# Python for Data Science 2

# Welcome!

**Welcome to the Python for Data Science class**

**About the Program**

**Four Part Series, Four Hours Each**

- **Session 1: Getting Started with Python**
- **Session 2: Applying Python – The Basics**
- **Session 3: Exploring Python Files, Dictionaries, Sets & methods**
- **Session 4: Expanding Python – methods, Error Handling, Importing and OO Classes**

**Quick Logistics: Format, Q&A and Follow-On Materials / Hand-Outs**

**About Me: Ernesto Lee**

# Today’s Agenda: Session 2

**Get started with Python! Learn how to apply logic (if/then and loops) to your Python code. This is the foundation of all future programming concepts!**
**We’ll also explore some advanced data types that are often used in Data Analysis and Data Science.**

- Write a program that accepts multiple positional arguments
- Use if, elif, and else to handle conditional branching with three or more options
- Find and alter items in a list
- Sort and reverse lists
- Format a list into a new string

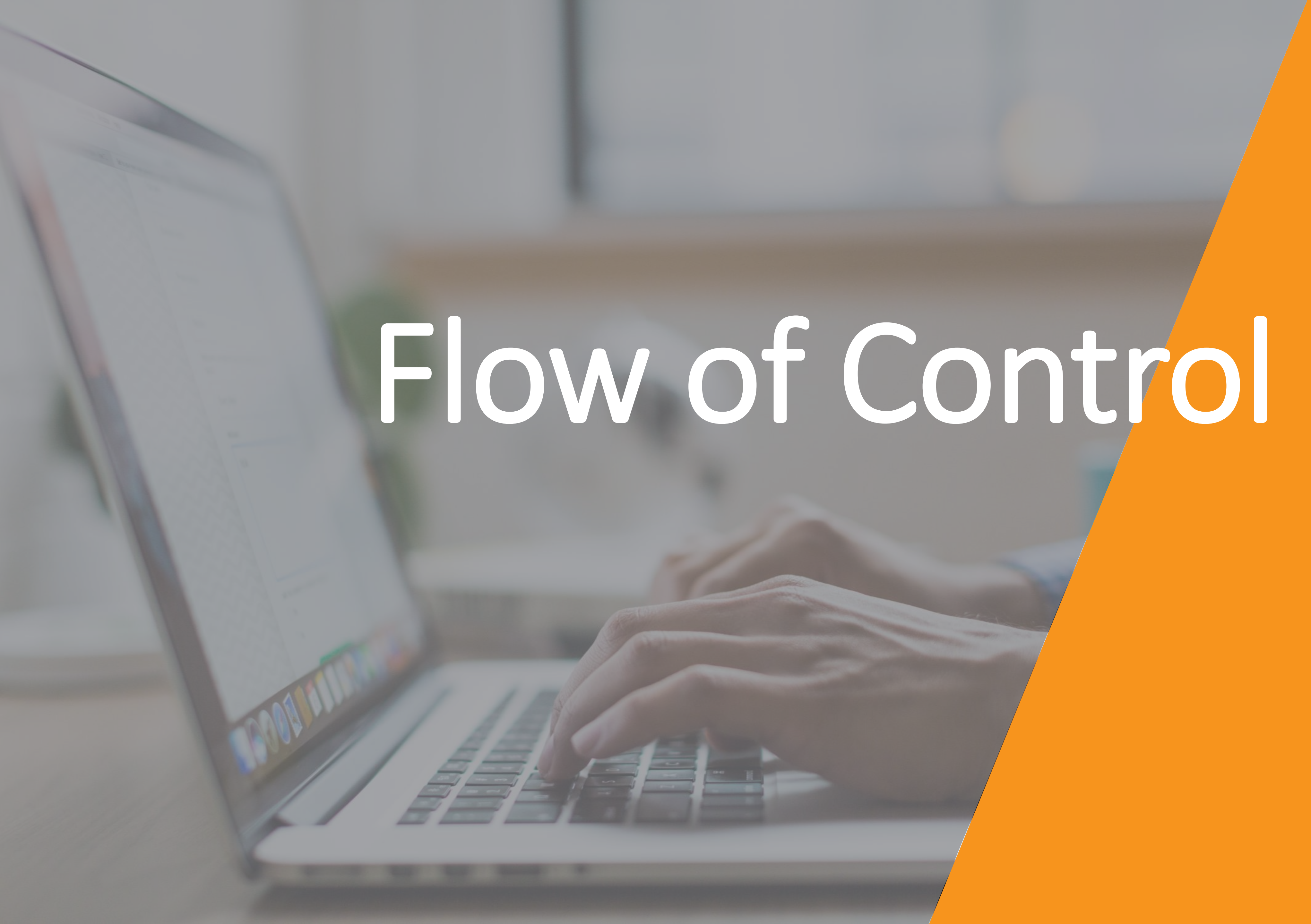
## Topics We’ll Explore Today:

**Flow Control**
About flow control
White space
Conditional expressions
Relational operators
Boolean operators
While loops
Alternate loop exits

**Array types**
About array types (AKA sequences)
Lists and list methods
Tuples
Indexing and slicing
Iterating through a sequence
Nested sequences
Sequence methods, keywords, and operators
List comprehensions
Generator Expressions

**Bonus Topics**
- Advanced Data Types
- Numpy
- Pandas
- Python / SAS Comparison

# Flow of Control



# Why are we starting with Booleans?

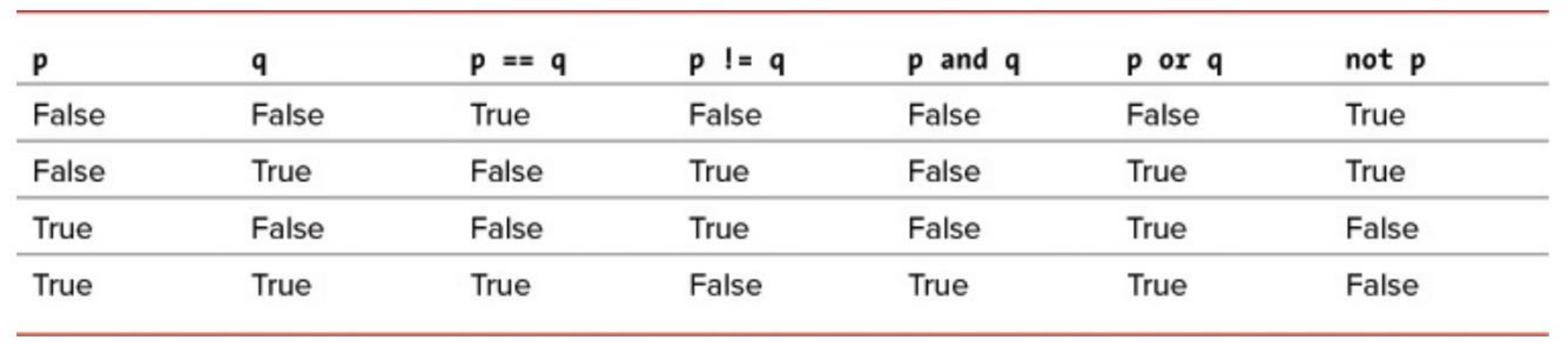
- Boolean expressions are used to control both if statements and loops, it is important to understand how they are evaluated.

# Boolean Logic

- Boolean logic is all about manipulating so-called truth values, which in Python are written True and False.

- We combine Boolean values using four main logical operators (or logical connectives): not, and, or, and ==.

# Boolean Logic



| `p` | `q` | `p == q` | `p != q` | `p and q` | `p or q` | `not p` |
|---|---|---|---|---|---|---|
| False | False | True | False | False | False | True |
| False | True | False | True | False | True | True |
| True | False | False | True | False | True | False |
| True | True | True | False | True | True | False |

# Logical equivalence

Let’s start with ==.

- The expression p == q is True only when p and q both have the same truth value—that is, when p and q are either both True or both False.
- The expression p != q tests if p and q are not the same and returns True only when they have different values.

# COMPARISON OPERATORS ON `int, float, string`

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

**`i > j`**

**`i >= j`**

**`i < j`**

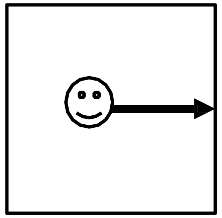
**`i <= j`**

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

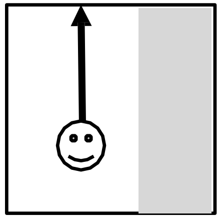
**`i != j`** **inequality** test, `True` if `i` not the same as `j`

# COMPARISON EXAMPLE

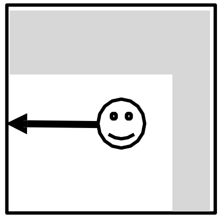
```
pset_time = 15
sleep_time = 8
print(sleep_time > pset_time)
drink = True
derive = False
both = drink and derive
print(both)
```



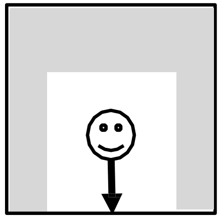
If right clear,
go right



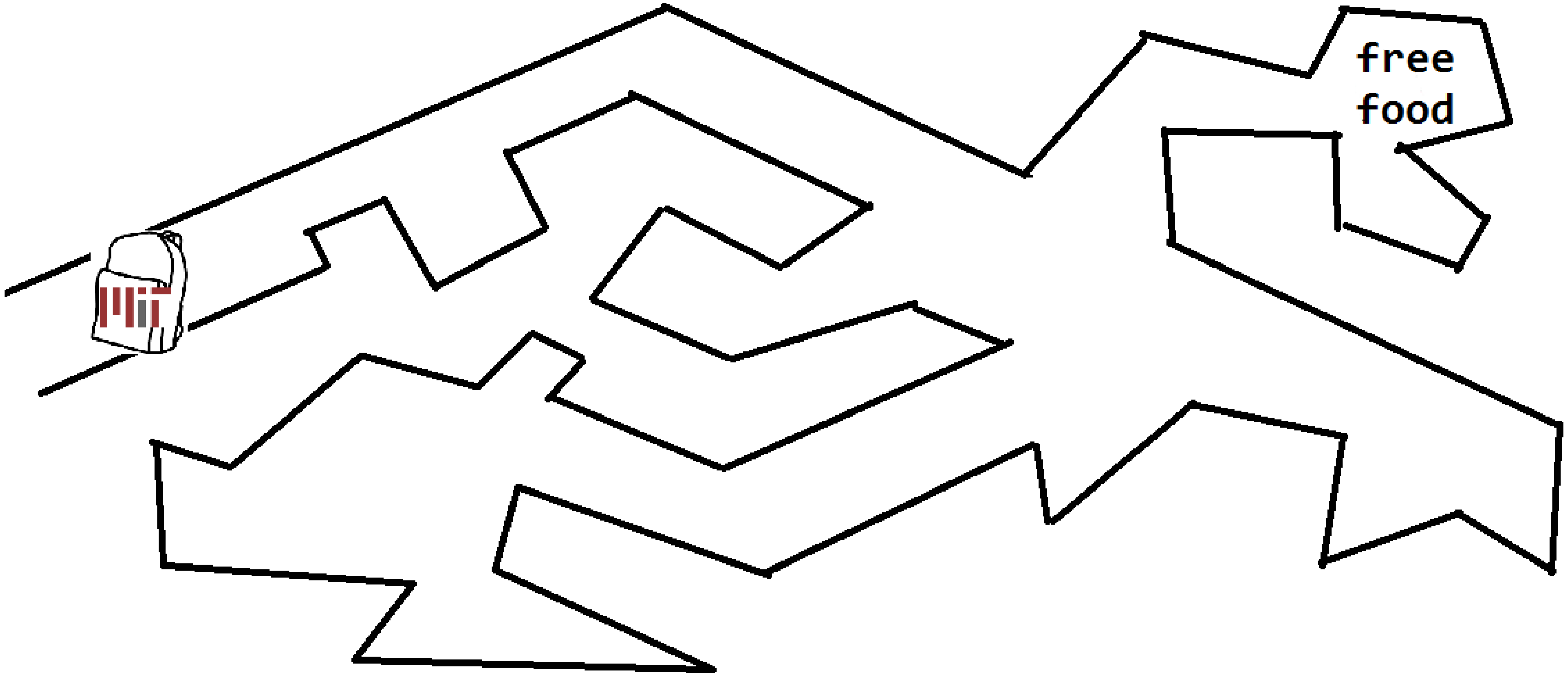
If right blocked,
go forward



If right and
front blocked,
go left



If right , front,
left blocked,
go back



# CONTROL FLOW - BRANCHING

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

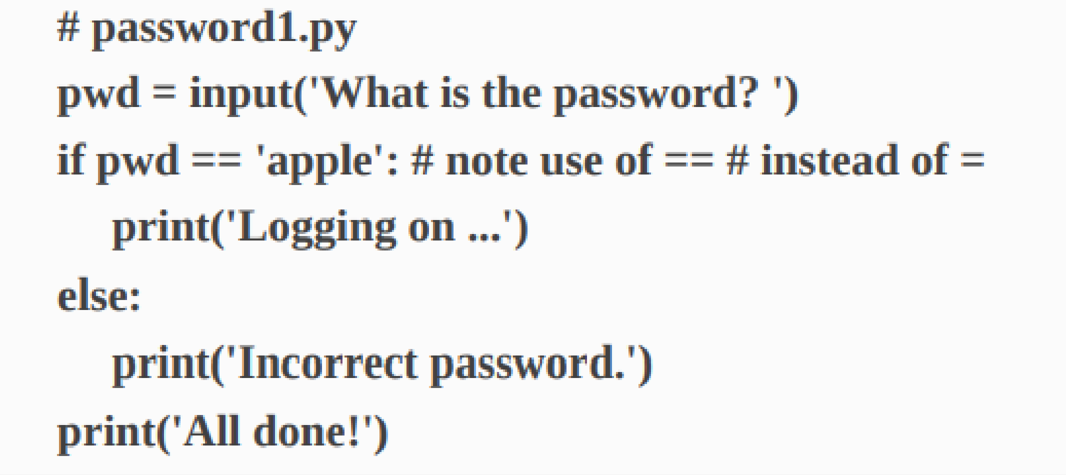
```
if <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

```
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`

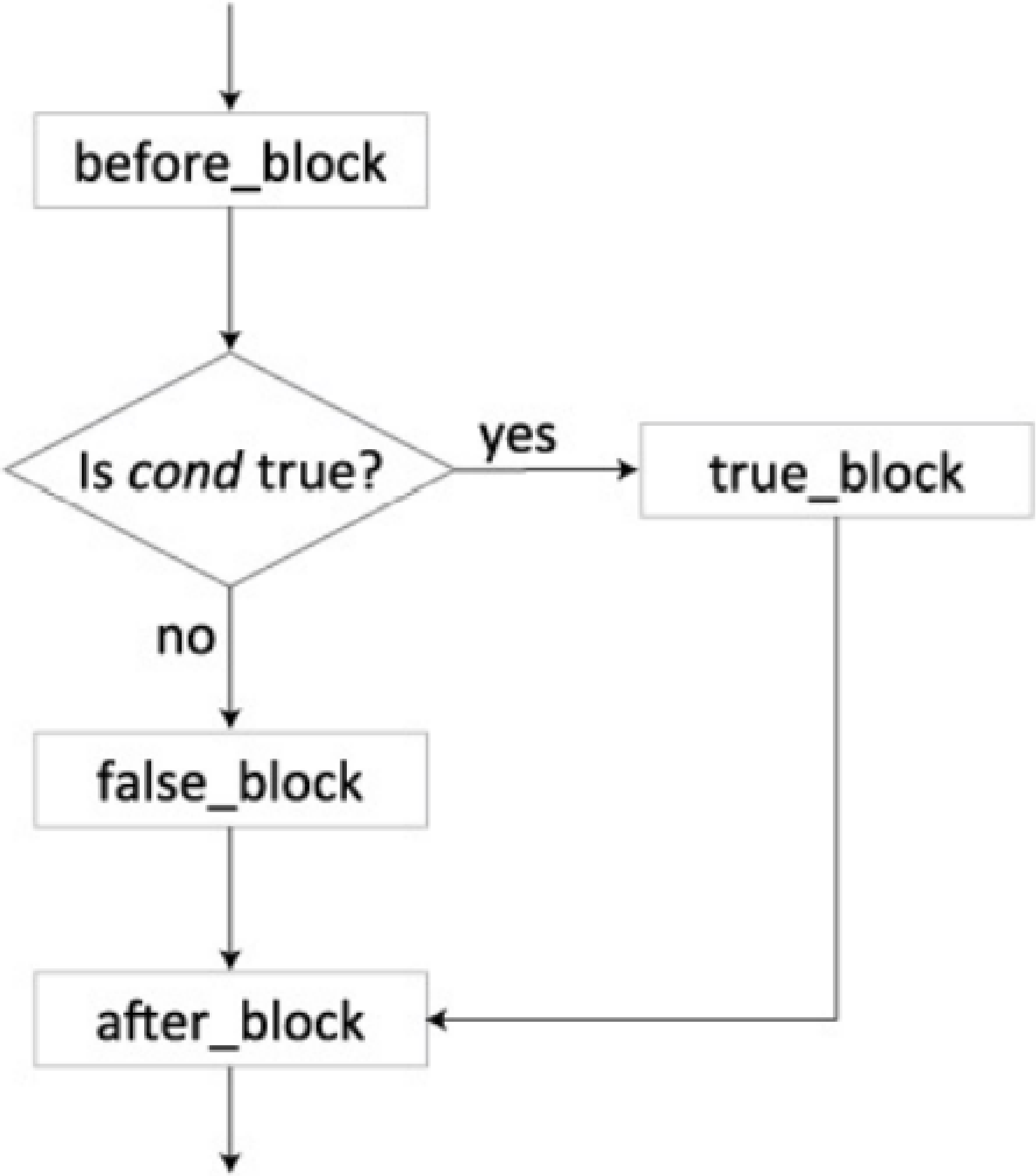
# If/else-statements

- Suppose you are writing a password-checking program.
- The user enters their password, and if it is correct, you log them in to their account.
- If it is not correct, then you tell them they’ve entered the wrong password:



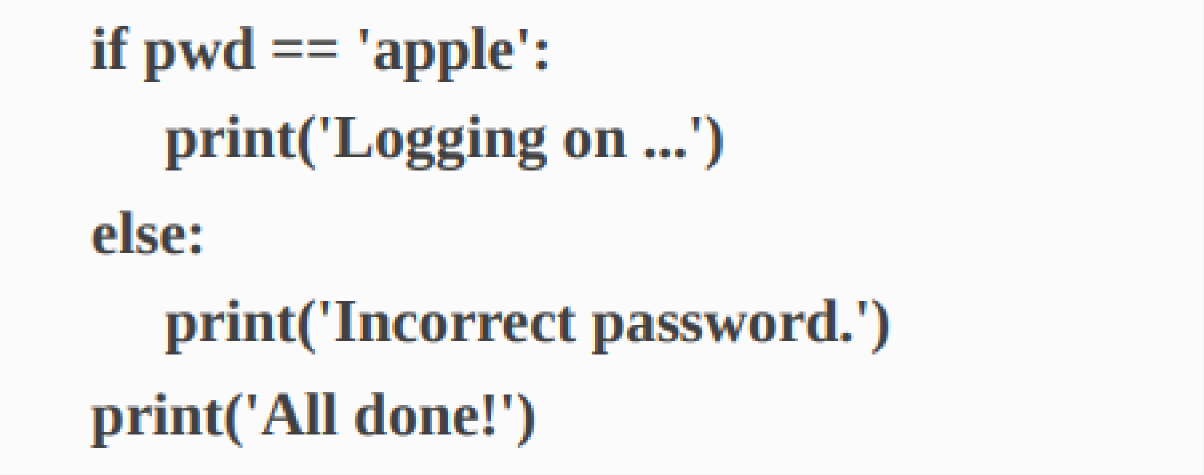
```
# password1.py
pwd = input('What is the password? ')
if pwd == 'apple': # note use of == # instead of =
    print('Logging on ...')
else:
    print('Incorrect password.')
print('All done!')
```

```
before_block
if cond:
   true_block
else:
   false_block
after_block
```



# Code Blocks and Indentation

- One of the most distinctive features of Python is its use of indentation to mark blocks of code.
- Consider the if statement from our password-checking program:

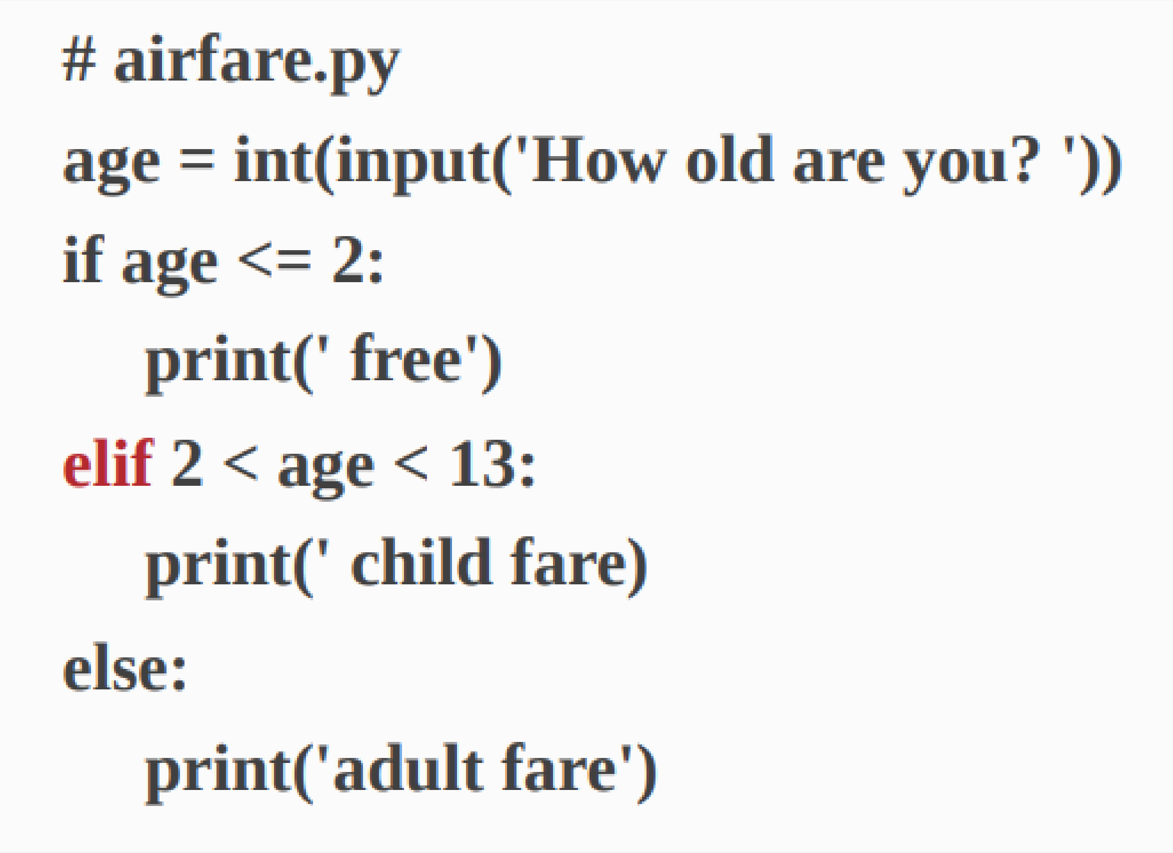


```
if pwd == 'apple':
    print('Logging on ...')
else:
    print('Incorrect password.')
print('All done!')
```

# If/elif-statements

- Program determines how much a passenger should pay:

- “child” ticket rates: Kids 2 years old or younger fly for free, kids older than 2 but younger than 13 pay a discounted child fare, and anyone 13 years or older pays a regular adult fare



```
# airfare.py
age = int(input('How old are you? '))
if age <= 2:
    print(' free')
elif 2 < age < 13:
    print(' child fare)
else:
    print('adult fare')
```

# Loops

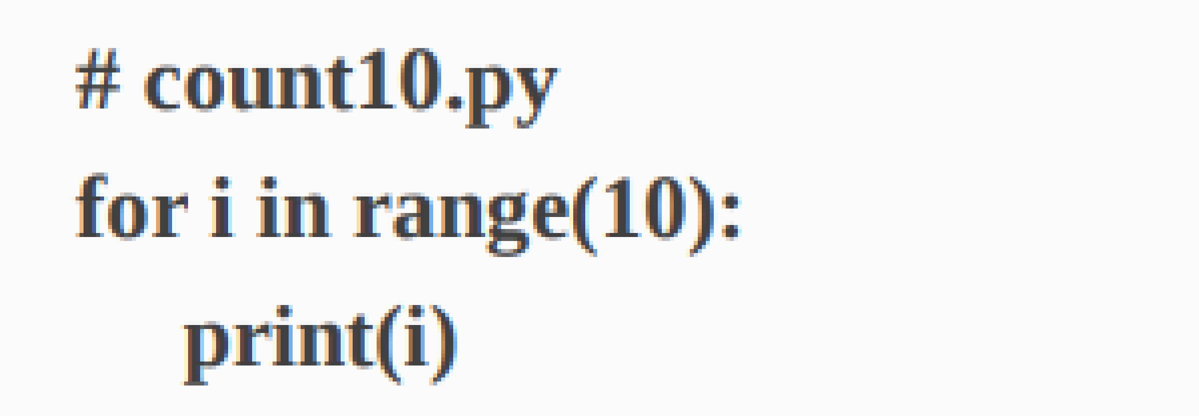
- Now we turn to loops, which are used to repeatedly execute blocks of code.

- Python has two main kinds of loops: for loops and while-loops.

- For-loops are generally easier to use and less error prone than while-loops, although not quite as flexible

# For-loops

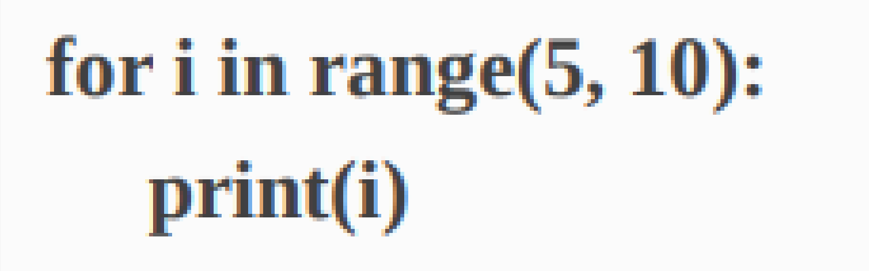
- The basic for-loop repeats a given block of code some specified number of times.
- For example, this snippet of code prints the numbers 0 to 9 on the screen:



```
# count10.py
for i in range(10):
    print(i)
```

# For-loops

- If you want to change the starting value of the loop, add a starting value to range:

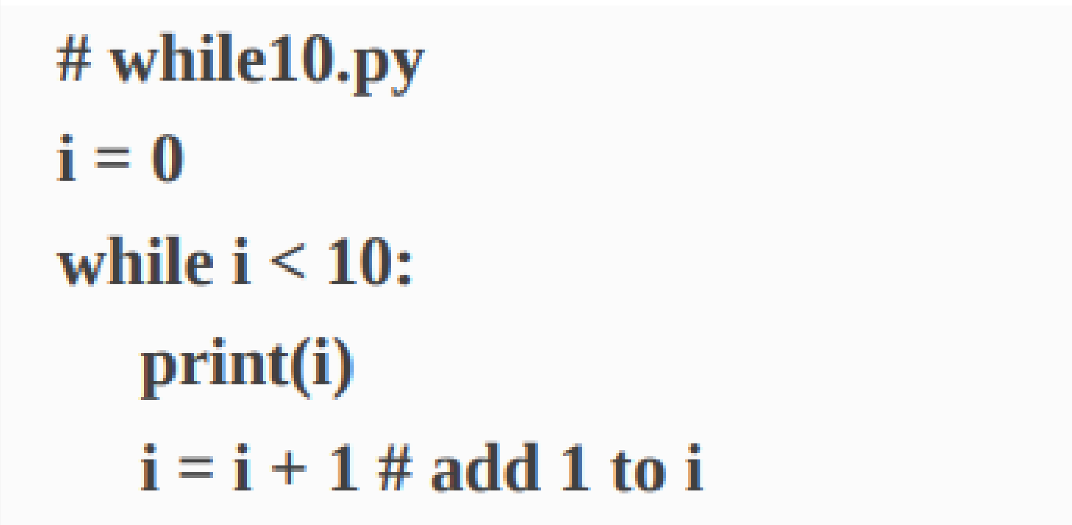


```
for i in range(5, 10):
    print(i)
```

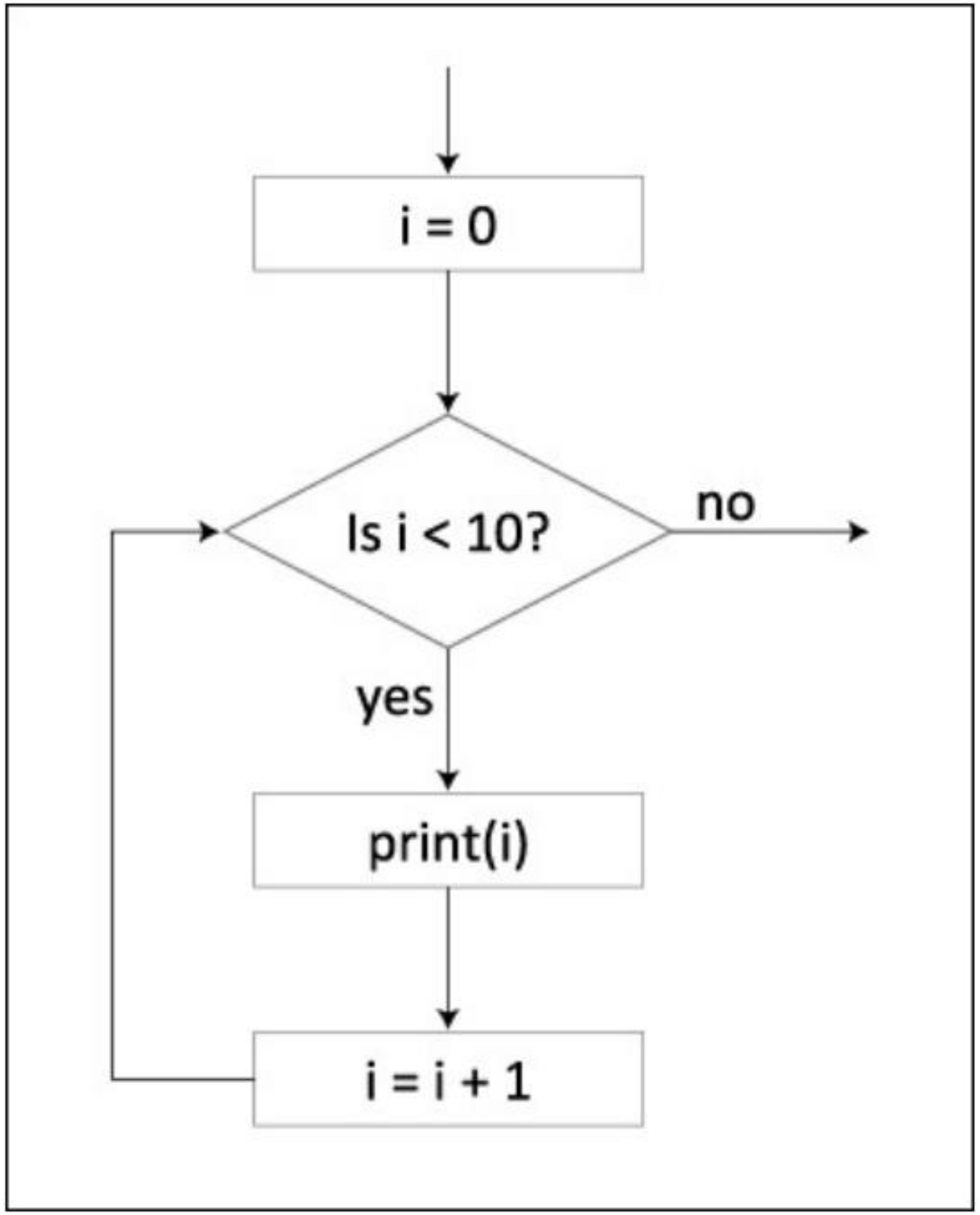
- This prints the numbers from 5 to 9.

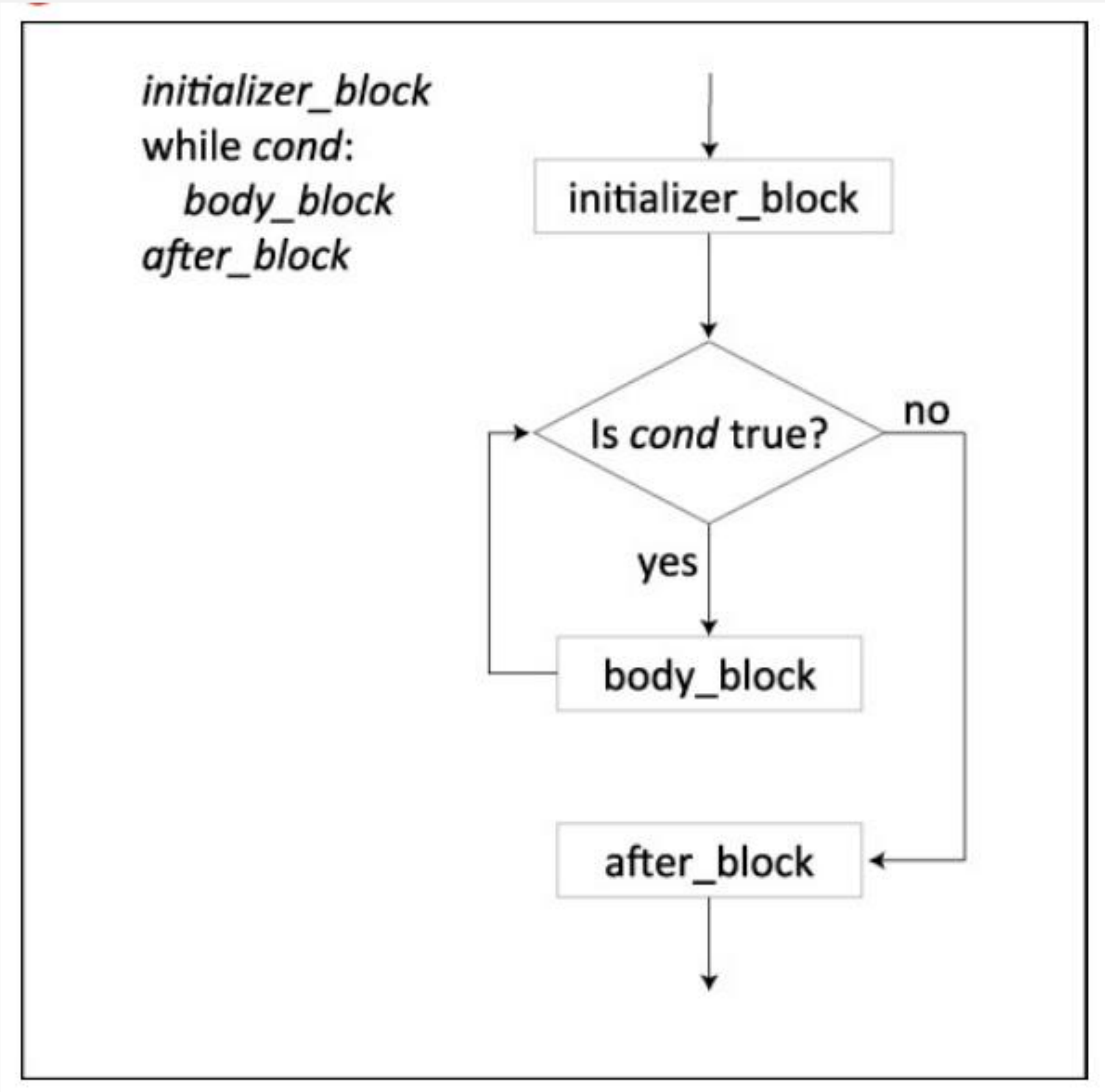
# While-loops

- The second kind of Python loop is a while-loop. Consider this program:



```
# while10.py
i = 0
while i < 10:
    print(i)
    i = i + 1 # add 1 to i
```


i = 0
Is i < 10?
no
yes
print(i)
i = i + 1



```
initializer_block
while cond:
   body_block
after_block
```

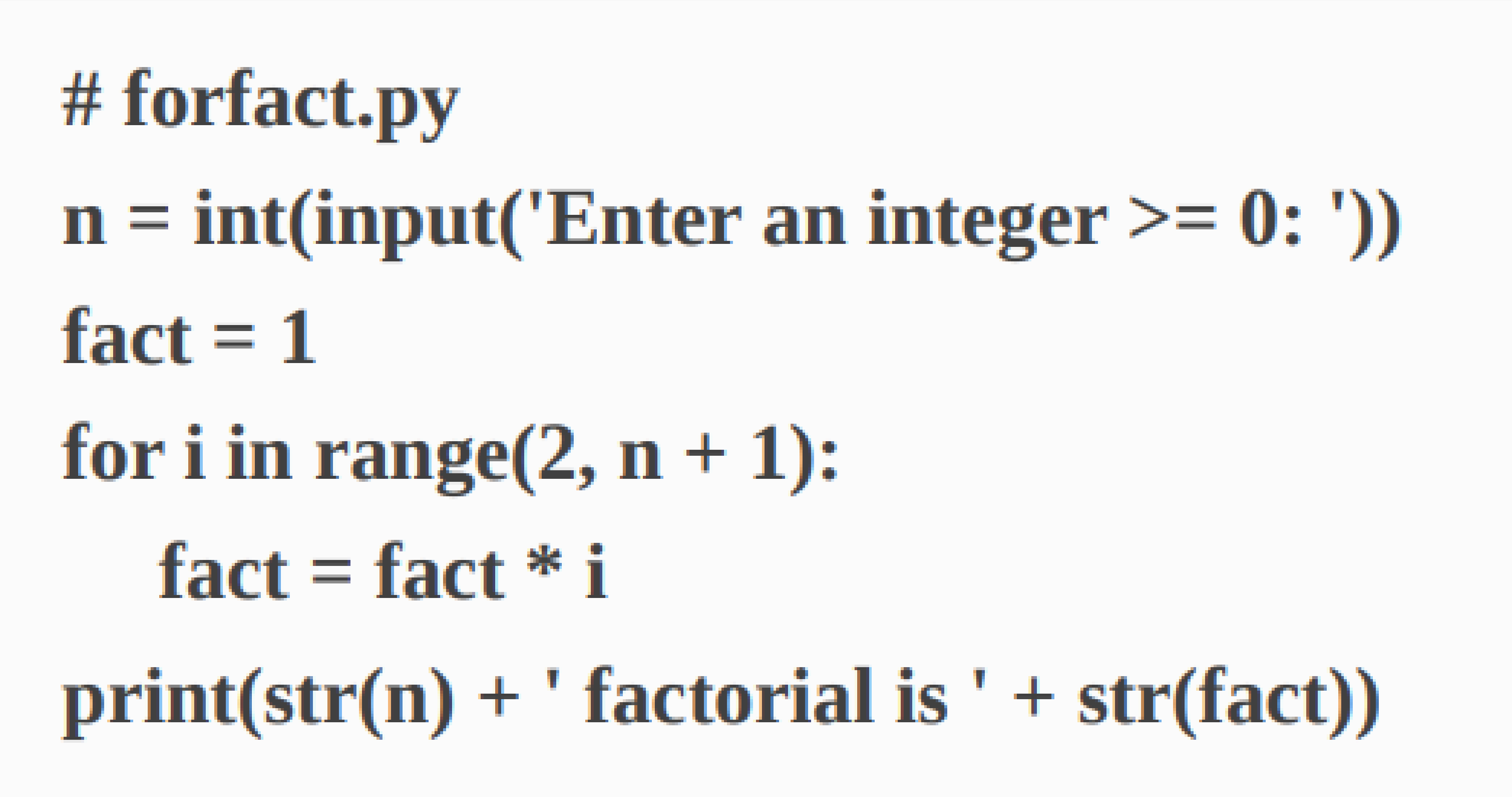
- The general form of a while-loop is shown in the flow chart.

# Comparing FOR and WHILE Loops

- Let’s take a look at a few examples of how for-loops and while-loops can be used to solve the same problems.

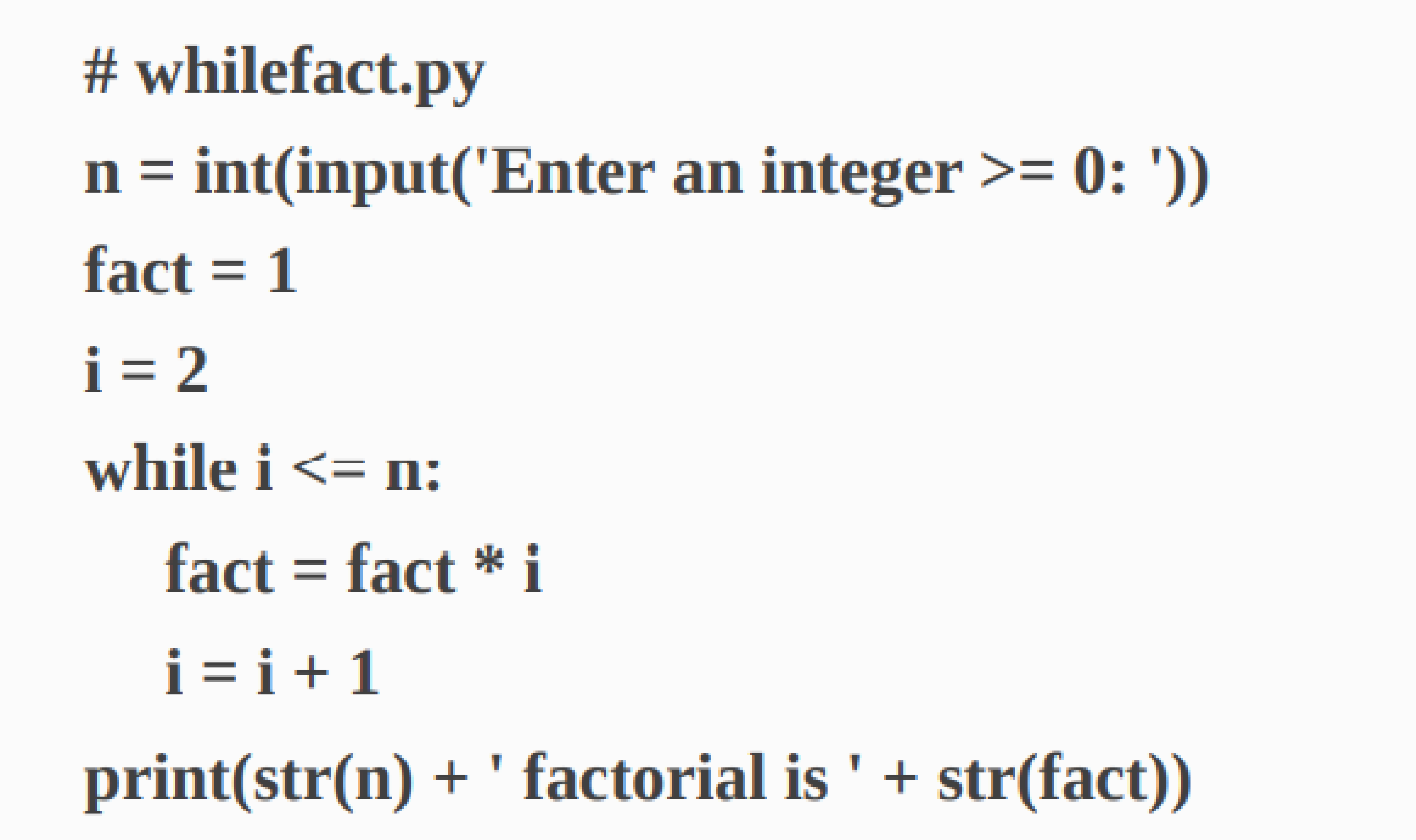
- We’ll see a simple program that can’t be written using a for loop.

# Calculating factorials



```
# forfact.py
n = int(input('Enter an integer >= 0: '))
fact = 1
for i in range(2, n + 1):
    fact = fact * i
print(str(n) + ' factorial is ' + str(fact))
```

# Here’s another way to do it using a while-loop:



```
# whilefact.py
n = int(input('Enter an integer >= 0: '))
fact = 1
i = 2
while i <= n:
    fact = fact * i
    i = i + 1
print(str(n) + ' factorial is ' + str(fact))
```

# for VS while LOOPS

for loops

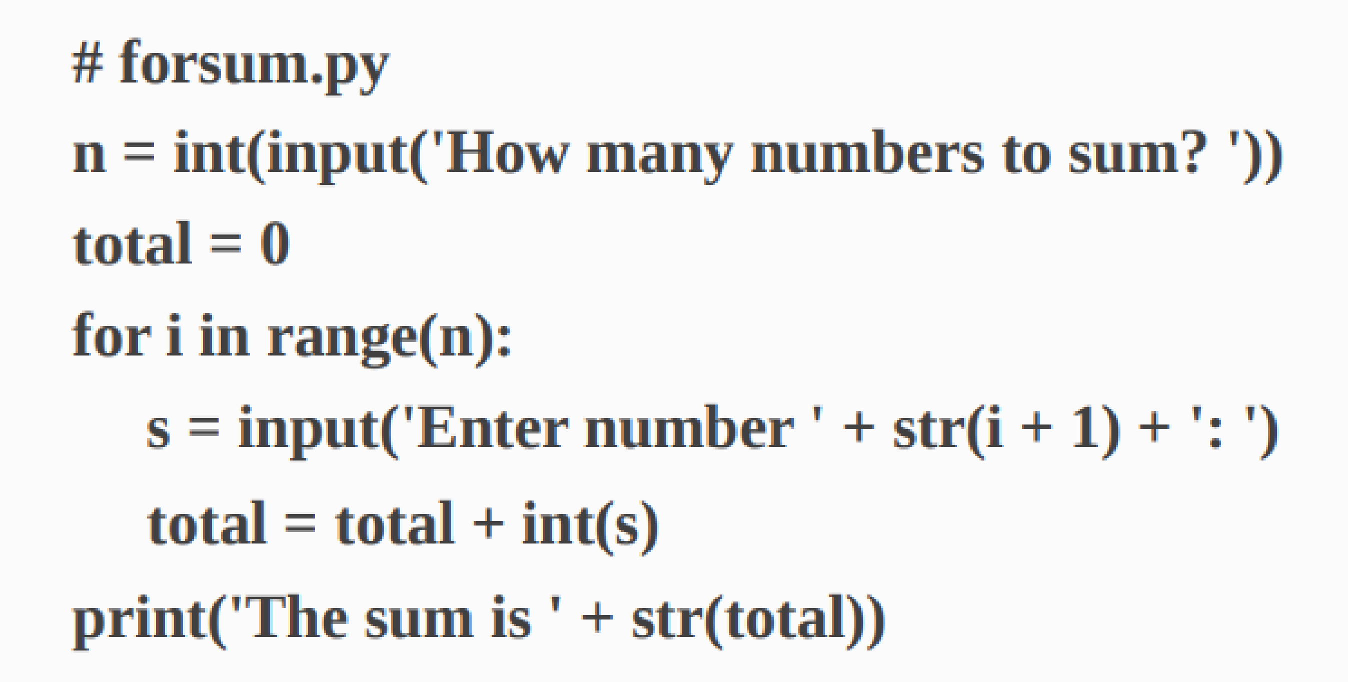
- **know** number of iterations
- can **end early** via break
- uses a **counter**
- **can rewrite** a for loop using a while loop

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

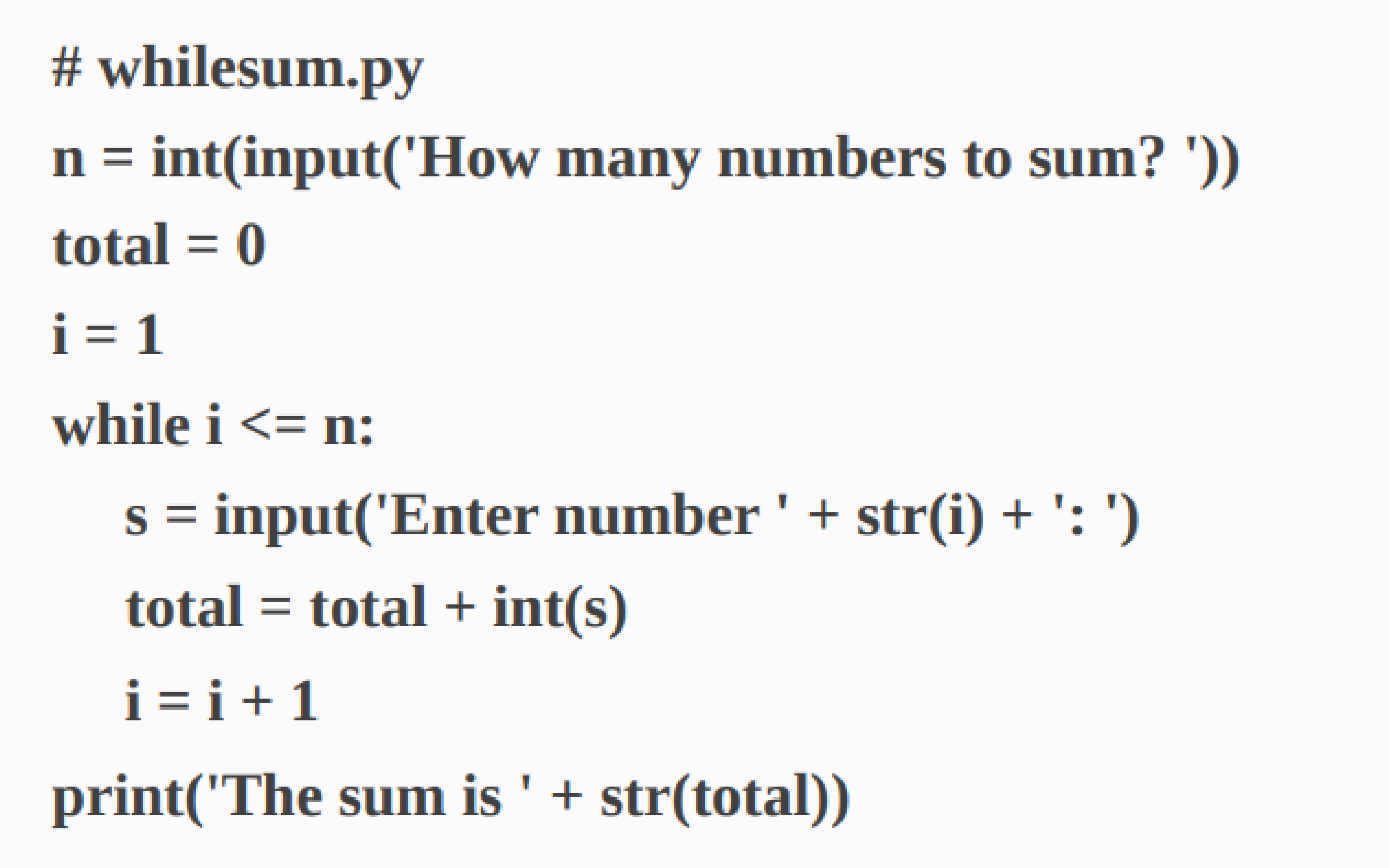
# Summing numbers from the user

- The following programs ask the user to enter some numbers, and then prints their sum.
- Here is a version using a for-loop:



```
# forsum.py
n = int(input('How many numbers to sum? '))
total = 0
for i in range(n):
    s = input('Enter number ' + str(i + 1) + ': ')
    total = total + int(s)
print('The sum is ' + str(total))
```

# Program that does that same thing using a while-loop:



```
# whilesum.py
n = int(input('How many numbers to sum? '))
total = 0
i = 1
while i <= n:
    s = input('Enter number ' + str(i) + ': ')
    total = total + int(s)
    i = i + 1
print('The sum is ' + str(total))
```

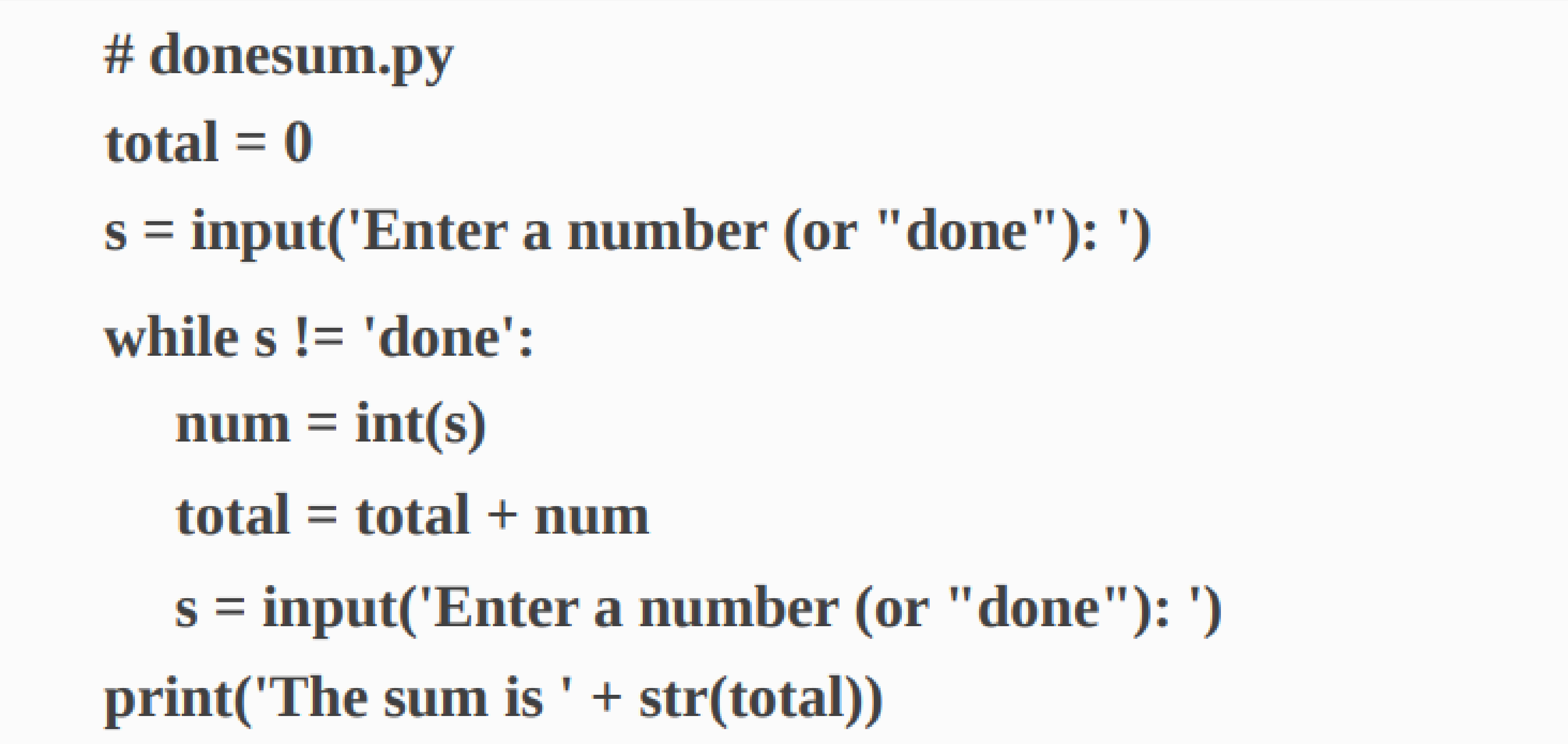
# Summing an unknown number of numbers

- Suppose we want to let users enter a list of numbers to be summed without asking them ahead of time how many numbers they have.

- Instead, they just type 'done' when they have no more numbers to add.

# Summing an unknown number of numbers

- Here’s how to do it using a while-loop:



```
# donesum.py
total = 0
s = input('Enter a number (or "done"): ')
while s != 'done':
    num = int(s)
    total = total + num
    s = input('Enter a number (or "done"): ')
print('The sum is ' + str(total))
```

# Summing an unknown number of numbers

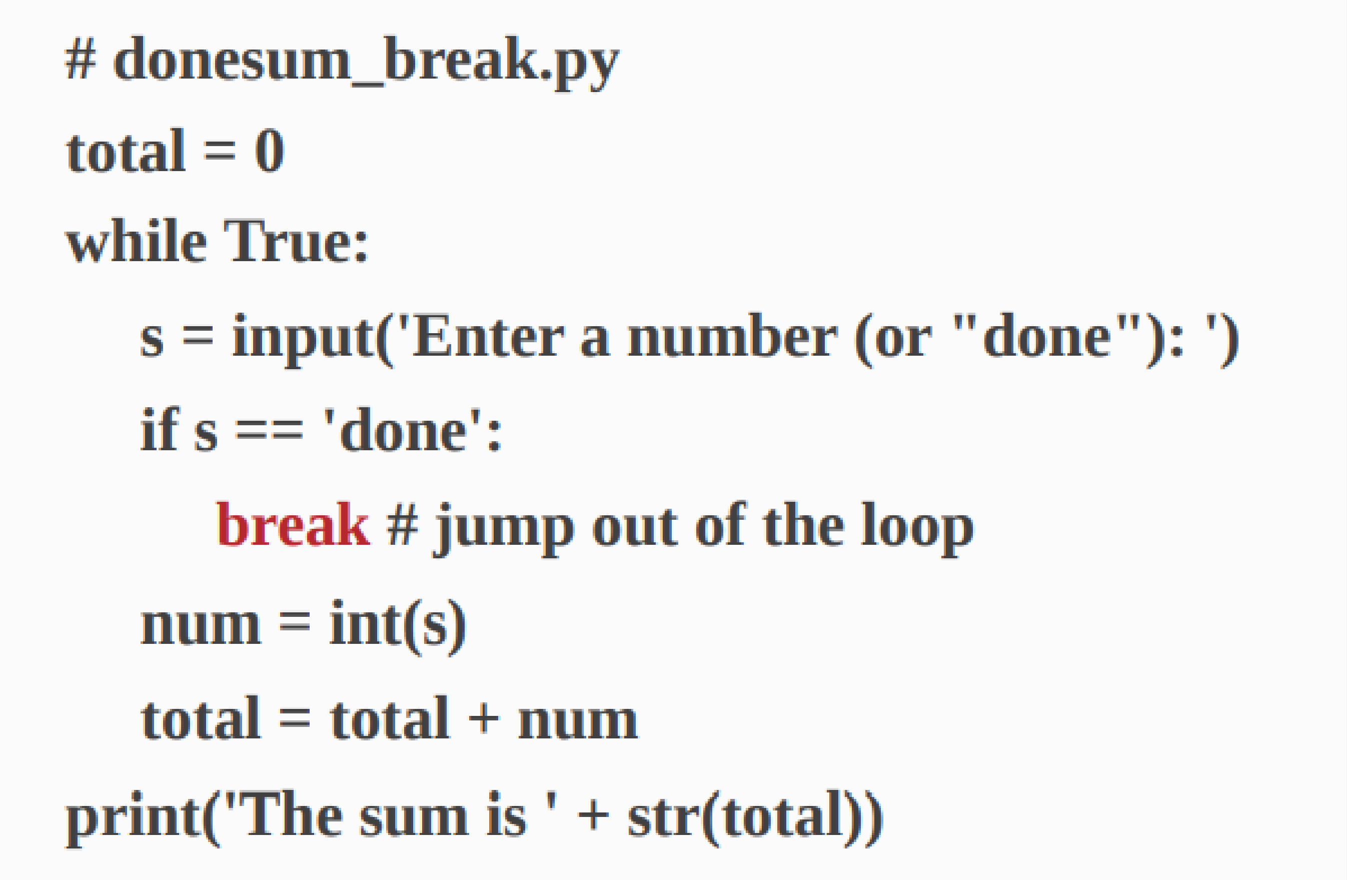
- We convert the input string s to an integer only after we know s is not the string 'done'. If we had written

```
s = int(input('Enter a number (or "done"): '))
```

# Breaking Out of Loops and Blocks

- The break statement is a handy way for exiting a loop from anywhere within the loop’s body.

- For example, here is an alternative way to sum an unknown number of numbers:

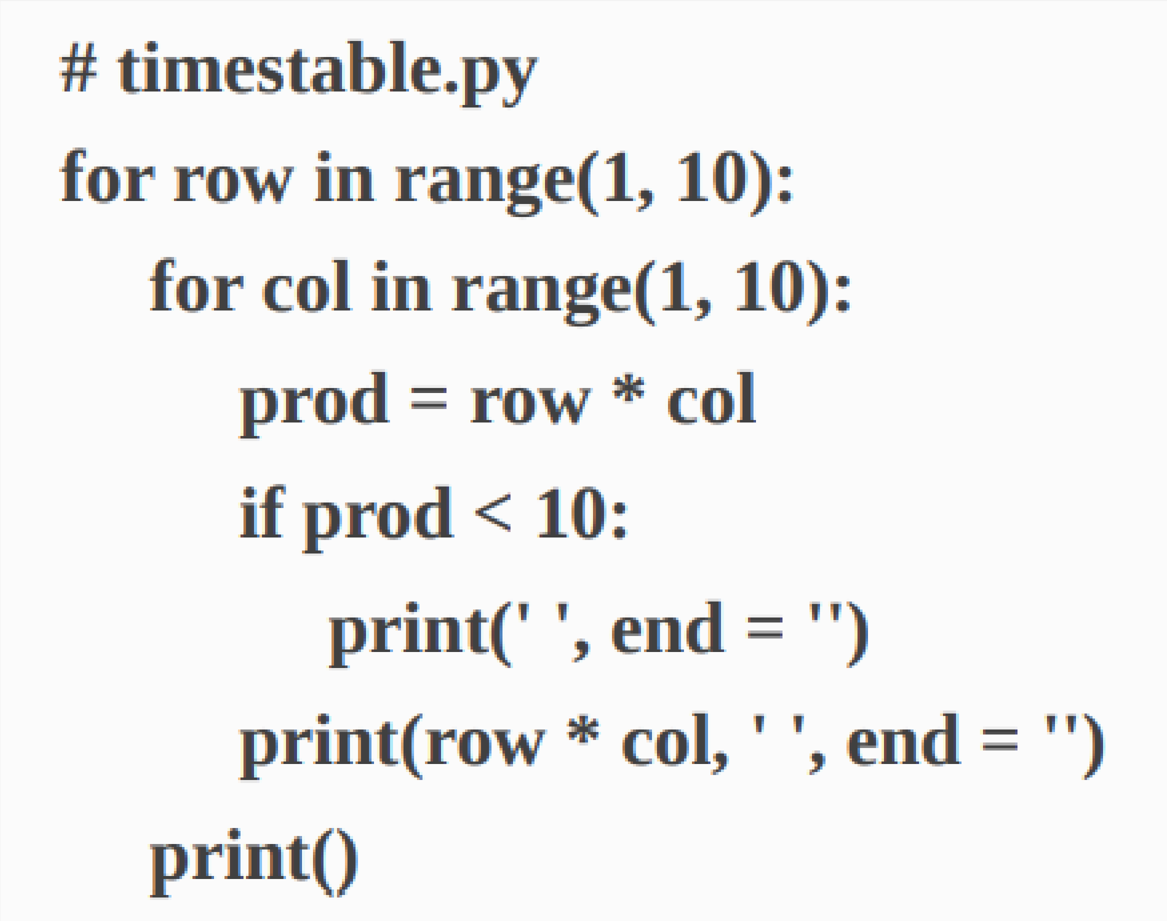


```
# donesum_break.py
total = 0
while True:
    s = input('Enter a number (or "done"): ')
    if s == 'done':
        break # jump out of the loop
    num = int(s)
    total = total + num
print('The sum is ' + str(total))
```

# Loops Within Loops

- Loops within loops, also known as nested loops, occur frequently in programming.

- For instance, here’s a program that prints the times tables up to 10:



```
# timestable.py
for row in range(1, 10):
    for col in range(1, 10):
        prod = row * col
        if prod < 10:
            print(' ', end = '')
        print(row * col, ' ', end = '')
    print()
```

# Data Structures

# Data Structures

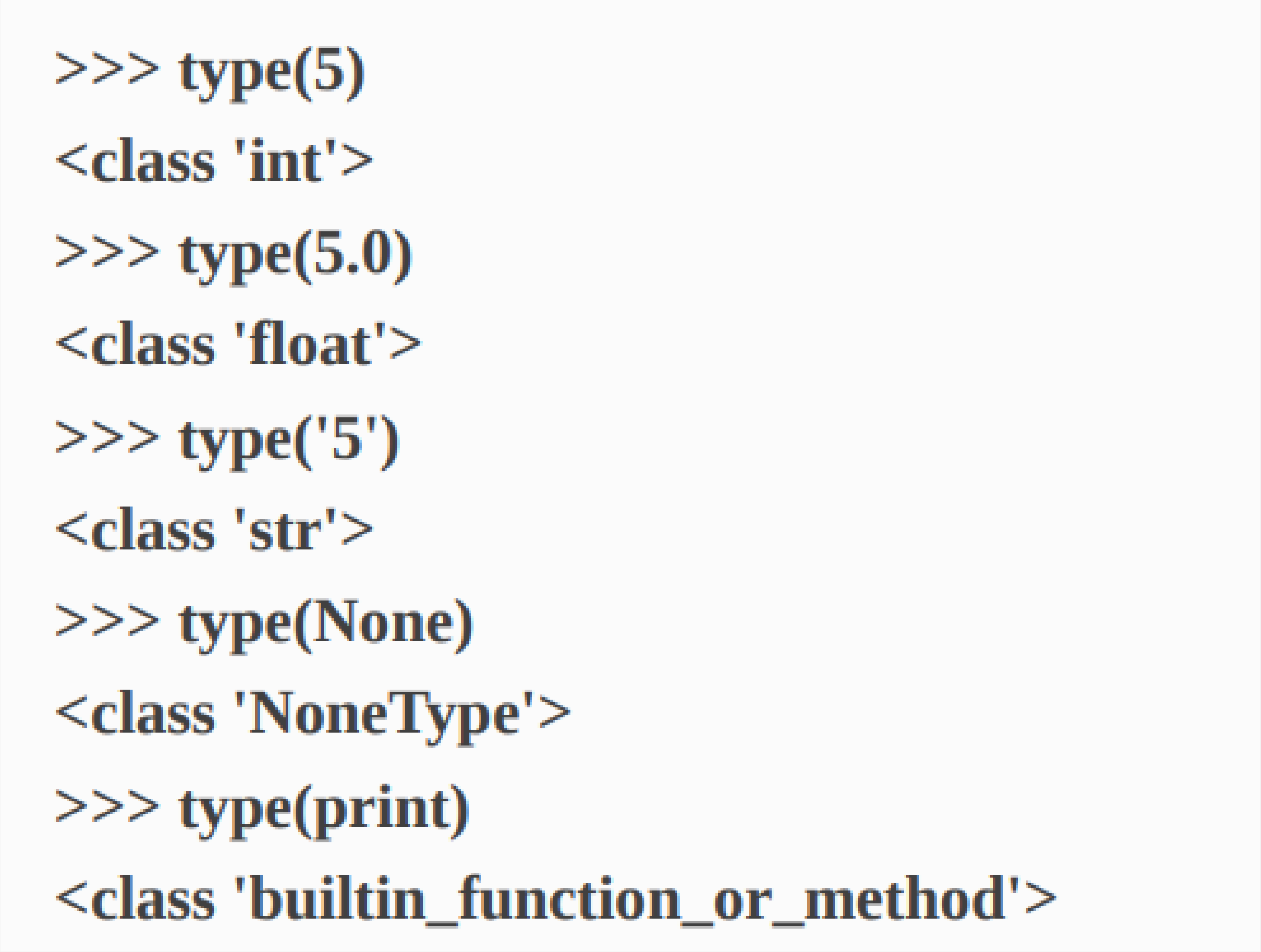
In this lesson, you will learn

- The type Command
- Sequences
- Tuples
- Lists
- List methods
- Sorting Lists
- List Comprehensions
- Dictionaries
- Sets

# The type Command

- It’s occasionally useful to check the data type of a value or a variable.

- This is easily done with the built-in type command:



```
>>> type(5)
<class 'int'>
>>> type(5.0)
<class 'float'>
>>> type('5')
<class 'str'>
>>> type(None)
<class 'NoneType'>
>>> type(print)
<class 'builtin_function_or_method'>
```

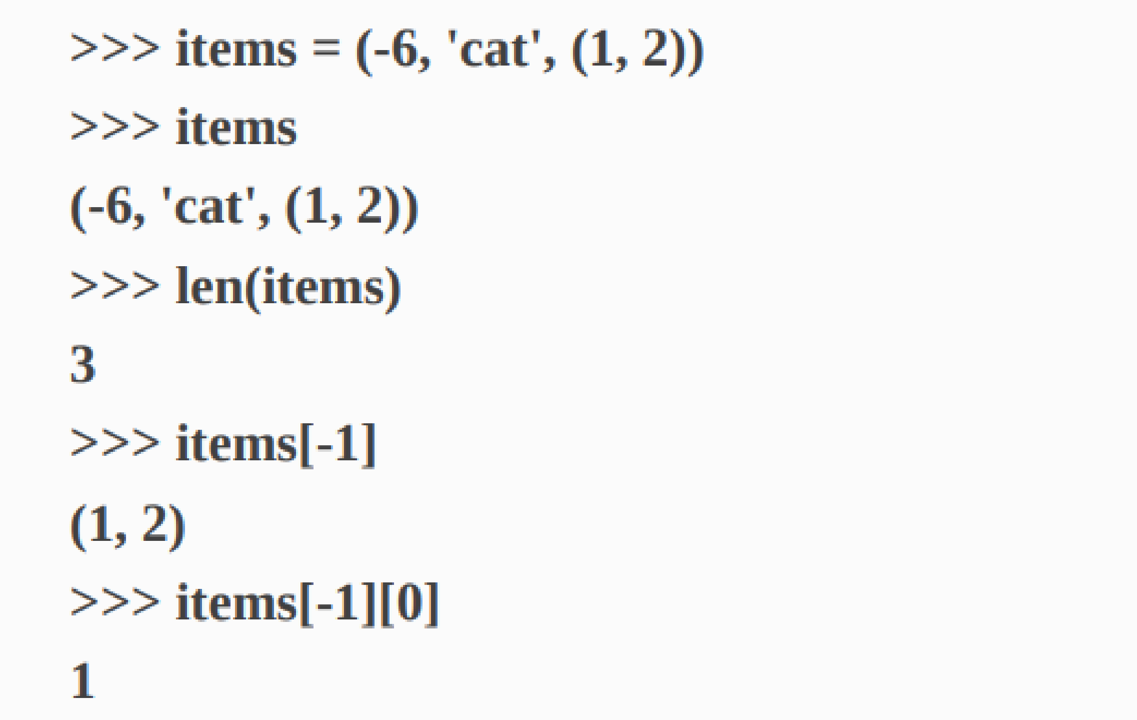
# Sequences

- In Python, a sequence is an ordered collection of values.
- Python has three built-in sequence types: strings, tuples, and lists.
- One very nice feature of sequences is that they can be indexed and sliced, just as we saw for strings in the previous lesson.

# Tuples

- A tuple is an immutable sequence of 0 or more values.
- It can contain any Python value—even other tuples.

For example:



```
>>> items = (-6, 'cat', (1, 2))
>>> items
(-6, 'cat', (1, 2))
>>> len(items)
3
>>> items[-1]
(1, 2)
>>> items[-1][0]
1
```

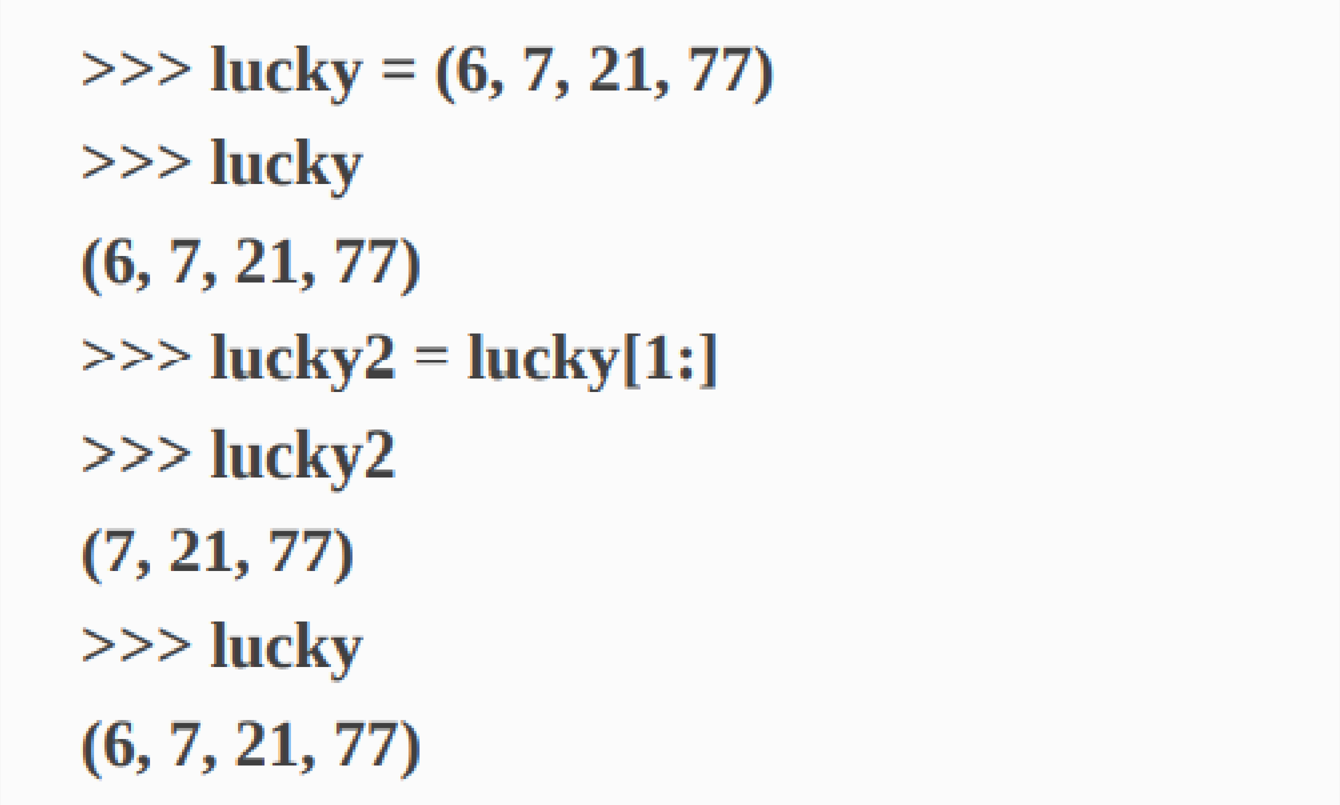
# Tuples

- For instance:

```
>>> type(())
<class 'tuple'>
>>> type((5,))
<class 'tuple'>
>>> type((5))
<class 'int'>
```

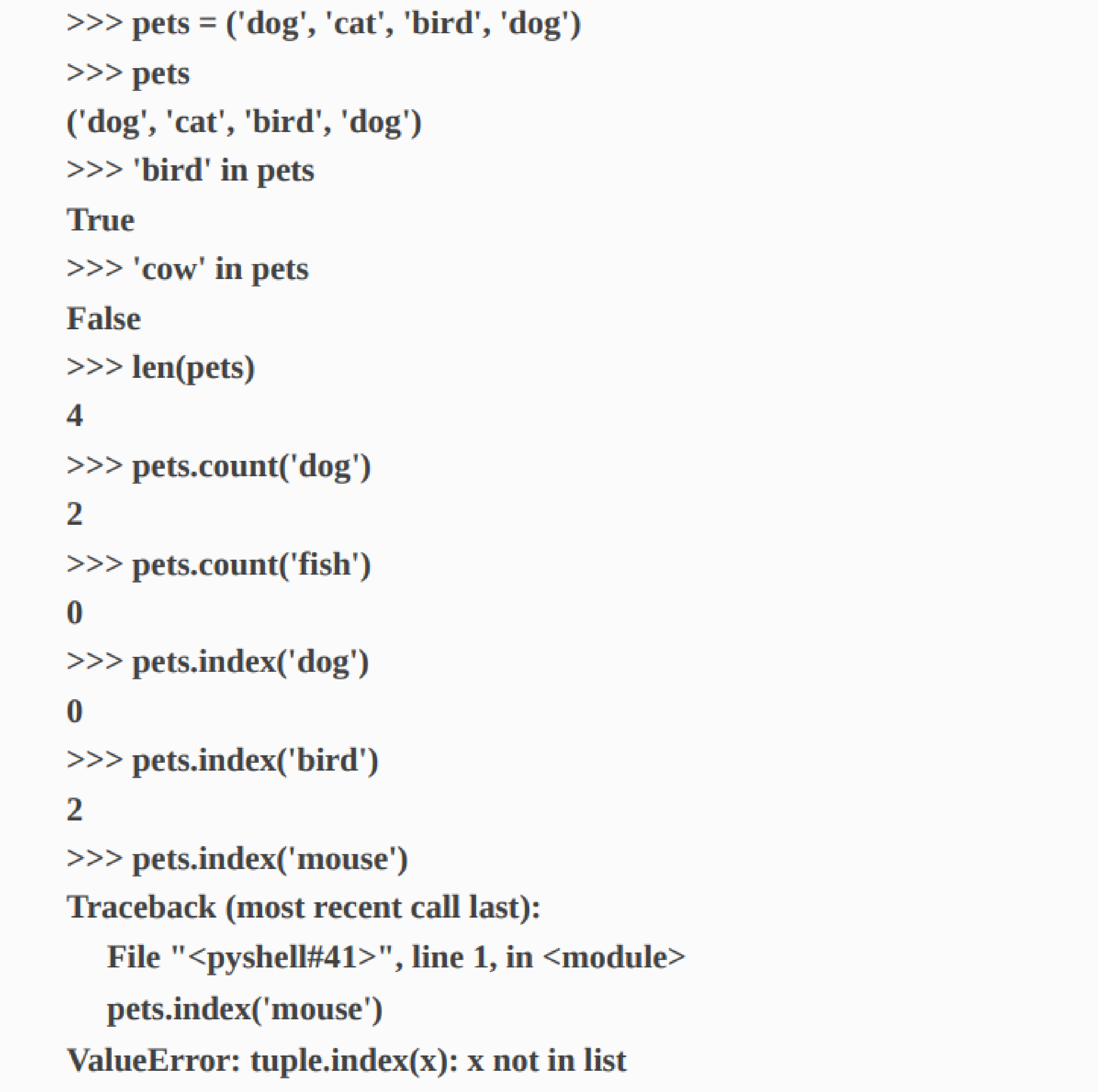
# Tuple immutability

- For example, here’s how you can chop off the first element of a tuple:



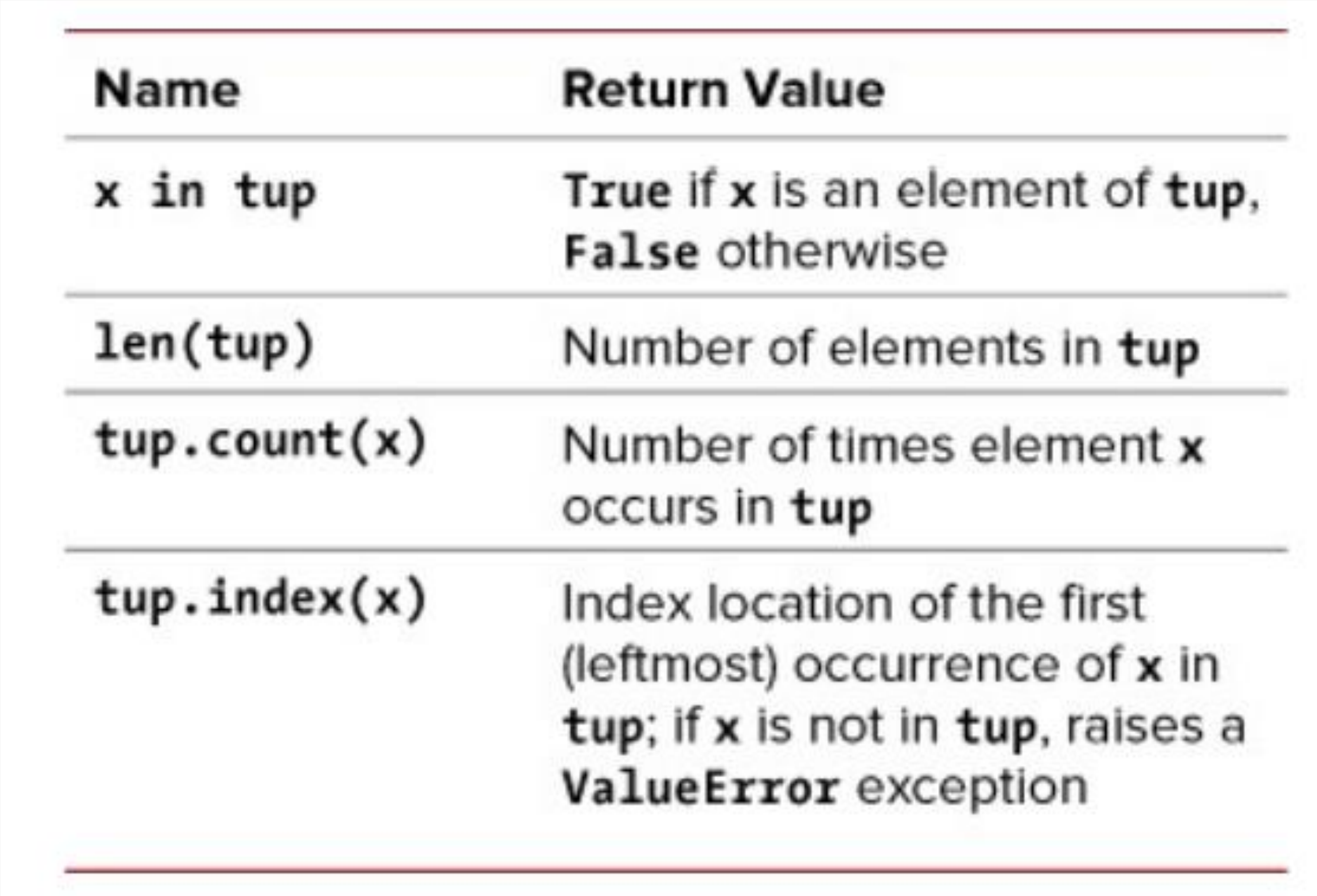
```
>>> lucky = (6, 7, 21, 77)
>>> lucky
(6, 7, 21, 77)
>>> lucky2 = lucky[1:]
>>> lucky2
(7, 21, 77)
>>> lucky
(6, 7, 21, 77)
```

# Tuple methods



```
>>> pets = ('dog', 'cat', 'bird', 'dog')
>>> pets
('dog', 'cat', 'bird', 'dog')
>>> 'bird' in pets
True
>>> 'cow' in pets
False
>>> len(pets)
4
>>> pets.count('dog')
2
>>> pets.count('fish')
0
>>> pets.index('dog')
0
>>> pets.index('bird')
2
>>> pets.index('mouse')
Traceback (most recent call last):
    File "<pyshell#41>", line 1, in <module>
    pets.index('mouse')
ValueError: tuple.index(x): x not in list
```

# Tuple methods



| Name | Return Value |
|---|---|
| `x in tup` | `True` if `x` is an element of `tup`, `False` otherwise |
| `len(tup)` | Number of elements in `tup` |
| `tup.count(x)` | Number of times element `x` occurs in `tup` |
| `tup.index(x)` | Index location of the first (leftmost) occurrence of `x` in `tup`; if `x` is not in `tup`, raises a `ValueError` exception |

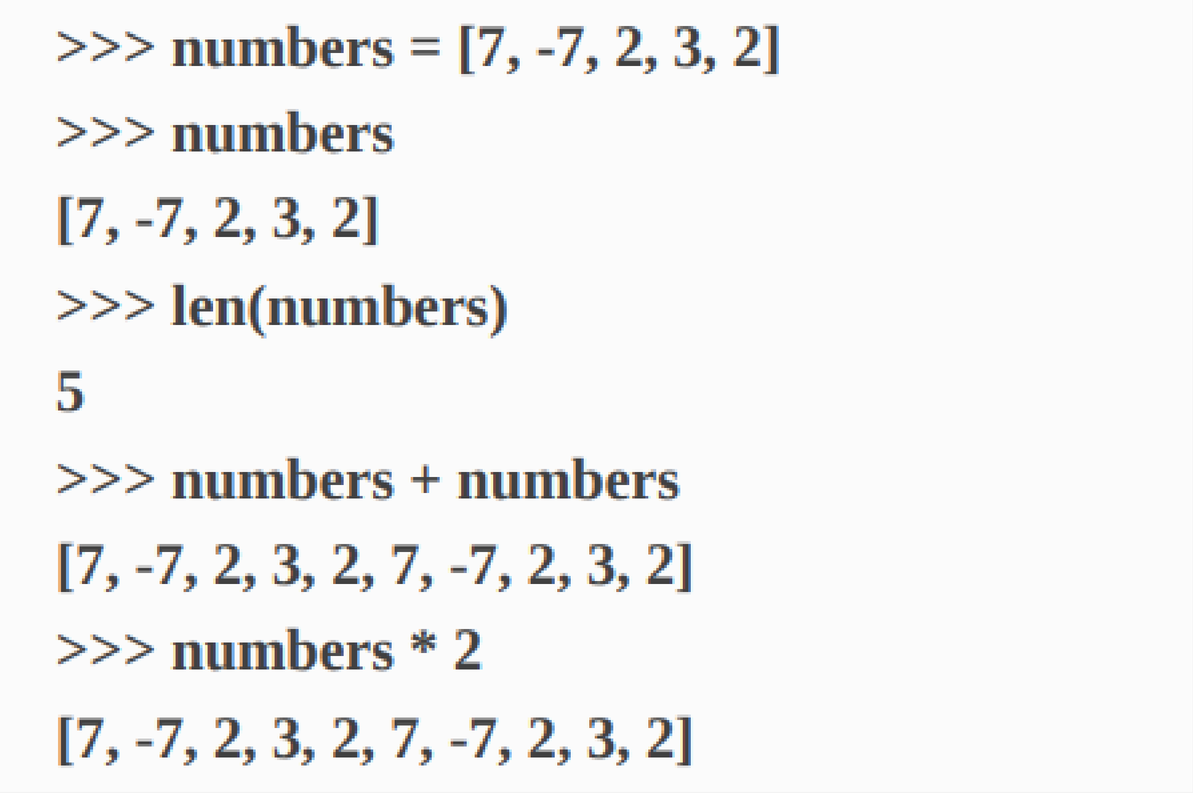
# Tuple methods

- As with strings, you can use + and * to concatenate tuples:

```
>>> tup1 = (1, 2, 3)
>>> tup2 = (4, 5, 6)
>>> tup1 + tup2
(1, 2, 3, 4, 5, 6)
>>> tup1 * 2
(1, 2, 3, 1, 2, 3)
```

# Lists

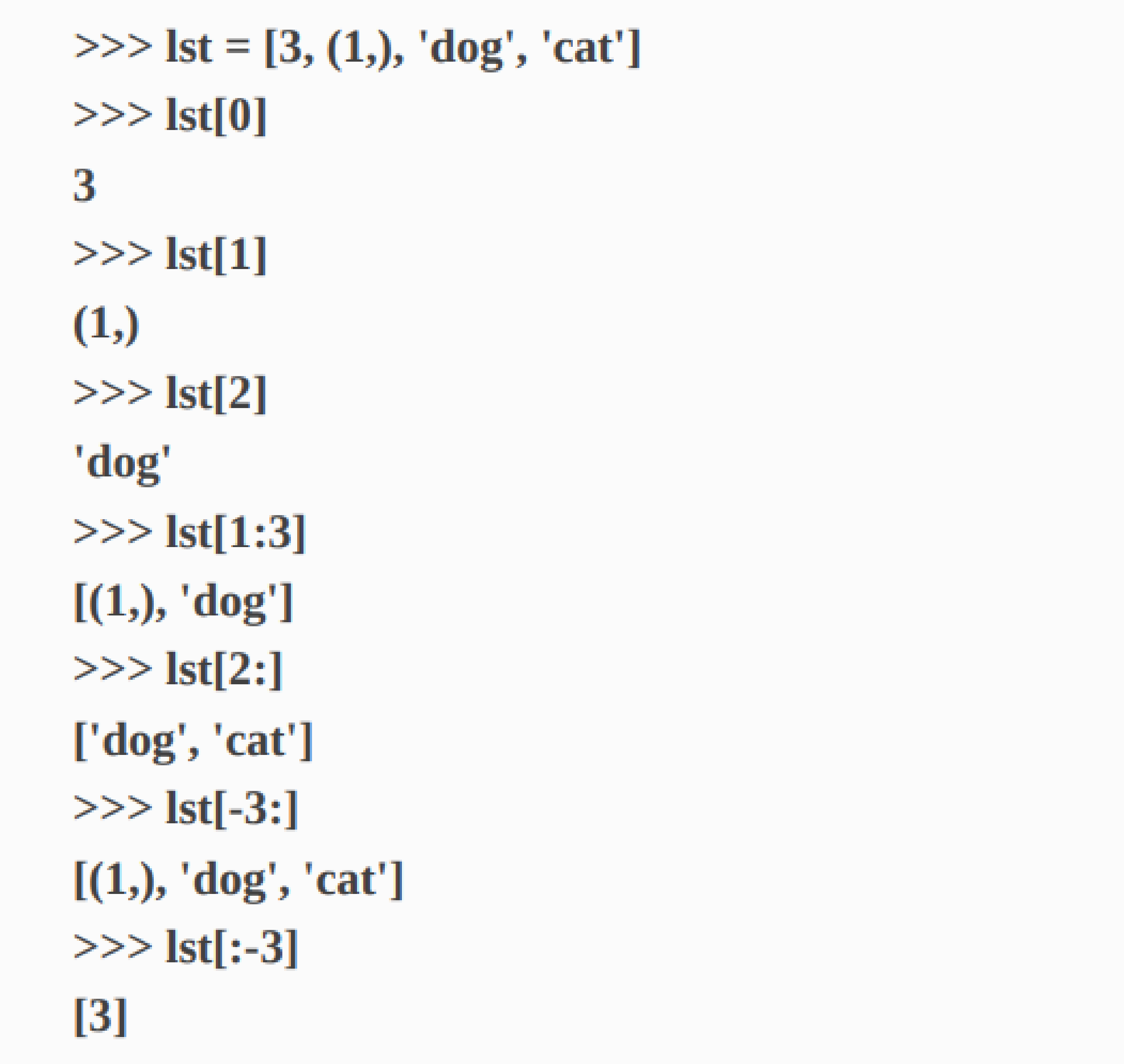
- Lists are essentially the same as tuples but with one key difference: Lists are mutable.



```
>>> numbers = [7, -7, 2, 3, 2]
>>> numbers
[7, -7, 2, 3, 2]
>>> len(numbers)
5
>>> numbers + numbers
[7, -7, 2, 3, 2, 7, -7, 2, 3, 2]
>>> numbers * 2
[7, -7, 2, 3, 2, 7, -7, 2, 3, 2]
```

# Lists

- And just as with strings and tuples, you can use indexing and slicing to access individual elements and sublists:

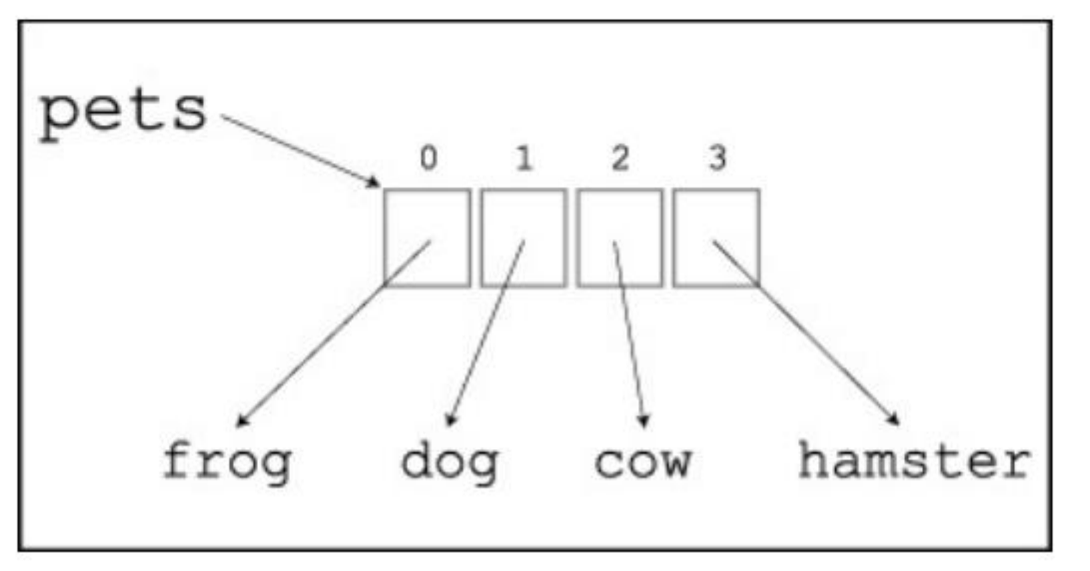


```
>>> lst = [3, (1,), 'dog', 'cat']
>>> lst[0]
3
>>> lst[1]
(1,)
>>> lst[2]
'dog'
>>> lst[1:3]
[(1,), 'dog']
>>> lst[2:]
['dog', 'cat']
>>> lst[-3:]
[(1,), 'dog', 'cat']
>>> lst[:-3]
[3]
```

# Mutability

- Mutability is the key feature that distinguishes lists from tuples. For example:

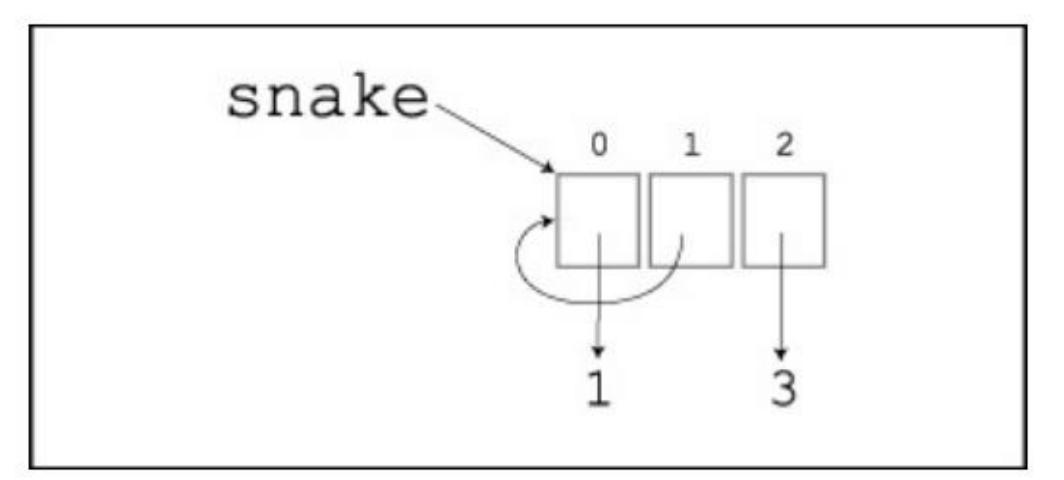
```
>>> pets = ['frog', 'dog', 'cow', 'hamster']
>>> pets
['frog', 'dog', 'cow', 'hamster']
>>> pets[2] = 'cat'
>>> pets
['frog', 'dog', 'cat', 'hamster']
```



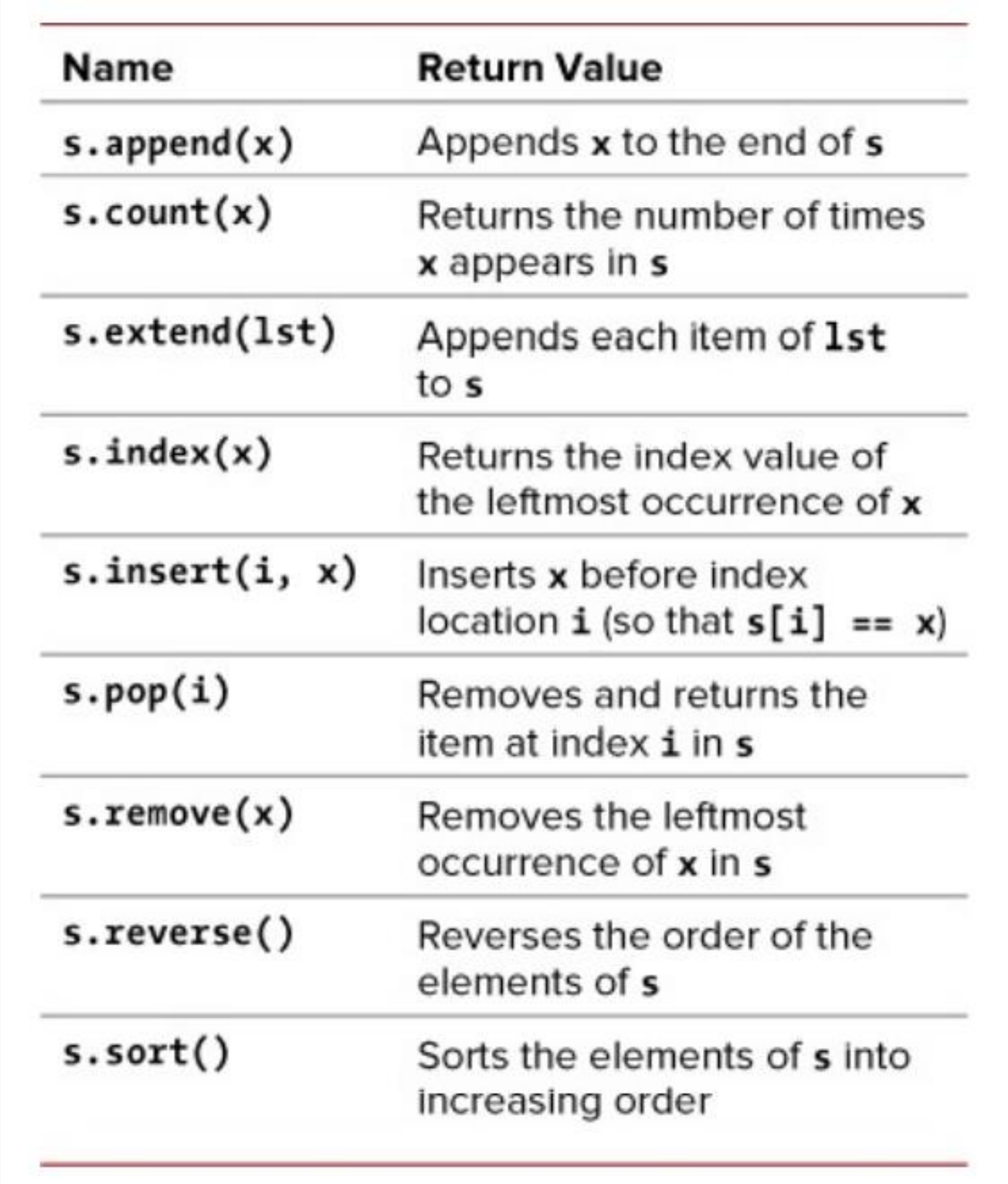
# Mutability

- The fact that lists point to their values can be the source of some surprising behavior. Consider this nasty example:

```
>>> snake = [1, 2, 3]
>>> snake[1] = snake
>>> snake
[1, […], 3]
```



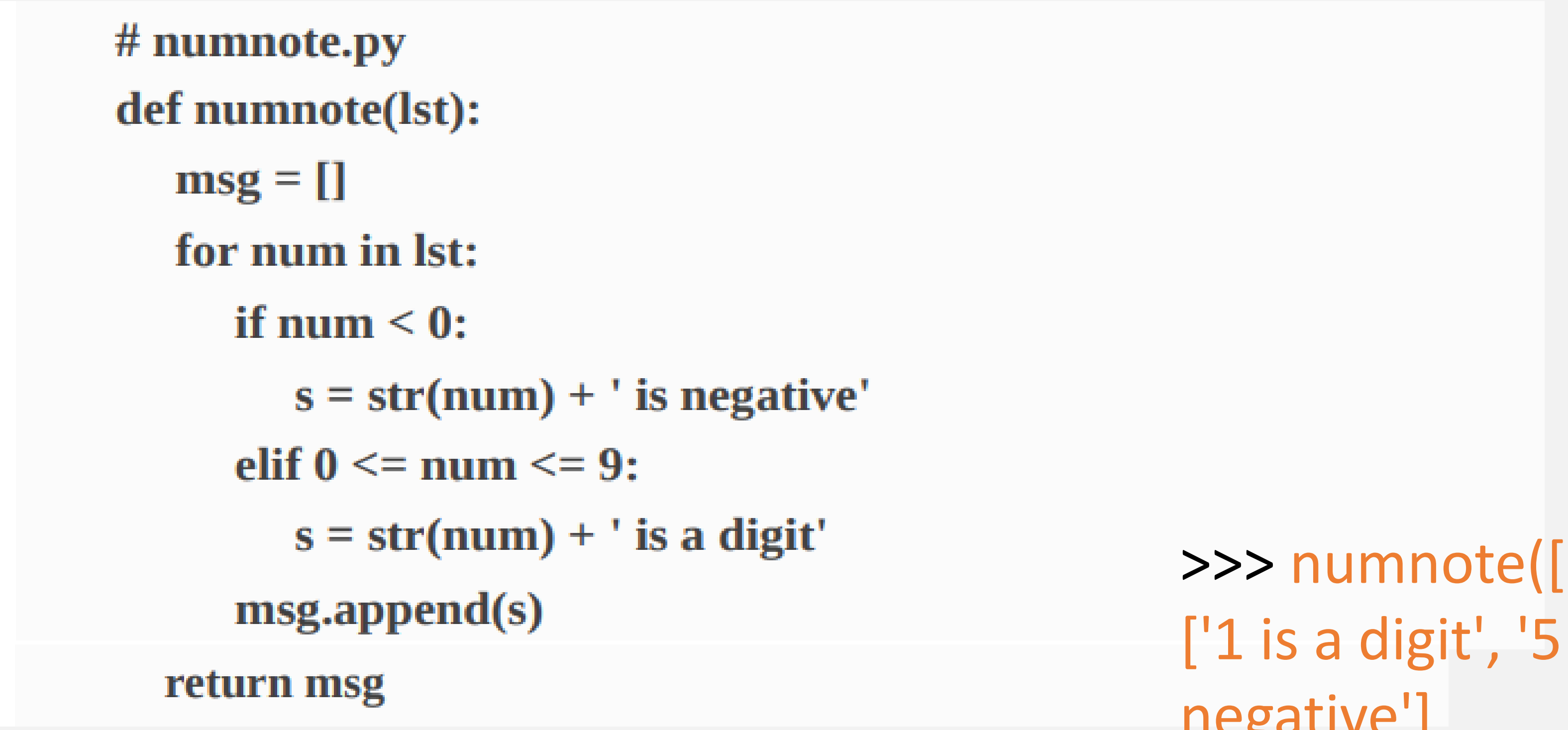
# List methods



| Name | Return Value |
|---|---|
| `s.append(x)` | Appends **x** to the end of **s** |
| `s.count(x)` | Returns the number of times **x** appears in **s** |
| `s.extend(lst)` | Appends each item of **lst** to **s** |
| `s.index(x)` | Returns the index value of the leftmost occurrence of **x** |
| `s.insert(i, x)` | Inserts **x** before index location **i** (so that `s[i] == x`) |
| `s.pop(i)` | Removes and returns the item at index **i** in **s** |
| `s.remove(x)` | Removes the leftmost occurrence of **x** in **s** |
| `s.reverse()` | Reverses the order of the elements of **s** |
| `s.sort()` | Sorts the elements of **s** into increasing order |

# List methods

- A method that creates a string of messages based on a list of input numbers:



```
# numnote.py
def numnote(lst):
    msg = []
    for num in lst:
        if num < 0:
            s = str(num) + ' is negative'
        elif 0 <= num <= 9:
            s = str(num) + ' is a digit'
        msg.append(s)
   return msg
```

```
>>> numnote([1, 5, -6, 22])
['1 is a digit', '5 is a digit’, '-6 is negative']
```

# List methods

- To print the messages on their own individual lines, you could do this:

```
>>> for msg in numnote([1, 5, -6, 22]):
        print(msg)

1 is a digit
5 is a digit
-6 is negative
```

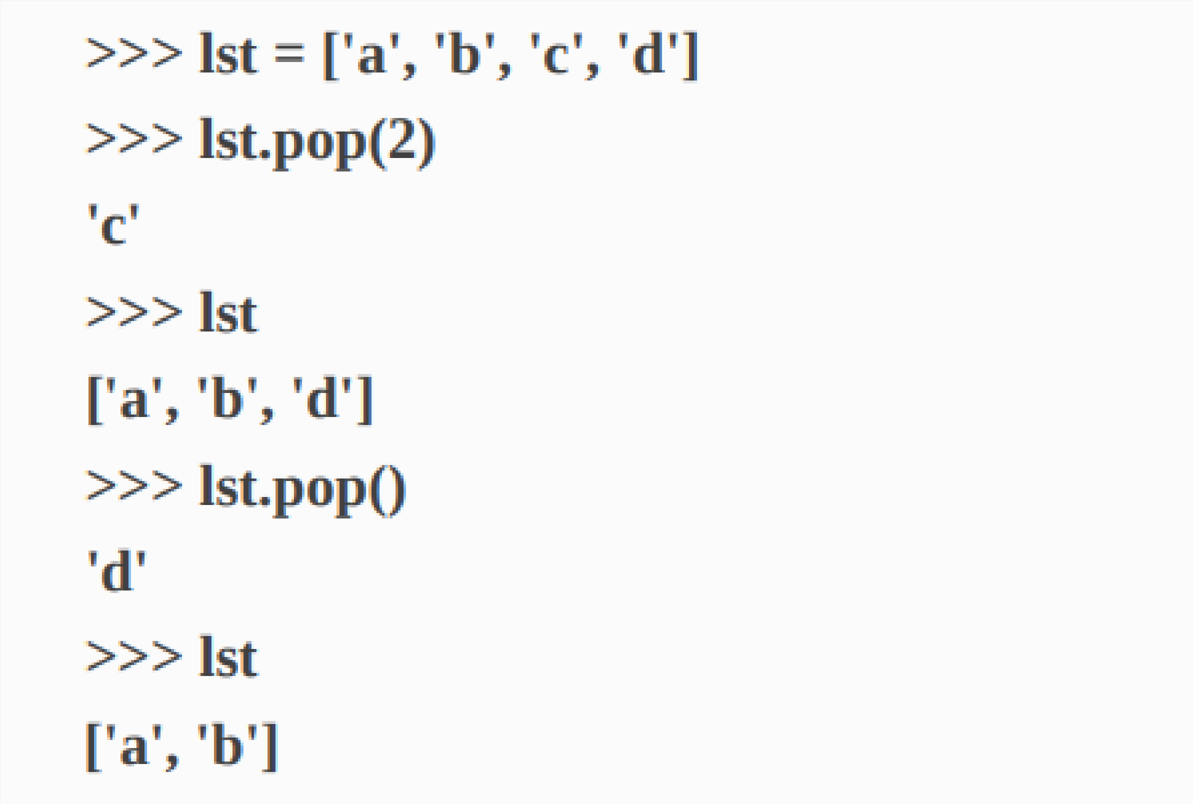
# List methods

- The extend method is similar to append, but it adds an entire sequence:

```
>>> lst = []
>>> lst.extend('cat')
>>> lst['c', 'a', 't']
>>> lst.extend([1, 5, -3])
>>> lst
['c', 'a', 't', 1, 5, -3]
```

# List methods

- The pop method removes an element at a given index position and then returns it. For example:



```
>>> lst = ['a', 'b', 'c', 'd']
>>> lst.pop(2)
'c'
>>> lst
['a', 'b', 'd']
>>> lst.pop()
'd'
>>> lst
['a', 'b']
```

# List methods

- The remove(x) method removes the first occurrence of x from a list. However, it does not return x:

```
>>> lst = ['a', 'b', 'c', 'a']
>>> lst.remove('a')
>>> lst
['b', 'c', 'a']
```

# List methods

- As the name suggests, reverse reverses the order of the elements of a list:

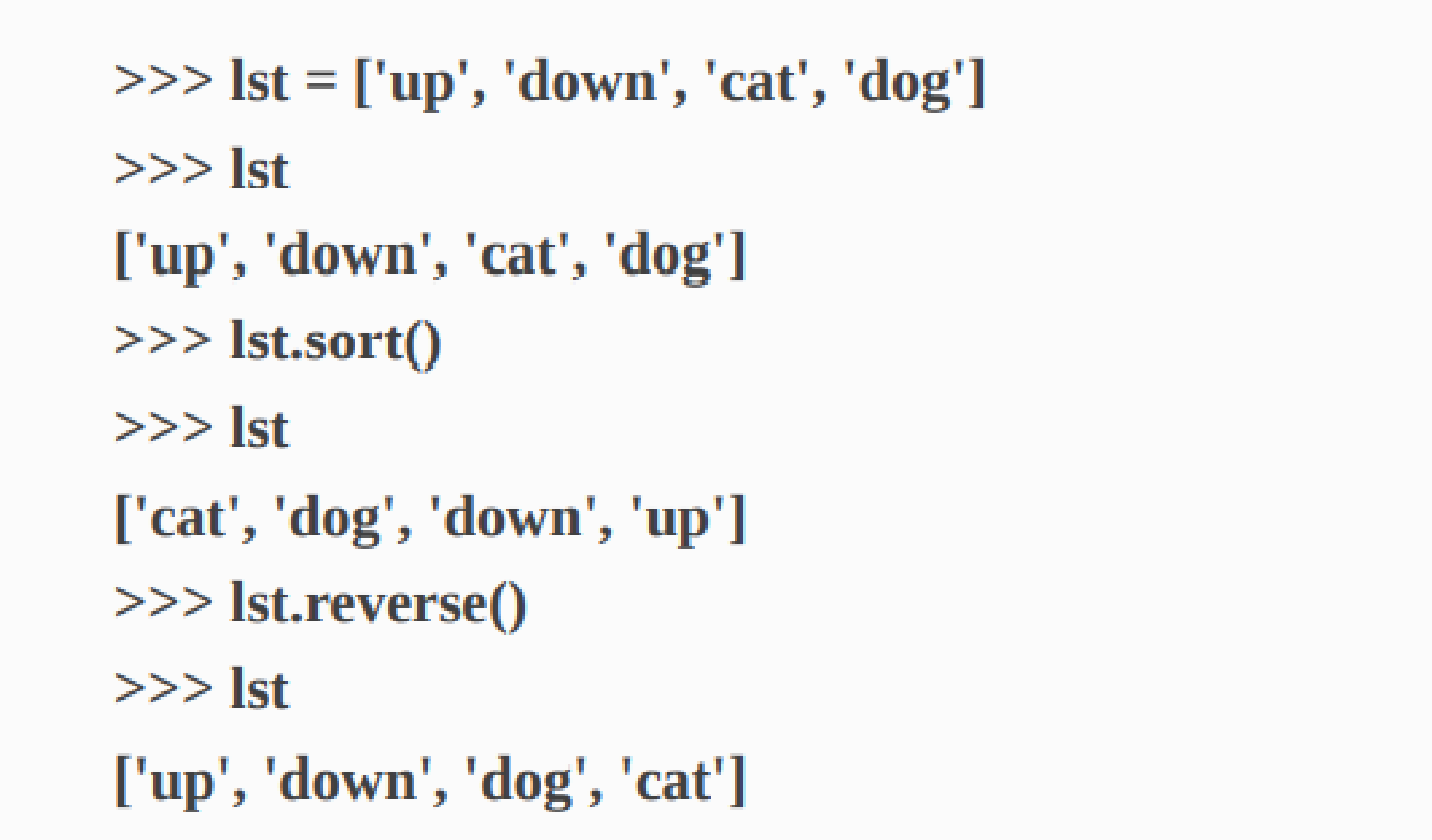
```
>>> lst = ['a', 'b', 'c', 'a']
>>> lst
['a', 'b', 'c', 'a']
>>> lst.reverse()
>>> lst
['a', 'c', 'b', 'a']
```

# Sorting Lists

- In Python, sorting is most easily done using the list sort() method.
- In practice, it can be used to quickly sort lists withtens of thousands of elements.
- Like reverse(), sort() modifies the list in place:

```
>>> lst = [6, 0, 4, 3, 2, 6]
>>> lst
[6, 0, 4, 3, 2, 6]
>>> lst.sort()
>>> lst
[0, 2, 3, 4, 6, 6]
```

# Sorting Lists



```
>>> lst = ['up', 'down', 'cat', 'dog']
>>> lst
['up', 'down', 'cat', 'dog']
>>> lst.sort()
>>> lst
['cat', 'dog', 'down', 'up']
>>> lst.reverse()
>>> lst
['up', 'down', 'dog', 'cat']
```

# Sorting Lists

- Python also knows how to sort tuples and lists. For example:

```
>>> pts = [(1, 2), (1, -1), (3, 5), (2, 1)]
>>> pts
[(1, 2), (1, -1), (3, 5), (2, 1)]
>>> pts.sort()
>>> pts
[(1, -1), (1, 2), (2, 1), (3, 5)]
```

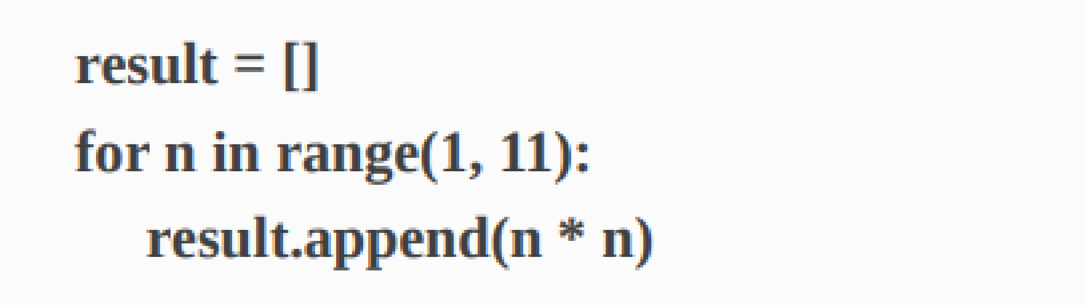
# List Comprehensions

- For example, here’s how you can use a list comprehension to create a list of the squares of the numbers from 1 to 10:

```
>>> [n * n for n in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

# List Comprehensions

- Compare this with equivalent code without a comprehension:



```
result = []
for n in range(1, 11):
    result.append(n * n)
```

# Examples of list comprehensions

- If you want to double each number on the list and 7, you can do this:

```
>>> [2 * n + 7 for n in range(1, 11)]
[9, 11, 13, 15, 17, 19, 21, 23, 25, 27]
```

# Examples of list comprehensions

- Or if you want the first ten cubes:

```
>>> [n ** 3 for n in range(1, 11)]
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

- You can also use strings in comprehensions.

For example:

```
>>> [c for c in 'pizza']
['p', 'i', 'z', 'z', 'a']
>>> [c.upper() for c in 'pizza']
['P', 'I', 'Z', 'Z', 'A']
```

# Examples of list comprehensions

- A common application of comprehensions is to modify an existing list in some way. For instance:

```
>>> names = ['al', 'mei', 'jo', 'del']
>>> names
['al', 'mei', 'jo', 'del']
>>> cap_names = [n.capitalize() for n in names]
>>> cap_names['Al', 'Mei', 'Jo', 'Del']
>>> names
['al', 'mei', 'jo', 'del']
```

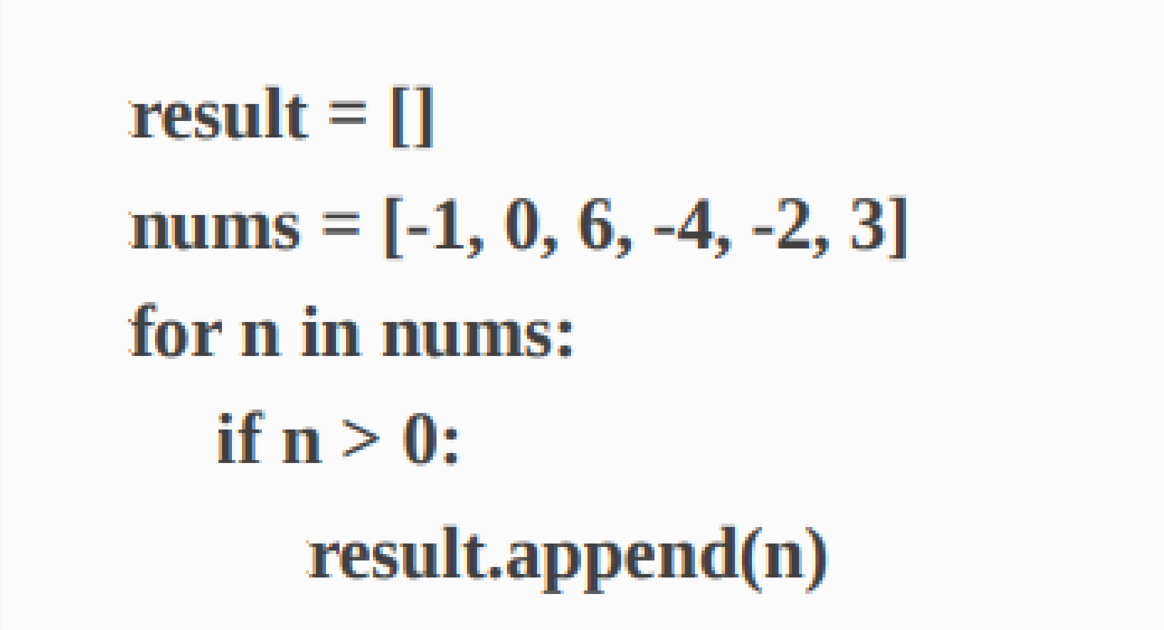
# Filtered comprehensions

- List comprehensions can also filter out elements you don’t want.
- For example, the following comprehension returns a list containing just the positive elements of nums:

```
>>> nums = [-1, 0, 6, -4, -2, 3]
>>> result = [n for n in nums if n > 0]
>>> result
[6, 3]
```

# Filtered comprehensions

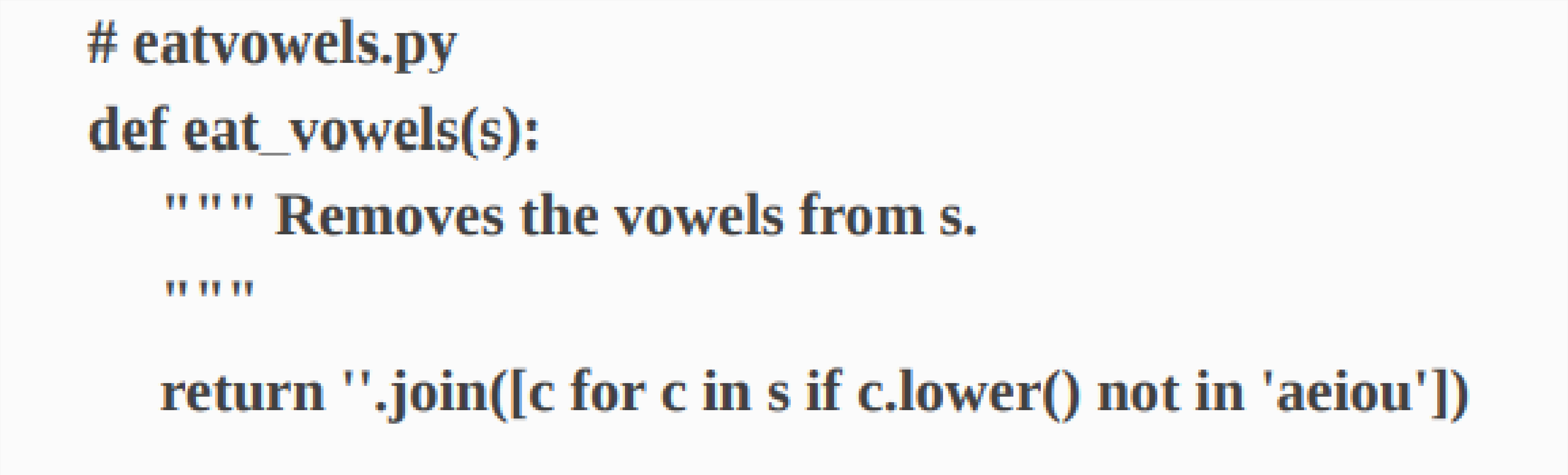
- Here’s equivalent code without a comprehension:



```
result = []
nums = [-1, 0, 6, -4, -2, 3]
for n in nums:
    if n > 0:
        result.append(n)
```

# Filtered comprehensions

- A comprehension that removes all the vowels from a word written inside a method:



```
# eatvowels.py
def eat_vowels(s):
    """ Removes the vowels from s.
    """

    return ''.join([c for c in s if c.lower() not in 'aeiou'])
```

# Filtered comprehensions

- It works like this:

```
>>> eat_vowels('Apple Sauce')
'ppl Sc’
```

- The body of eat_vowels looks rather cryptic at first, and the trick to understanding it is to read it a piece at a time.
- First, look at the comprehension:

```
[c for c in s if c.lower() not in 'aeiou']
```

# Advanced Complex Data Structures

- NumPy
- Pandas

# Python / SAS Comparison

# Code Blocks for SAS and Python

**PYTHON:**

```
numbers = [2, 4, 6, 8, 11]
product = 1
for i in numbers:
   product = product * i
print('The product is:', product)
```

**SAS:**

```
data _null_;

retain product 1;
   do i = 2 to 8 by 2, 11;
      product = product*i;
   end;

put 'The product is: ' product;
run;
```

# Indentation Matters (SAS Users often don’t like this…)

**PYTHON:**

```
numbers = [2, 4, 6, 8, 11]
product = 1
for i in numbers:
product = product * i
print('The product is:', product)
```

# Line Continuation

**PYTHON:**

```
x = 6 + \
    8 + \
   21
print('Sum of X:', x)
```

# Case Sensitivity

## PYTHON:

```
x = 201
print(X) #Cap X
```

## SAS:

```
data _null_;
        X = 201;
        put x;
Run;
```

# Demo / Use Case Python with Data Science (using Lists and Loops)