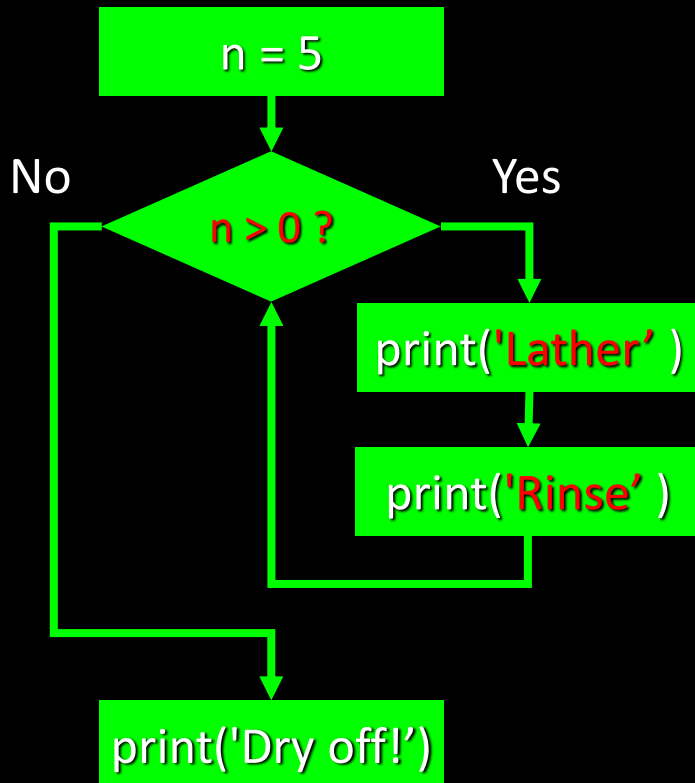
```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```

Output:

```
5
4
3
2
1
Blastoff!
0
```

An Infinite Loop

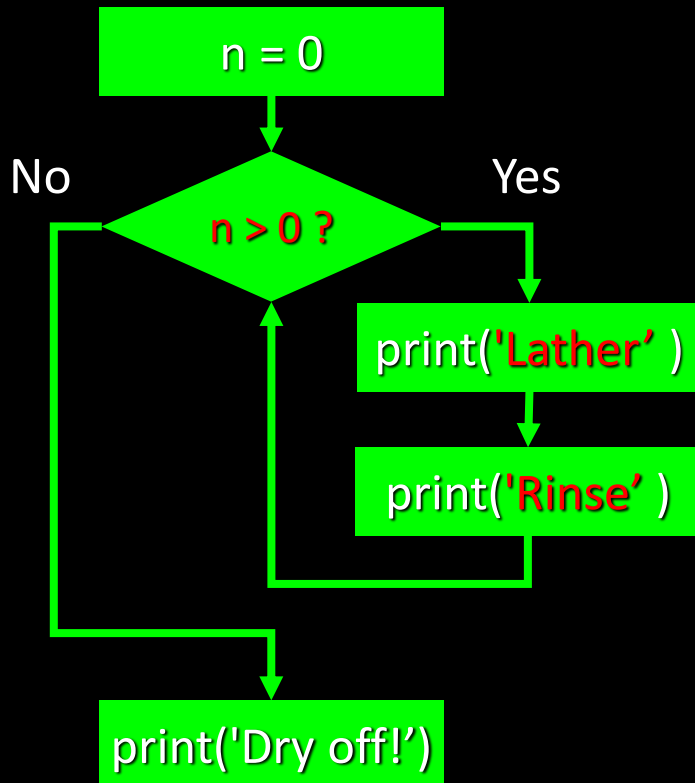
- What is wrong with this loop?



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

Another Loop

- What does this loop do?



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

break STATEMENT

- immediately exits whatever loop it is in
- skips remaining expressions in code block
- exits only innermost loop!

```
while <condition_1>:  
    while <condition_2>:  
        <expression_a>    break  
        <expression_b>  
    <expression_c>
```

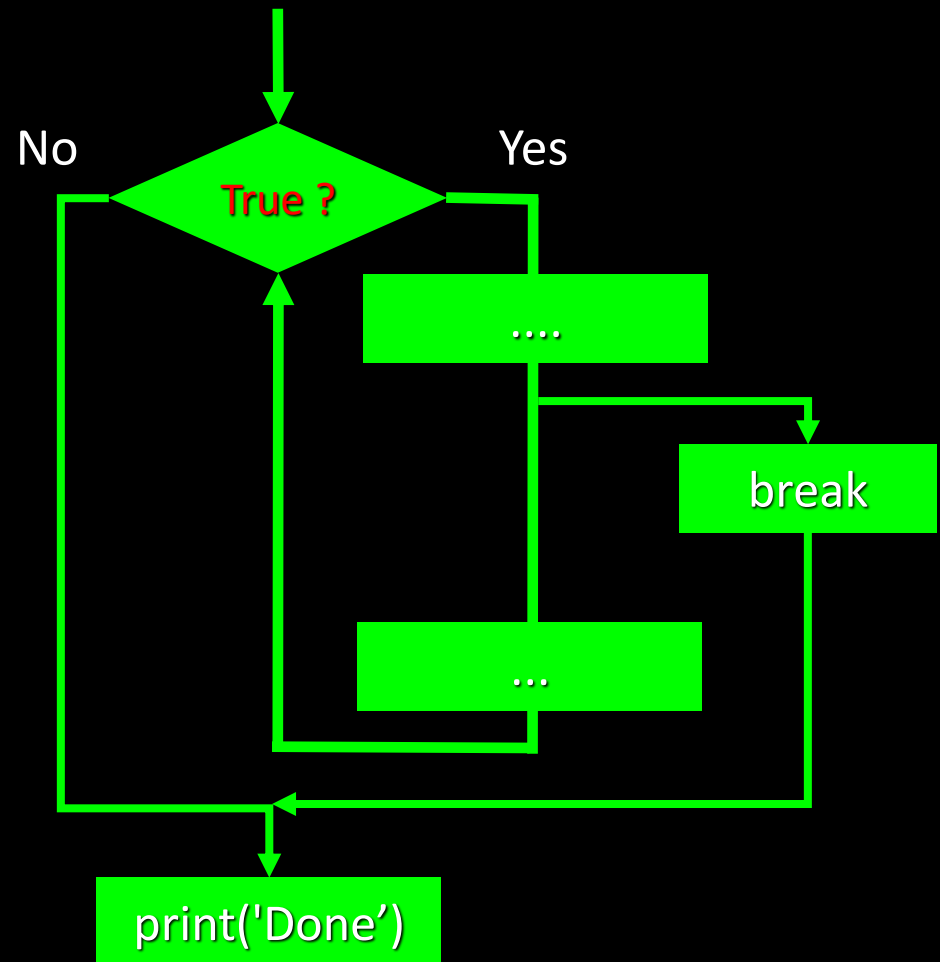

Breaking Out of a Loop

- The **break** statement ends the current loop and jumps to the statement immediately following the loop

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

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

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



Using continue in a loop

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

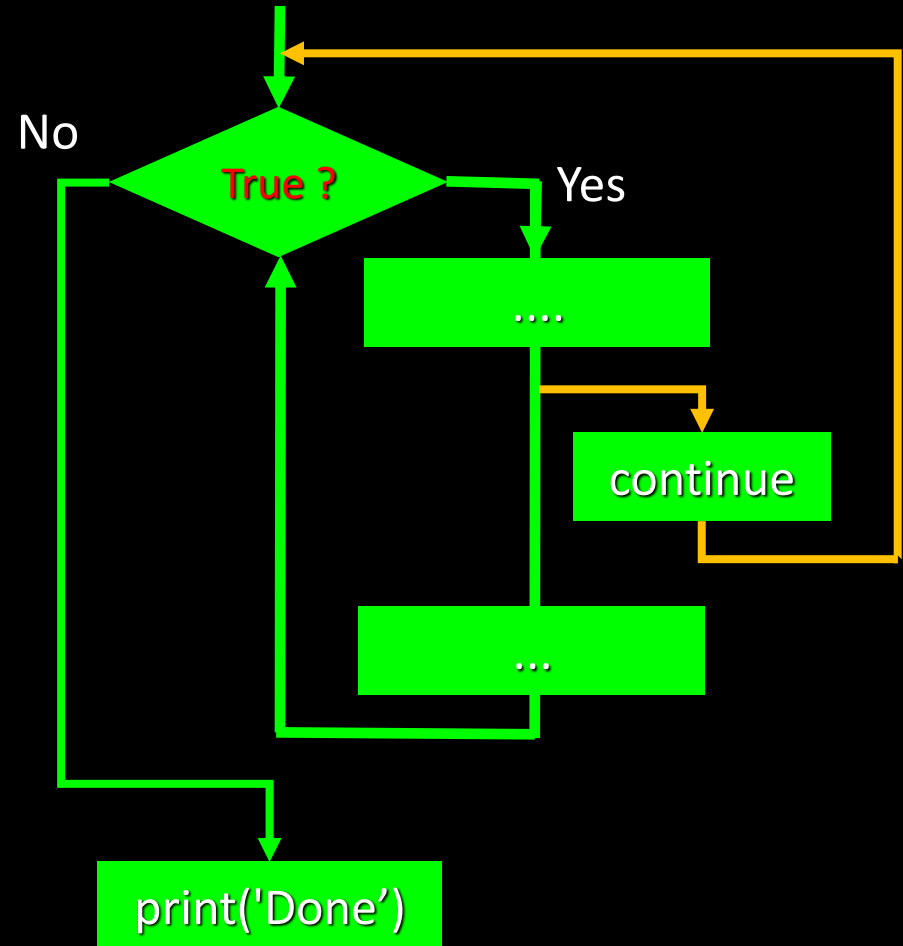
```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')
```

```
> hello there
hello there
> # don't printthis
> printthis!
printthis!
> done
Done!
```

```

while True:
    line = 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

- 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**"

CONTROL FLOW: for LOOPS

```
for <variable> in range(<some_num>):  
    <expression>  
    <expression>  
    ...
```

- each time through the loop, <variable> takes a value
- first time, <variable> starts at the smallest value
- next time, <variable> gets the prev value + 1
- etc.

A Simple Definite Loop

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

5

4

3

2

1

Blastoff!

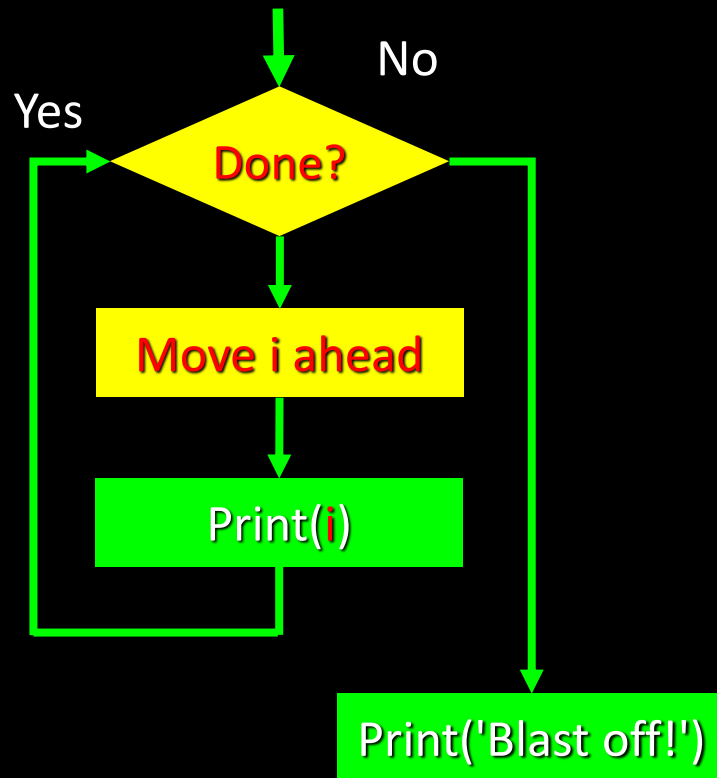
A Simple Definite Loop

```
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

- Definite loops (for loops) have explicit **iteration variables** that change each time through a loop. These **iteration variables** move through the sequence or set.



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

5
4
3
2
1
Blastoff!

The `range(start, stop, step)` function

- `range()` is a built-in function that allows you to create a sequence of numbers in a range
- Very useful in “for” loops which are discussed later in the Iteration chapter
- Takes as an input 1, 2, or 3 arguments. See examples.
- default values are `start = 0` and `step = 1` and optional
- loop until value is `stop - 1` if the step value is `positive` or `stop + 1` if the step value is `negative`

```
x = range(5)
print(list(x))
[0, 1, 2, 3, 4]
```

```
x = range(3, 7)
print(list(x))
[3, 4, 5, 6]
```

```
x = range(10, 1, -2)
print(list(x))
[10, 8, 6, 4, 2]
```

A Simple Definite Loop iterating over a range

```
for i in range(7, 0, -1):  
    print(i)  
print('Blastoff!')
```

7

6

5

4

3

2

1

Blastoff!

Question

- What happens in this program?

```
mysum = 0  
for i in range(5, 11, 2):  
    mysum += i  
    if mysum == 5:  
        break  
    mysum += 1  
print(mysum)
```

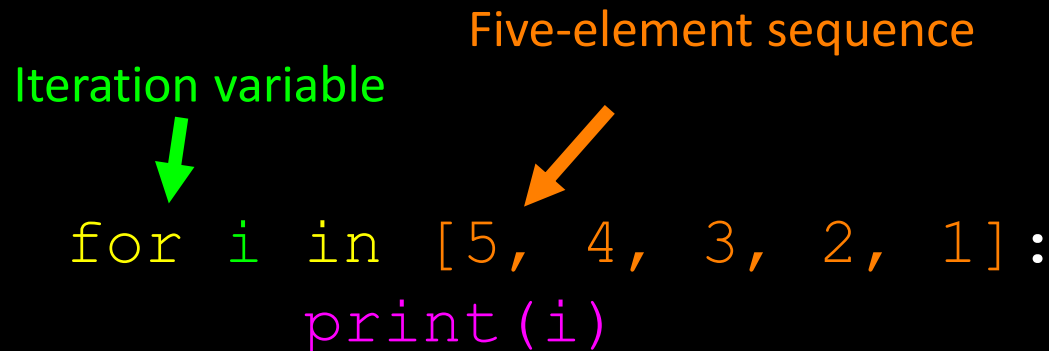
Looking at in

- The **iteration variable** “iterates” through the **sequence**
- 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)
```



Looping through a Set

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

Before

9

41

12

3

74

15

After

Counting in a Loop

- 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.

<code>zork = 0</code>	Before 0
<code>print('Before', zork)</code>	1 9
<code>for thing in [9, 41, 12, 3, 74, 15]:</code>	2 41
<code> zork = zork + 1</code>	3 12
<code> print(zork, thing)</code>	4 3
<code>print('After', zork)</code>	5 74
	6 15
	After 6

Summing in a Loop

- 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.

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

Before 0
9 9
50 41
62 12
65 3
139 74
154 15
After 154

Finding the Average in a Loop

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

count = 0	Before 0 0
sum = 0	1 9 9
print('Before', count, sum)	2 50 41
for value in [9, 41, 12, 3, 74, 15]:	3 62 12
count = count + 1	4 65 3
sum = sum + value	5 139 74
print(count, sum, value)	6 154 15
print('After', count, sum, sum / count)	After 6 154 25

Filtering in a Loop

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

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

Before

Large number 41

Large number 74

After

Search Using a Boolean Variable

- 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.

```
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)
```

Before False
False 9
False 41
False 12
True 3
True 74
True 15
After True

Finding the **smallest** value

- 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**.

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

The "is" and "is not" Operators

- 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

```
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)
```

for vs while loops

for loops

- **know** number of iterations
- can **end early** via `break`
- uses a **counter**
- **can rewrite** a `for` loop using a `while` loop

while loops

- **unbounded** number of iterations
- can **end early** via `break`
- can use a **counter but must initialize** before loop and increment it inside loop
- **may not be able to rewrite** a `while` loop using a `for` loop

Exercise

- Write a program to print odd positive integers less than n in descending order

Exercise

- Write a program to input number n and print its factorial

Exercise

- Calculate sum of harmonic series $1 + 1/2 + \dots + 1/n$

Exercise

- An Armstrong number of 3 digit is an integer that the sum of the cubes of its digits is equal to the number itself. Find Armstrong numbers of 3 digits.

Exercise

- Calculate sum of the first n integers except those divisible by 5

Exercise

- The mathematician Srinivasa Ramanujan found an infinite series that can be used to generate a numerical approximation of $1 / \pi$:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103+26390k)}{(k!)^4 396^{4k}}$$

- Write a program that uses this formula to compute and return an estimate of π . It should use a while loop to compute terms of the summation until the last term is smaller than $1e-15$ (which is Python notation for 10^{-15}). You can check the result by comparing it to `math.pi`.

References

1. [MIT Introduction to Computer Science and Programming in Python](#)
2. Think Python: How to Think Like a Computer Scientist:
<https://greenteapress.com/thinkpython2/html/index.html>



25 YEARS ANNIVERSARY
SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for
your attention!



soict.hust.edu.vn/



fb.com/groups/soict

