SI100B: Introduction to Information Science and Technology

Lecture 2

By Ana Bell of MIT

Announcement

- Tutorial Sessions
  - Every Friday (week 1-9) class 11-12 (starting at 18:55)
  - Classroom: class A @101, class B @301
- Office Hours
  - See BB announcement

	TA	Time Slot	Location
Week 1-5	邵奎翔	周一 19:00-20:00	信息学院1C307
	钟阳	周四 20:00-21:00	信息学院1号楼大厅「阳光很好，BUG很少」标语下
	陈振彬	周四 20:00-21:00	信息学院1号楼大厅「阳光很好，BUG很少」标语下
	林迦勒	周四 16:00-17:00	信息学院1B205
	田丰硕	周三 20:00-21:00	8号楼一楼活动室
	叶昰钊	周三 20:00-21:00	8号楼一楼活动室
	祝元	周二 19:00-20:00	9号楼研讨室
	徐启翰	周四 20:00-21:00	信息学院1号楼大厅「阳光很好，BUG很少」标语下
Week 6-9	TBA		

INPUT/OUTPUT

PRINTING

- Values are shown in the interactive mode, but not to the user

```
>>>3+2
5
```

- `print` shows values to the user

```
print(3+2)
```

- Printing many objects in the same command
  - Separate objects using commas to output them separated by spaces
  - Concatenate strings together using + to print as single object

  -
    ```
    a = "the"
    b = 3
    c = "musketeers"
    print(a, b, c)
    print(a + str(b) + c)
    ```

Every piece being concatenated must be a string

INPUT

- `x = input(s)`
  - Prints the value of the string `s`
  - User types in something and hits enter
  - That value is assigned to the variable `x`
- **Binds that value to a variable**

```
    text = input("Type anything: ")
    print(5*text)
```

SHELL:

```
Type anything:
```

And it waits for characters and Enter to be hit

INPUT

- `x = input(s)`
  - Prints the value of the string `s`
  - User types in something and hits enter
  - That value is assigned to the variable `x`
- **Binds that value to a variable**

“howdy”

```
text = input("Type anything: ")
print(5*text)
```

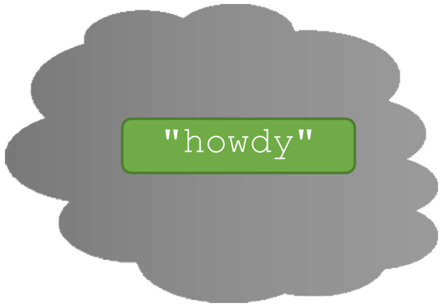
SHELL:

```
Type anything: howdy
```

INPUT

- `x = input(s)`
  - Prints the value of the string `s`
  - User types in something and hits enter
  - That value is assigned to the variable `x`
- **Binds that value to a variable**

```
text = input("Type anything: ")
print(5*text)
```



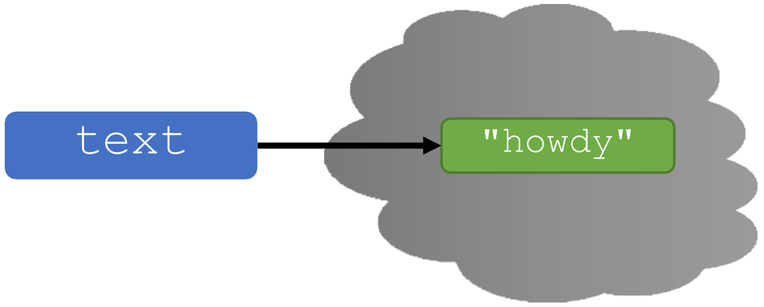
SHELL:

```
Type anything: howdy
```

INPUT

- `x = input(s)`
  - Prints the value of the string `s`
  - User types in something and hits enter
  - That value is assigned to the variable `x`
- **Binds that value to a variable**

```
text = input("Type anything: ")
print(5*text)
```



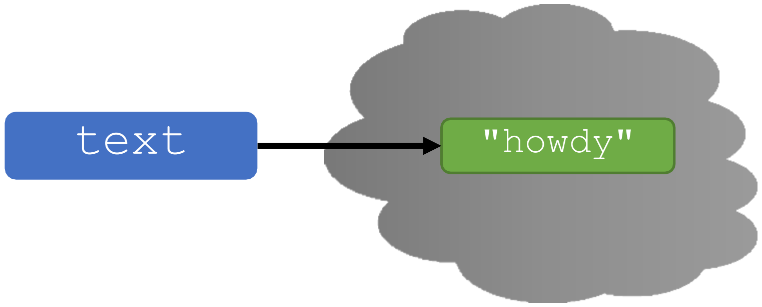
SHELL:

```
Type anything: howdy
```

INPUT

- `x = input(s)`
  - Prints the value of the string s
  - User types in something and hits enter
  - That value is assigned to the variable x
- **Binds that value to a variable**

```
    text = input("Type anything: ")
   print(5*text)
```



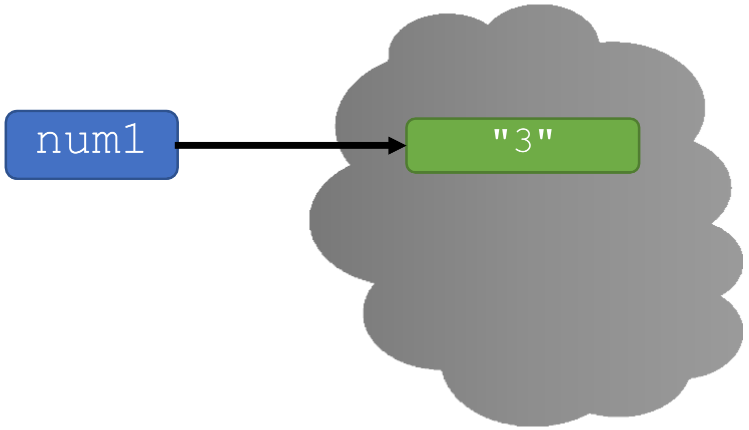
SHELL:

```
Type anything: howdy
howdyhowdyhowdyhowdyhowdy
```

INPUT

- `input` always returns an **str**, must cast if working with numbers

```
num1 = input("Type a number: ")
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```



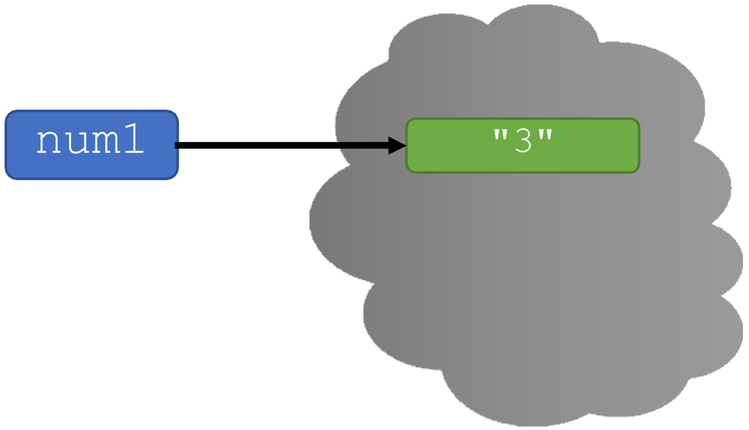
SHELL:

```
Type a number: 3
```

INPUT

- `input` always returns an **str**, must cast if working with numbers

```
num1 = input("Type a number: ")
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```



SHELL:

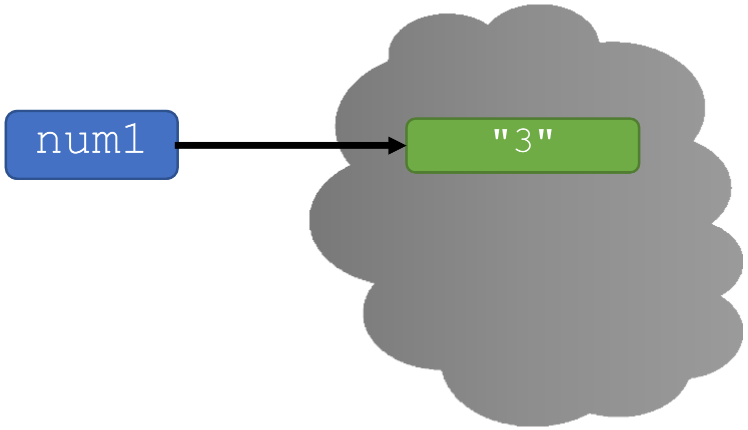
```
Type a number: 3
33333
```

INPUT

- `input` always returns an **str**, must cast if working with numbers

```
num1 = input("Type a number: ")
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```

"3"



SHELL:

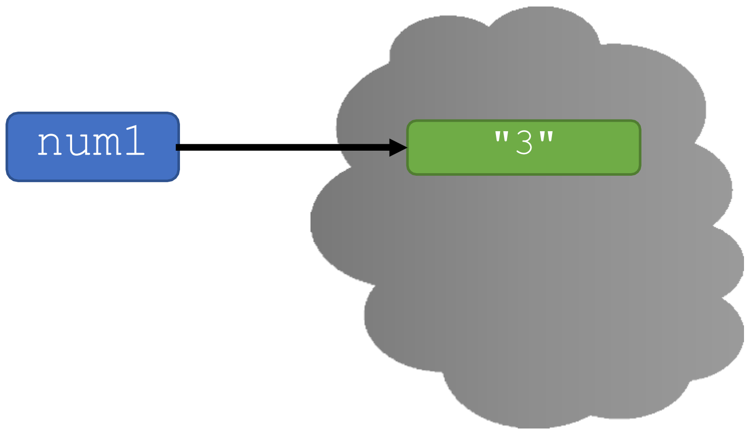
```
Type a number: 3
33333
Type a number: 3
```

INPUT

- `input` always returns an **str**, must cast if working with numbers

```
num1 = input("Type a number: ")
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```

3



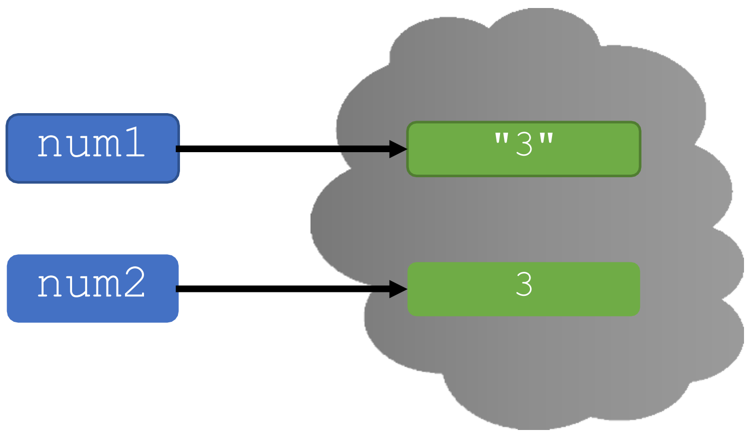
SHELL:

```
Type a number: 3
33333
Type a number: 3
```

INPUT

- `input` always returns an **str**, must cast if working with numbers

```
num1 = input("Type a number: ")
print(5*num1)
num2 = int(input("Type a number: "))
print(5*num2)
```



SHELL:

```
Type a number: 3
33333
Type a number: 3
15
```

YOU TRY IT!

- Write a program that
  - Asks the user for a verb
  - Prints "I can _ better than you" where you replace _ with the verb.
  - Then prints the verb 5 times in a row separated by spaces.
  - For example, if the user enters run, you print:

```
I can run better than you!
run run run run run
```

F-STRINGS

- Available starting with Python 3.6
- Character `f` followed by a **formatted string literal**
  - Anything that can appear in a normal string literal
  - Expressions bracketed by curly braces { }
- Expressions in curly braces evaluated at runtime, automatically converted to strings, and concatenated to the string preceding them

```
num = 3000
fraction = 1/3

print(num*fraction, 'is', fraction*100, '% of', num)
print(num*fraction, 'is', str(fraction*100) + '% of', num)
print(f'{num*fraction} is {fraction*100}% of {num}')
```

Introduces an extra space

expressions

CONDITIONS for BRANCHING

BINDING VARIABLES and VALUES

- In CS, there are two **notions of equal**
  - Assignment and Equality test
- `variable = value`
  - **Change the stored value** of variable to value
  - Nothing for us to solve, computer just does the action
- `some_expression == other_expression`
  - A **test for equality**
  - No binding is happening
  - Expressions are replaced by values and computer just does the comparison
  - Replaces the **entire line** with `True` or `False`

COMPARISON OPERATORS

- `i` and `j` are variable names
  - They can be of type ints, float, strings, etc.
- Comparisons below evaluate to the type **Boolean**
  - The Boolean type only has 2 values: `True` and `False`

```
i > j
i >= j
i < j
i <= j
i == j
i != j
```

With strings, be careful about case sensitivity: 'March' != 'march'

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

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

LOGICAL OPERATORS on bool

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

`not a` → `True` if `a` is `False`
`False` if `a` is `True`

`a and b` → `True` if both are `True`

`a or b` → `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

YOU TRY IT!

- Write a program that
  - Saves a secret number in a variable.
  - Asks the user for a number guess.
  - Prints a bool `False` or `True` depending on whether the guess matches the secret.

WHY bool?

- When we get to flow of control, i.e. branching to different expressions based on values, we need a way of knowing if a condition is true
- E.g., if $\boxed{\text{something is true}}$, $\boxed{\text{do this}}$, otherwise $\boxed{\text{do that}}$

Boolean

Some commands

Some other commands

BRANCHING IN PYTHON

```
if <condition>:
    <code>
    <code>
    ...
<rest of program>
```

- `<condition>` has a value `True` or `False`
- **Indentation matters** in Python!
- Do code within if block if condition is `True`

BRANCHING IN PYTHON

```
if <condition>:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
else:
    <code>
    <code>
    ...
<rest of program>
```

- `<condition>` has a value `True` or `False`
- **Indentation matters** in Python!
- Do code within if block when condition is `True` **or** code within else block when condition is `False`

BRANCHING IN PYTHON

```
if <condition>:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
else:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
<rest of program>
```

- `<condition>` has a value `True` or `False`
- **Indentation matters** in Python!
- Run the **first block** whose corresponding `<condition>` is `True`

BRANCHING IN PYTHON

```
if <condition>:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
else:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
<rest of program>
```

```
if <condition>:
    <code>
    <code>
    ...
elif <condition>:
    <code>
    <code>
    ...
else:
    <code>
    <code>
    ...
<rest of program>
```

- `<condition>` has a value `True` or `False`
- **Indentation matters** in Python!
- Run the **first block** whose corresponding `<condition>` is `True`. The else block runs when no conditions were `True`

BRANCHING EXAMPLE

```
pset_time = ???
sleep_time = ???
if (pset_time + sleep_time) > 24:
    print("impossible!")
elif (pset_time + sleep_time) >= 24:
    print("full schedule!")
else:
    leftover = abs(24-pset_time-sleep_time)
    print(leftover,"h of free time!")
print("end of day")
```

Condition that evaluates to a Boolean

This indented code executed if line above is True

This indented code executed if line above is True and the if condition is False

This else block runs only if previous conditions were all False

YOU TRY IT!

- Fix this buggy code (hint, it has bad indentation)!

```
x = int(input("Enter a number for x: "))
y = int(input("Enter a different number for y: "))
if x == y:
     print(x,"is the same as",y)
print("These are equal!")
```

INDENTATION and NESTED BRANCHING

- Matters in Python
- How you **denote blocks of code**

```
x = float(input("Enter a number for x: "))    5     5     0
y = float(input("Enter a number for y: "))    5     0     0
if x == y:                                    True  False True
    print("x and y are equal")                <-          <-
    if y != 0:                                True        False
        print("therefore, x / y is", x/y)     <-
elif x < y:                                         False
    print("x is smaller")
else:
    print("y is smaller")                           <-
print("thanks!")                              <-    <-    <-
```

YOU TRY IT!

- Write a program that
  - Saves a secret number.
  - Asks the user for a number guess.
  - Prints whether the guess is too low, too high, or the same as the secret.

BIG IDEA

Debug early, debug often.

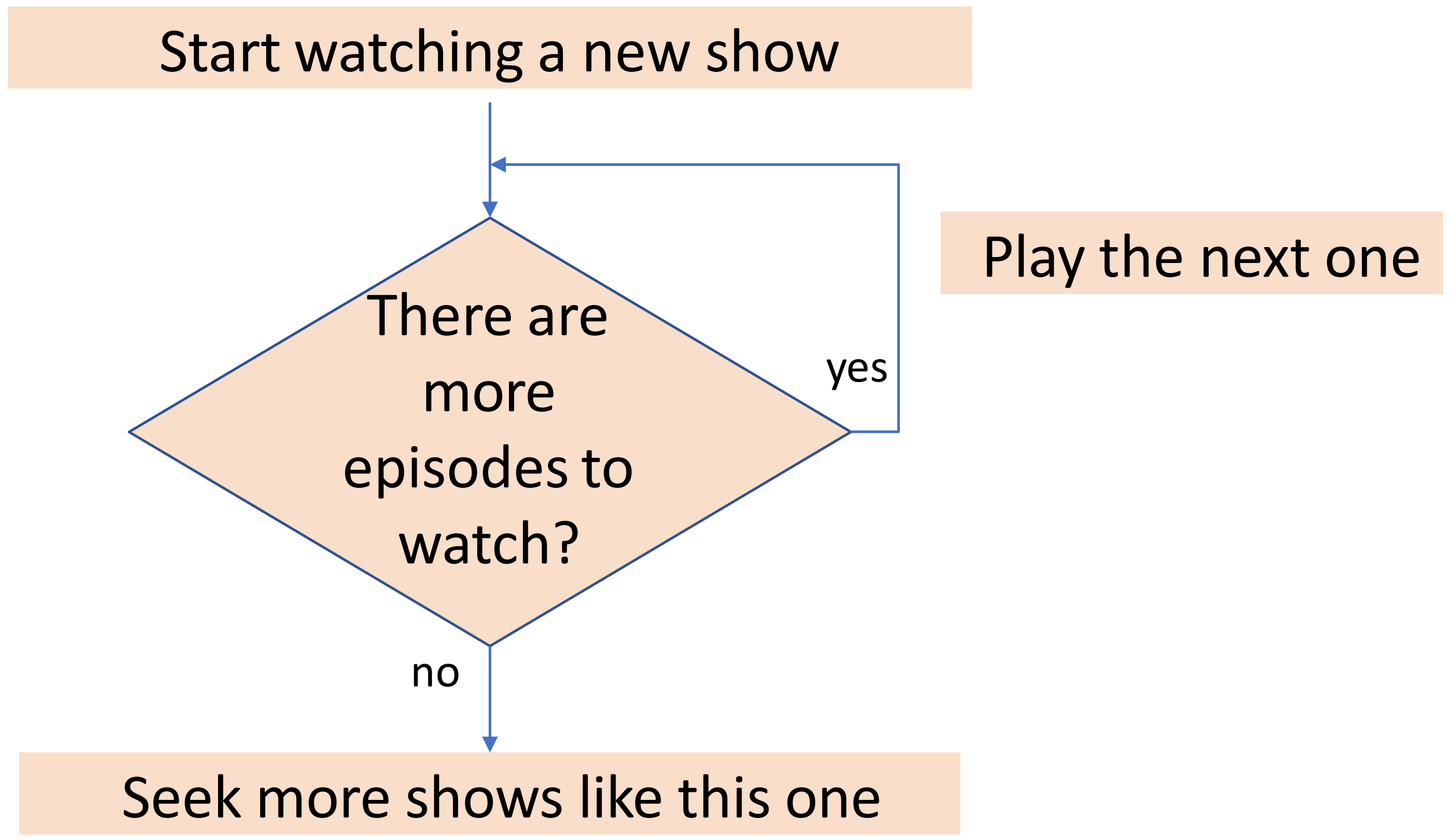
Write a little and test a little.

Don’t write a complete program at once. It introduces too many errors.

Use the Python IDE to step through code when you see something unexpected!

`while` LOOPS

BINGE ALL EPISODES OF ONE SHOW


Start watching a new show
There are more episodes to watch?
yes
Play the next one
no
Seek more shows like this one

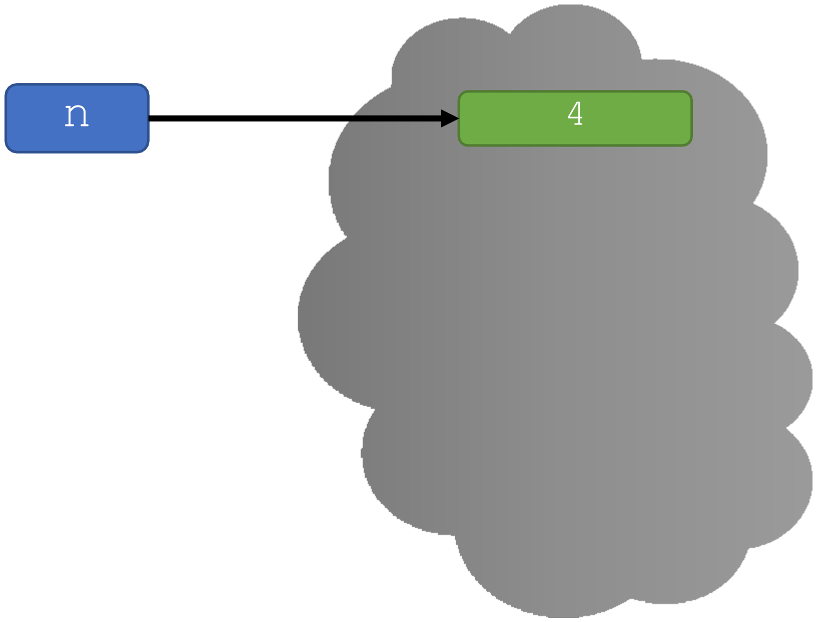
CONTROL FLOW: while LOOPS

```
while <condition>:
        <code>
        <code>
        ...
```

- `<condition>` **evaluates to a Boolean**
- If `<condition>` is `True`, **execute all the steps inside** the while code block
- **Check** `<condition>` again
- Repeat until `<condition>` is `False`
- If `<condition>` is never `False`, then will loop forever!!

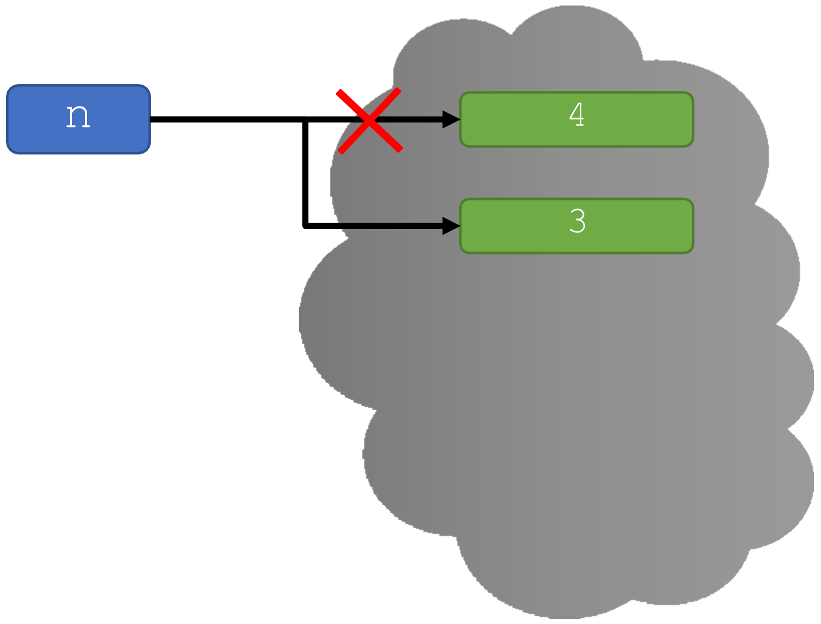
`while` LOOP EXAMPLE

```
n = int(input("Enter a non-negative integer: "))
while n > 0:
    print('x')
    n = n-1
```



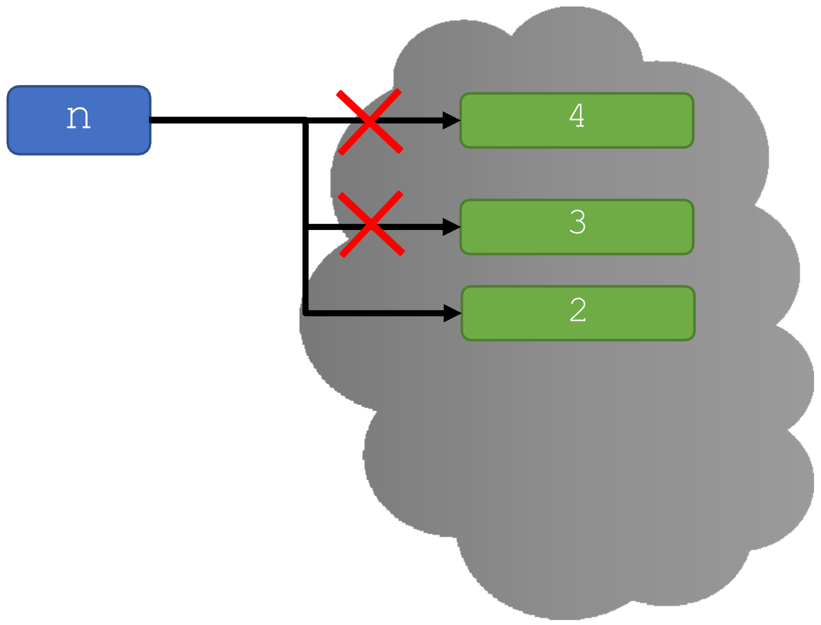
`while` LOOP EXAMPLE

```python
n = int(input("Enter a non-negative integer: "))
while n > 0:
    print('x')
    n = n-1
```



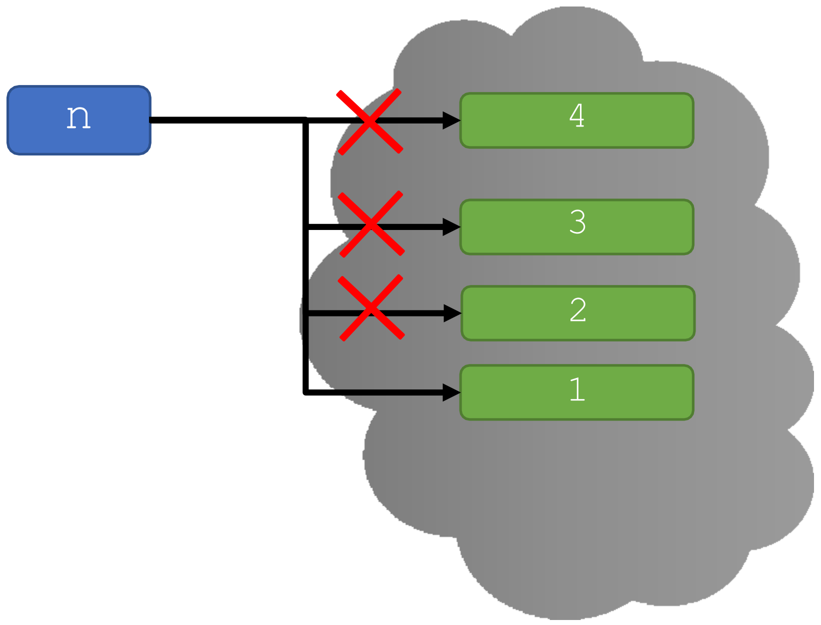
`while` LOOP EXAMPLE

```
n = int(input("Enter a non-negative integer: "))
while n > 0:
    print('x')
    n = n-1
```



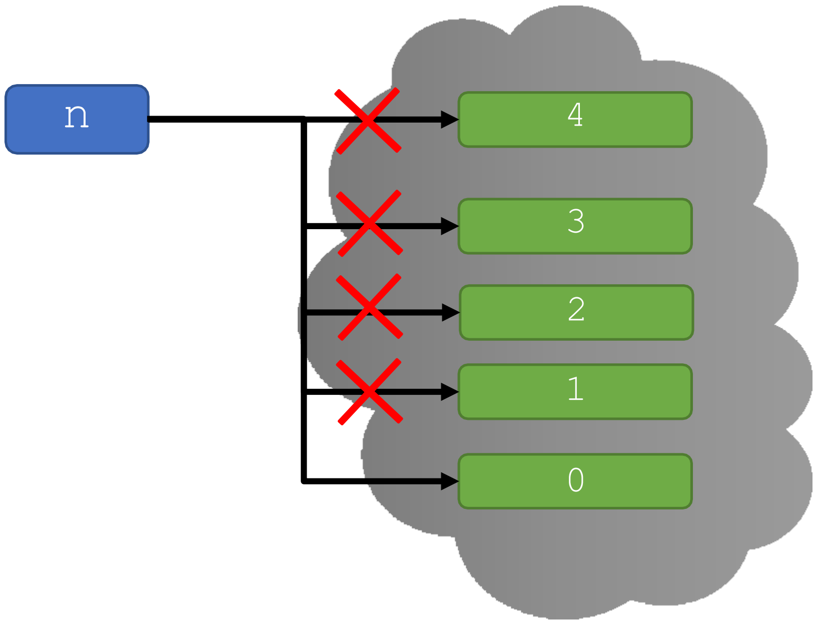
`while` LOOP EXAMPLE

```
n = int(input("Enter a non-negative integer: "))
while n > 0:
    print('x')
    n = n-1
```



`while` LOOP EXAMPLE

```
n = int(input("Enter a non-negative integer: "))
while n > 0:
    print('x')
    n = n-1
```



`while` LOOP EXAMPLE

```
n = int(input("Enter a non-negative integer: "))
while n > 0:
    print('x')
    ~~n = n-1~~
```

What happens without this last line?
Try it!

- To terminate:
  - Hit CTRL-c or CMD-c in the shell
  - Click the red square in the shell

YOU TRY IT!

- Run this code and stop the infinite loop in your IDE

```
while True:
    print("noooooo")
```

BIG IDEA

`while` loops can repeat code inside indefinitely!

Sometimes they need your intervention to end the program.

`while` LOOP EXAMPLE

- Compute 4!
- `i` is our loop variable
- `factorial` keeps track of the product

```
x = 4
i = 1
factorial = 1
while i <= x:
    factorial *= i
    i += 1
print(f'{x} factorial is {factorial}')
```

Set loop variable outside while loop
Initialize the factorial product to 1
Test loop variable in condition
Keep a running product (eq to factorial = factorial*i)
Increment loop variable inside while loop (eq to i = i+1)

`for` LOOPS

CONTROL FLOW: `while` and `for` LOOPS

- Iterate through **numbers in a sequence**

```
# very verbose with while loop
n = 0
while n < 5:
     print(n)
     n = n+1


# shortcut with for loop
for n in range(5):
     print(n)
```

STRUCTURE of for LOOPS

```
for <variable> in <sequence of values>:
    <code>
    ...
```

- **Each time through the loop**, `<variable>` takes a value

- First time, `<variable>` is the **first value in sequence**
- Next time, `<variable>` gets the **second value**
- etc. until `<variable>` runs out of values

A COMMON SEQUENCE of VALUES

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

- **Each time through the loop**, `<variable>` takes a value
- First time, `<variable>` **starts at 0**
- Next time, `<variable>` gets the value **1**
- Then, `<variable>` gets the value **2**
- ...
- etc. until `<variable>` gets **some_num -1**

range

- Generates a **sequence** of ints, following a pattern
- `range(start, stop, step)`
  - `start`: first int generated
  - `stop`: controls last int generated (go up to but not including this int)
  - `step`: used to generate next int in sequence
- A lot like what we saw for **slicing**
- Often omit start and step
  - e.g., `for i in range(4):`
    - `start` defaults to 0
    - `step` defaults to 1
  - e.g., `for i in range(3,5):`
    - `step` defaults to 1

Remember strings? It had a similar syntax, but with colons not commas and square brackets not parentheses.

YOU TRY IT!

- What do these print?
- `for i in range(1,4,1):`

```
    print(i)
```

- `for j in range(1,4,2):`

```
    print(j*2)
```

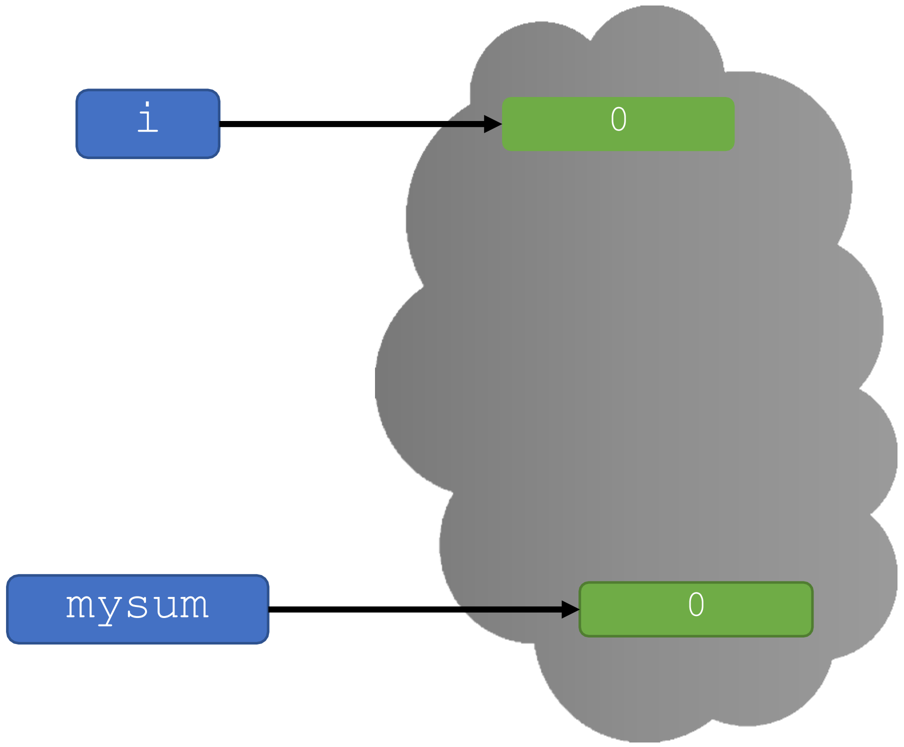
- `for me in range(4,0,-1):`

```
    print("$"*me)
```

RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

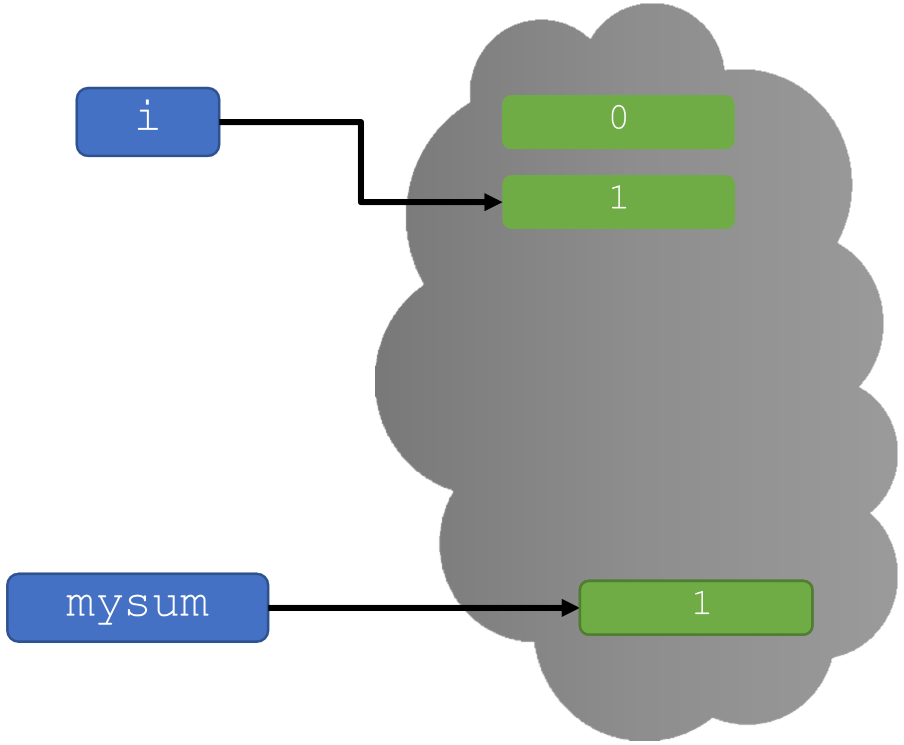
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

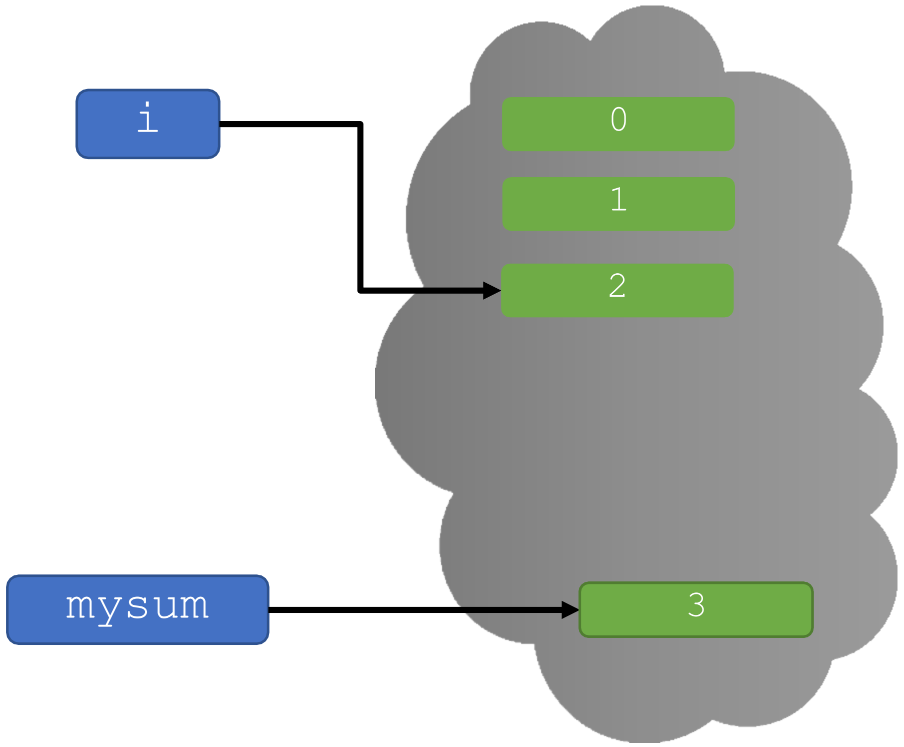
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

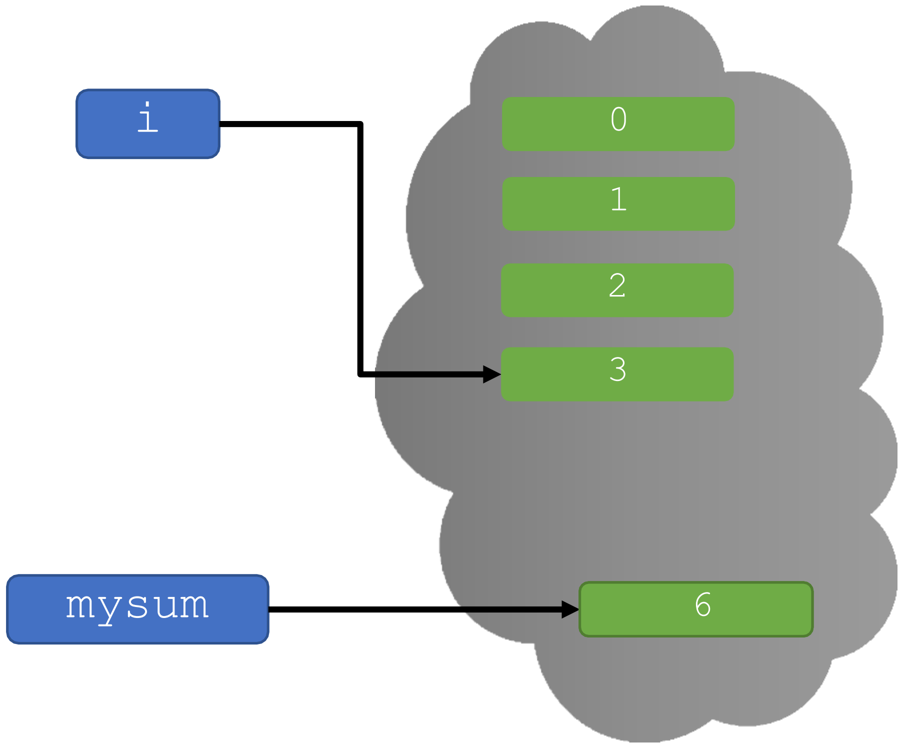
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

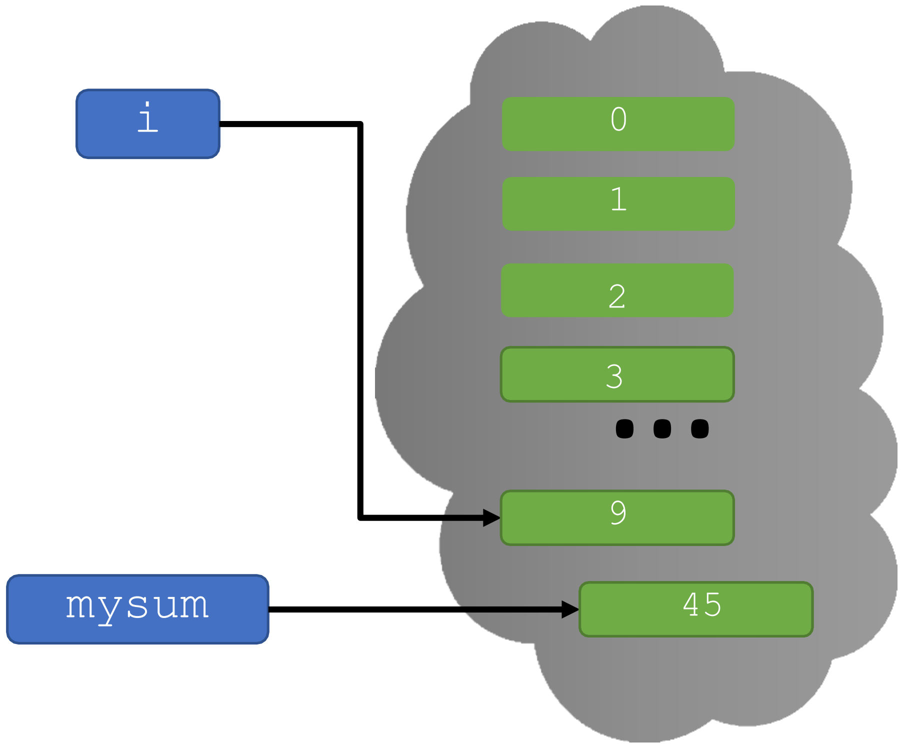
```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



RUNNING SUM

- `mysum` is a variable to store the **running sum**
- `range(10)` makes `i` be 0 then 1 then 2 then … then 9

```
mysum = 0
for i in range(10):
    mysum += i
print(mysum)
```



YOU TRY IT!

- Fix this code to use variables `start` and `end` in the `range`, to get the total sum between and including those values.
- For example, if `start=3` and `end=5` then the sum should be 12.

```
mysum = 0
start = 3
end = 5
for i in range(start, end):
    mysum += i
print(mysum)
```

for LOOPS and range

- Factorial implemented with a `while` loop (seen this already) and a `for` loop

```
x = 4
i = 1
factorial = 1
while i <= x:
    factorial *= i
    i += 1
print(f'{x} factorial is {factorial}’)
```

Uses a while loop

```
x = 4
factorial = 1
for i in range(1, x+1, 1):
    factorial *= i
print(f'{x} factorial is {factorial}')
```

Uses a for loop