Chapter 11
Tuples

A. Downey, *Think Python: How to Think Like a Computer Science,* 3rd ed., O’Reilly, 2024.

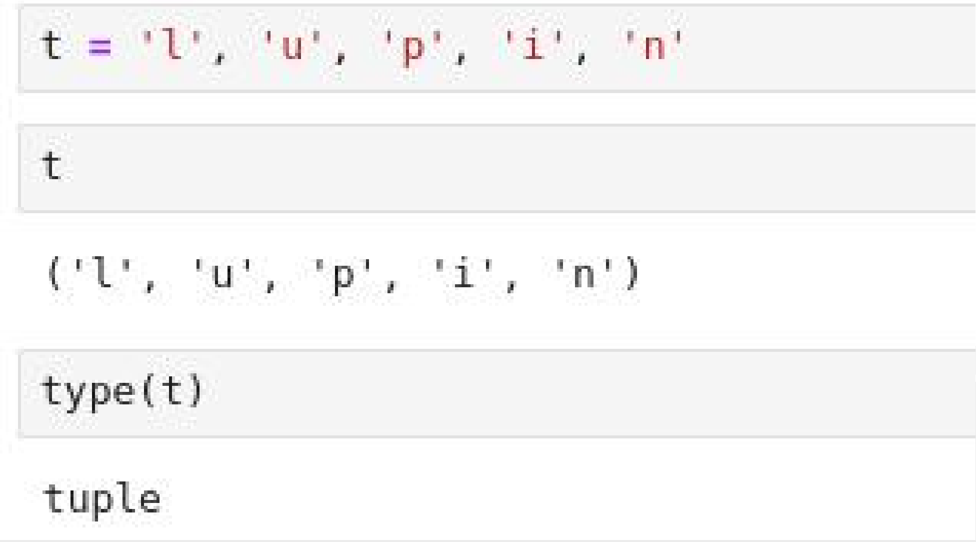
https://allendowney.github.io/ThinkPython/

Contents

- 1. Tuples Are Like Lists
- 2. But Tuples Are Immutable
- 3. Tuple Assignment
- 4. Tuples as Return Values
- 5. Argument Packing
- 6. Zip
- 7. Comparing and Sorting
- 8. Inverting a Dictionary

1. Tuples Are Like Lists

- A **tuple** is a sequence of values. The values can be any type, and they are indexed by integers, so tuples are a lot like lists.
- The important difference is that tuples are **immutable**.
- To create a tuple, you can write a comma-separated list of values
  – Although it is not necessary, it is common to enclose tuples in parentheses



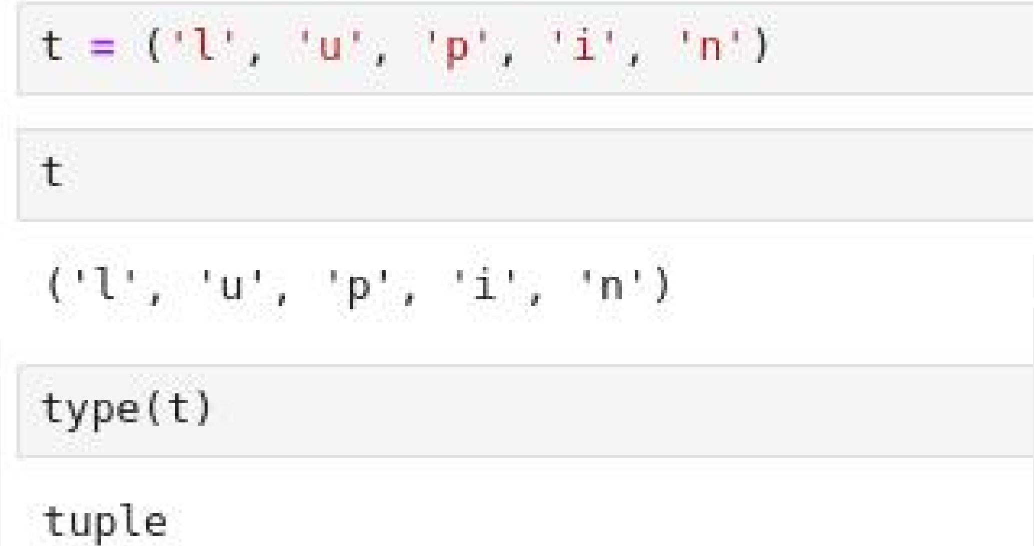
```
t = 'l', 'u', 'p', 'i', 'n'
```

```
t
```

```
('l', 'u', 'p', 'i', 'n')
```

```
type(t)
```

```
tuple
```



```
t = ('l', 'u', 'p', 'i', 'n')
```

```
t
```

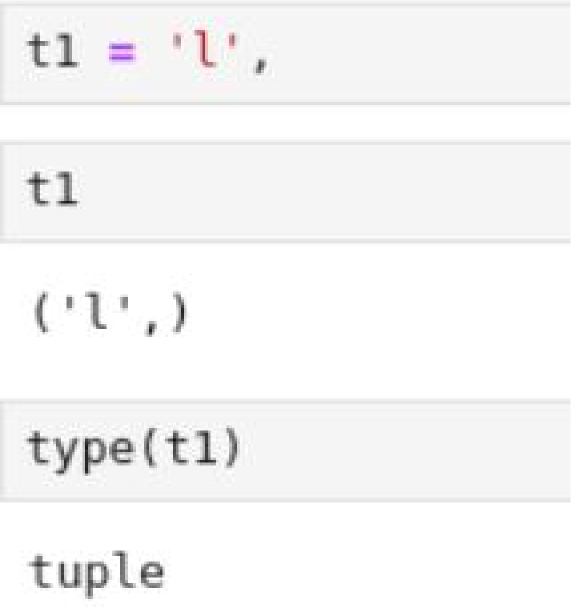
```
('l', 'u', 'p', 'i', 'n')
```

```
type(t)
```

```
tuple
```

1. Tuples Are Like Lists

- To create a tuple with a single element, you have to include a final comma, but a single value in parentheses is not a tuple:



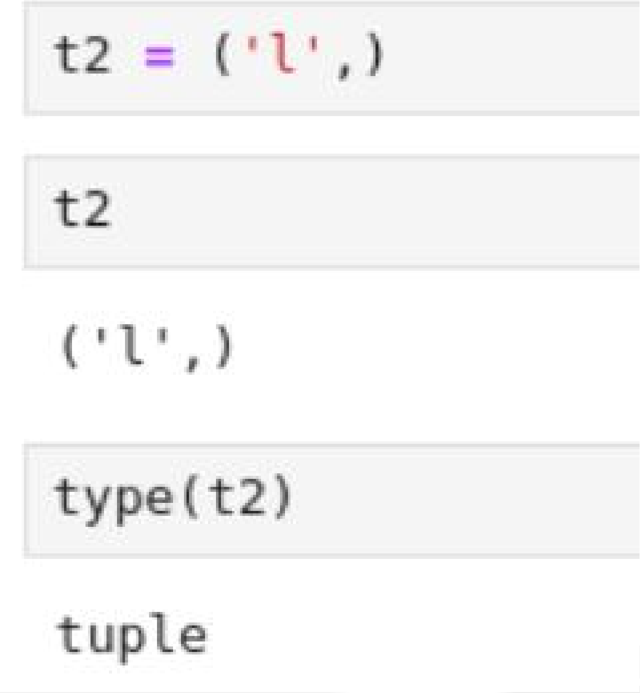
```
t1 = 'l',
```

```
t1
```

```
('l',)
```

```
type(t1)
```

```
tuple
```



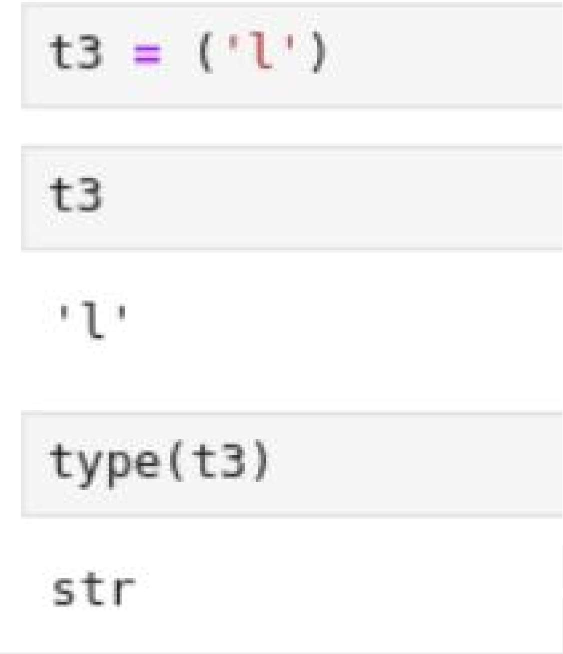
```
t2 = ('l',)
```

```
t2
```

```
('l',)
```

```
type(t2)
```

```
tuple
```



```
t3 = ('l')
```

```
t3
```

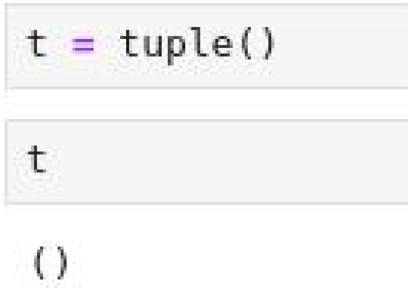
```
'l'
```

```
type(t3)
```

```
str
```

1. Tuples Are Like Lists

- Another way to create a tuple is the built-in function **tuple**. With no argument, it creates an empty tuple:

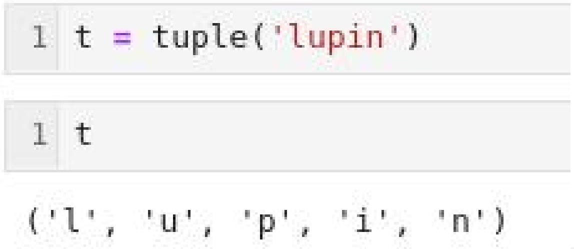


```
t = tuple()
```

```
t
```

```
()
```

- If the argument is a sequence (**string**, **list**, or **tuple**), the result is a tuple with the elements of the sequence:



```
t = tuple('lupin')
```

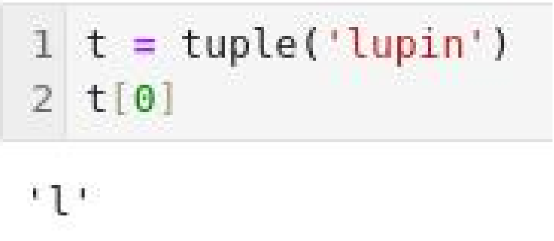
```
t
```

```
('l', 'u', 'p', 'i', 'n')
```

- Because **tuple** is the name of a built-in function, you should avoid using it as a variable name.

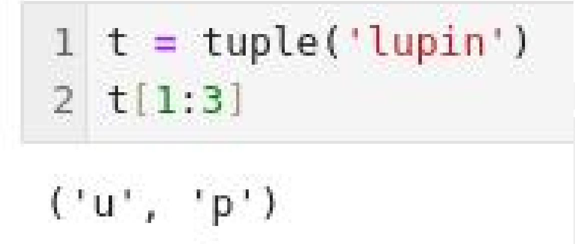
1. Tuples Are Like Lists

- Most list operators also work with tuples.
  - The bracket operator indexes an element.
  - The slice operator selects a range of elements.
  - The + operator concatenates tuples.
  - The * operator duplicates a tuple a given number of times.



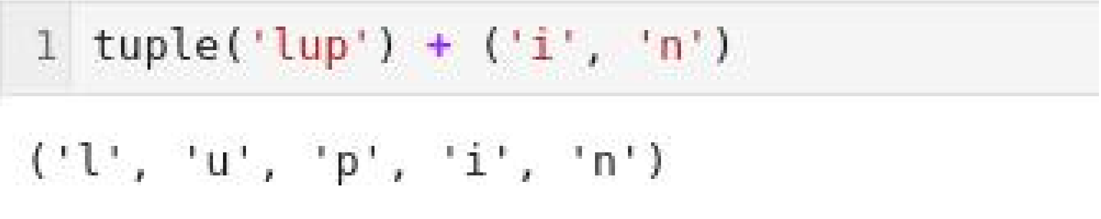
```
t = tuple('lupin')
t[0]
```

```
'l'
```



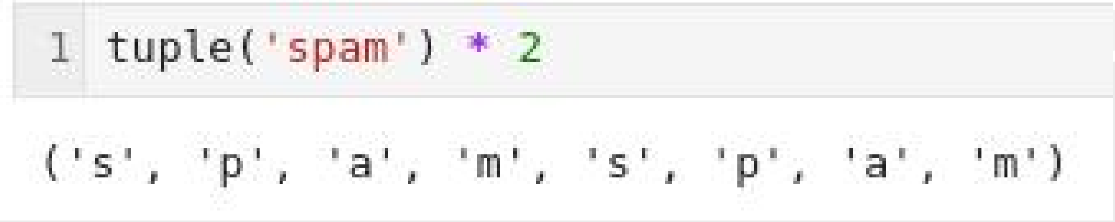
```
t = tuple('lupin')
t[1:3]
```

```
('u', 'p')
```



```
tuple('lup') + ('i', 'n')
```

```
('l', 'u', 'p', 'i', 'n')
```

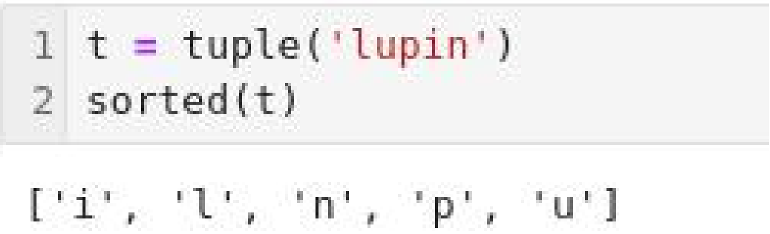


```
tuple('spam') * 2
```

```
('s', 'p', 'a', 'm', 's', 'p', 'a', 'm')
```

1. Tuples Are Like Lists

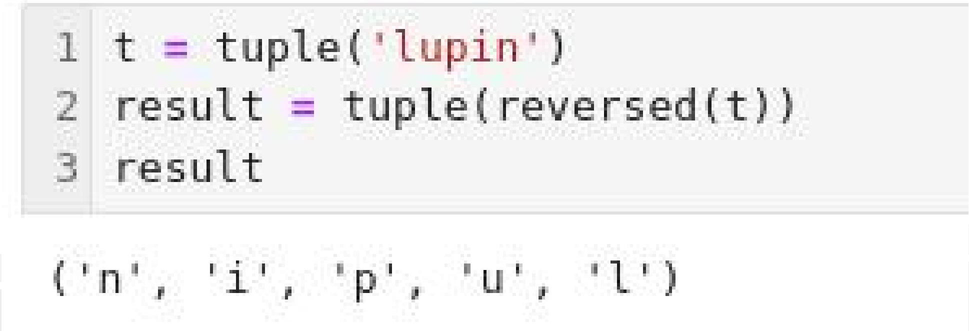
- The **sorted** function works with tuples—but the result is a list, not a tuple:



```
t = tuple('lupin')
sorted(t)
```

```
['i', 'l', 'n', 'p', 'u']
```

- The **reversed** function also works with tuples and the result is a **reversed** object, which we can convert to a list or tuple:

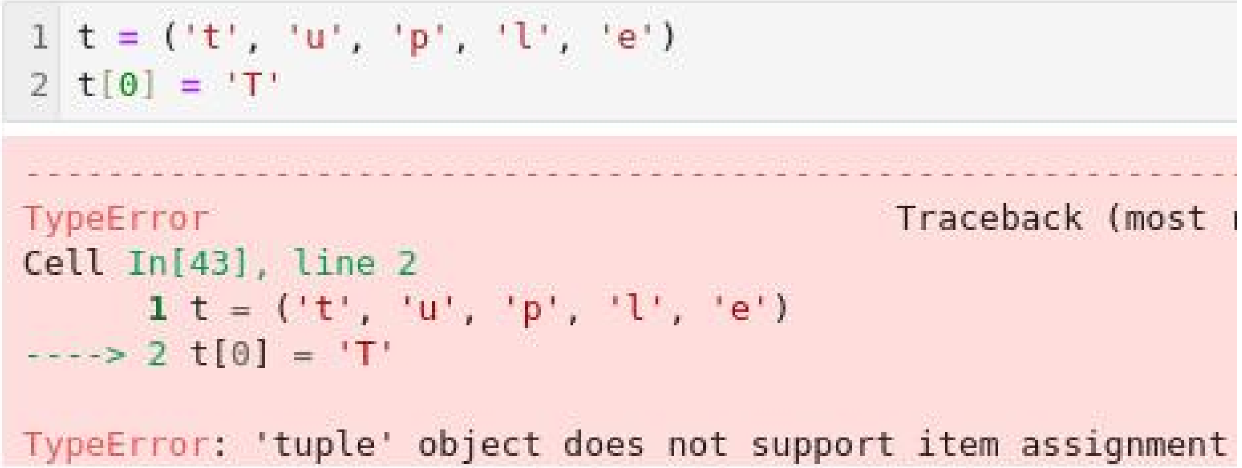


```
t = tuple('lupin')
result = tuple(reversed(t))
result
```

```
('n', 'i', 'p', 'u', 'l')
```

2. Tuples Are Immutable

- Tuples are immutable. We cannot modify a tuple.



```
t = ('t', 'u', 'p', 'l', 'e')
t[0] = 'T'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most
Cell In[43], line 2
      1 t = ('t', 'u', 'p', 'l', 'e')
----> 2 t[0] = 'T'

TypeError: 'tuple' object does not support item assignment
```

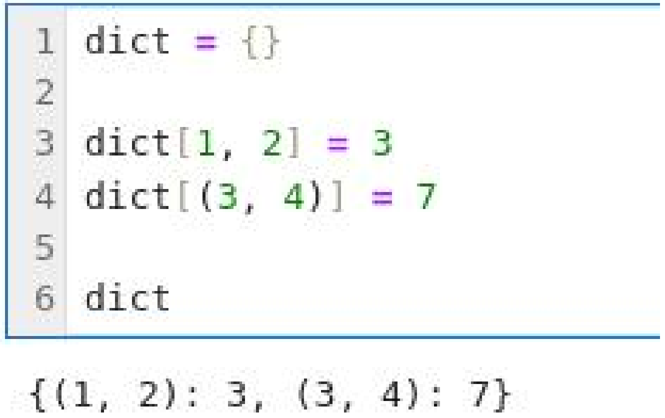
- And tuples don’t have any of the methods that modify tuples.

2. Tuples Are Immutable

- Because tuples are immutable, they are hashable, which means they can be used as keys in a dictionary.
- If a tuple contains a *mutable* value, like a list or a dictionary, the tuple is no longer hashable because it contains elements that are not hashable.

2. Tuples Are Immutable

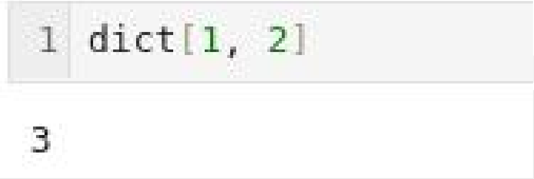
- For example, the following dictionary contains two tuples as keys that map to integers:



```python
dict = {}

dict[1, 2] = 3
dict[(3, 4)] = 7

dict
```

```
{(1, 2): 3, (3, 4): 7}
```

- We can look up a tuple in a dictionary like this:

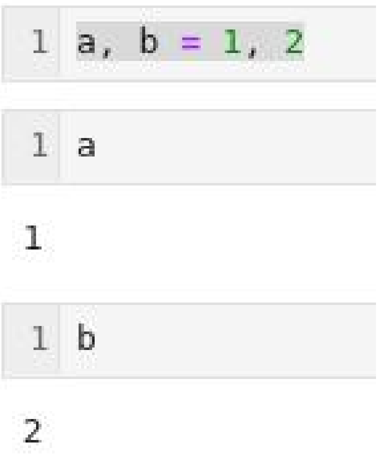


```python
dict[1, 2]
```

```
3
```

3. Tuple Assignment

- You can put a tuple of variables on the left side of an assignment, and a tuple of values on the right:



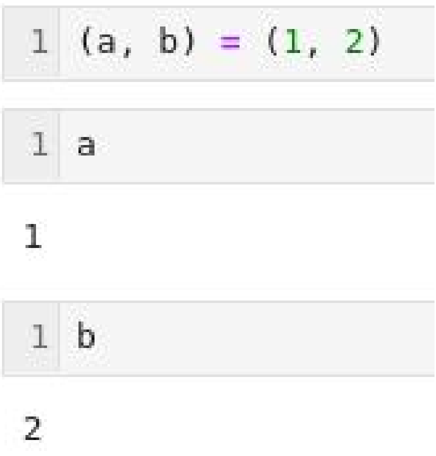
```
a, b = 1, 2
```

```
a
```

```
1
```

```
b
```

```
2
```



```
(a, b) = (1, 2)
```

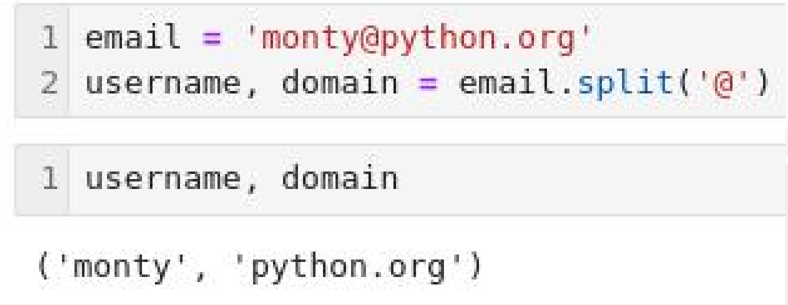
```
a
```

```
1
```

```
b
```

```
2
```

- More generally, if the left side of an assignment is a tuple, the right side can be any kind of sequence—string, list, or tuple.



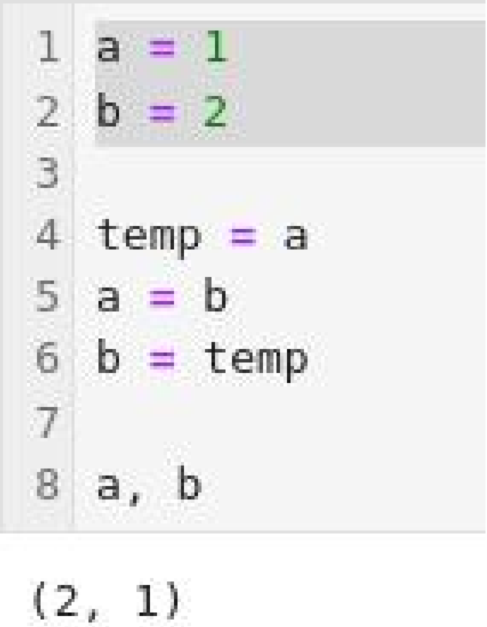
```
email = 'monty@python.org'
username, domain = email.split('@')
```

```
username, domain
```

```
('monty', 'python.org')
```

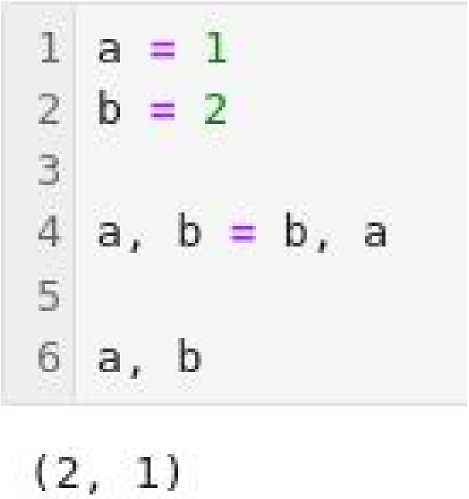
3. Tuple Assignment

- To swap the values of two variables:



```
a = 1
b = 2

temp = a
a = b
b = temp

a, b
```

```
(2, 1)
```



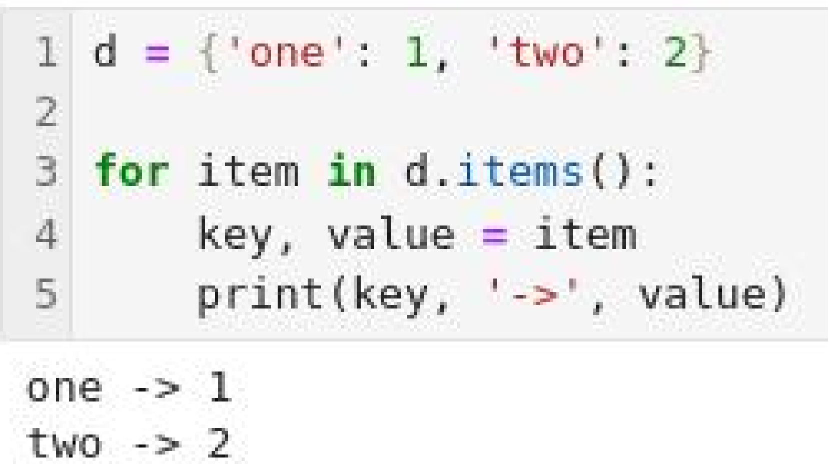
```
a = 1
b = 2

a, b = b, a

a, b
```

```
(2, 1)
```

This works because all of the expressions on the right side are evaluated before any of the assignments.

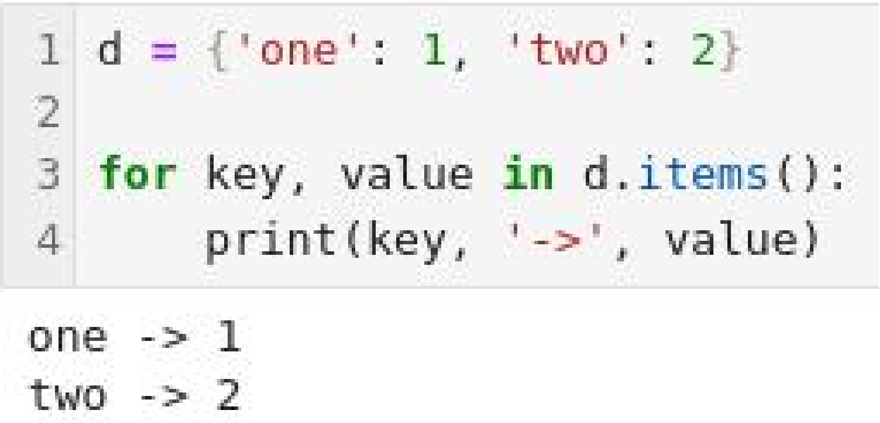
3. Tuple Assignment

- We can also use tuple assignment in a for statement. For example, to loop through the items in a dictionary, we can use the **items** method:



```
d = {'one': 1, 'two': 2}

for item in d.items():
    key, value = item
    print(key, '->', value)
```

```
one -> 1
two -> 2
```

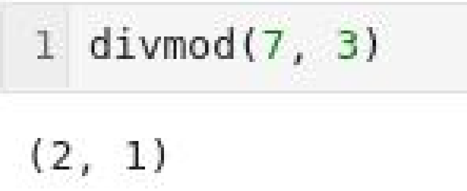


```
d = {'one': 1, 'two': 2}

for key, value in d.items():
    print(key, '->', value)
```

```
one -> 1
two -> 2
```

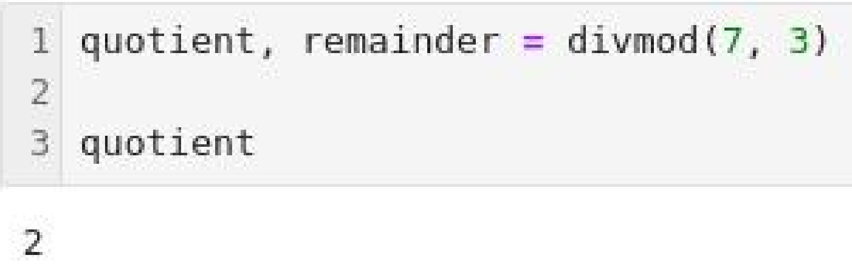
4. Tuples as Return Values

- Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning **multiple values**.
- The built-in function **divmod** takes two arguments and returns a tuple of two values, the quotient and remainder:



```
divmod(7, 3)
```

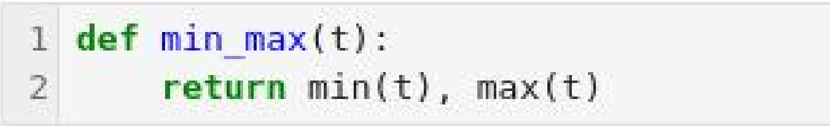
```
(2, 1)
```



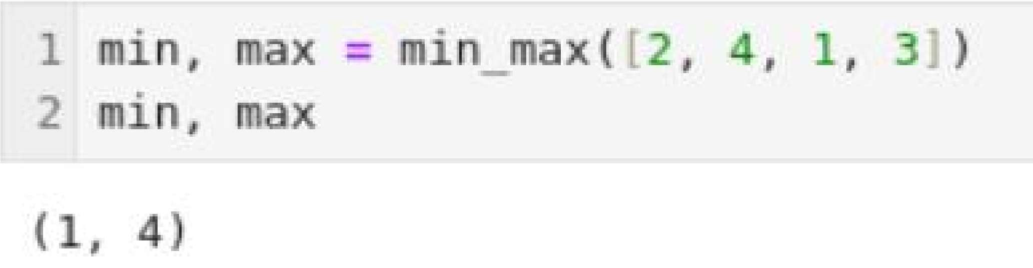
```
quotient, remainder = divmod(7, 3)

quotient
```

```
2
```

- Here is an example of a function that returns a tuple:



```
def min_max(t):
    return min(t), max(t)
```

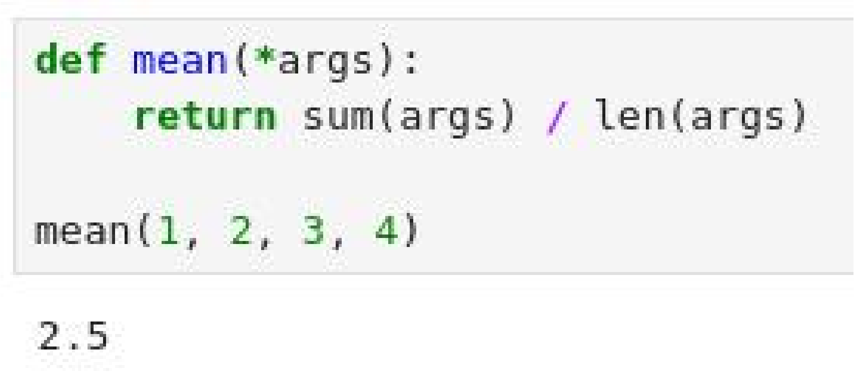


```
min, max = min_max([2, 4, 1, 3])
min, max
```

```
(1, 4)
```

5. Argument Packing

- Functions can take a variable number of arguments.
- A parameter name that begins with the * operator **packs** arguments into a tuple.
  - The parameter can have any name you like, but args is conventional.

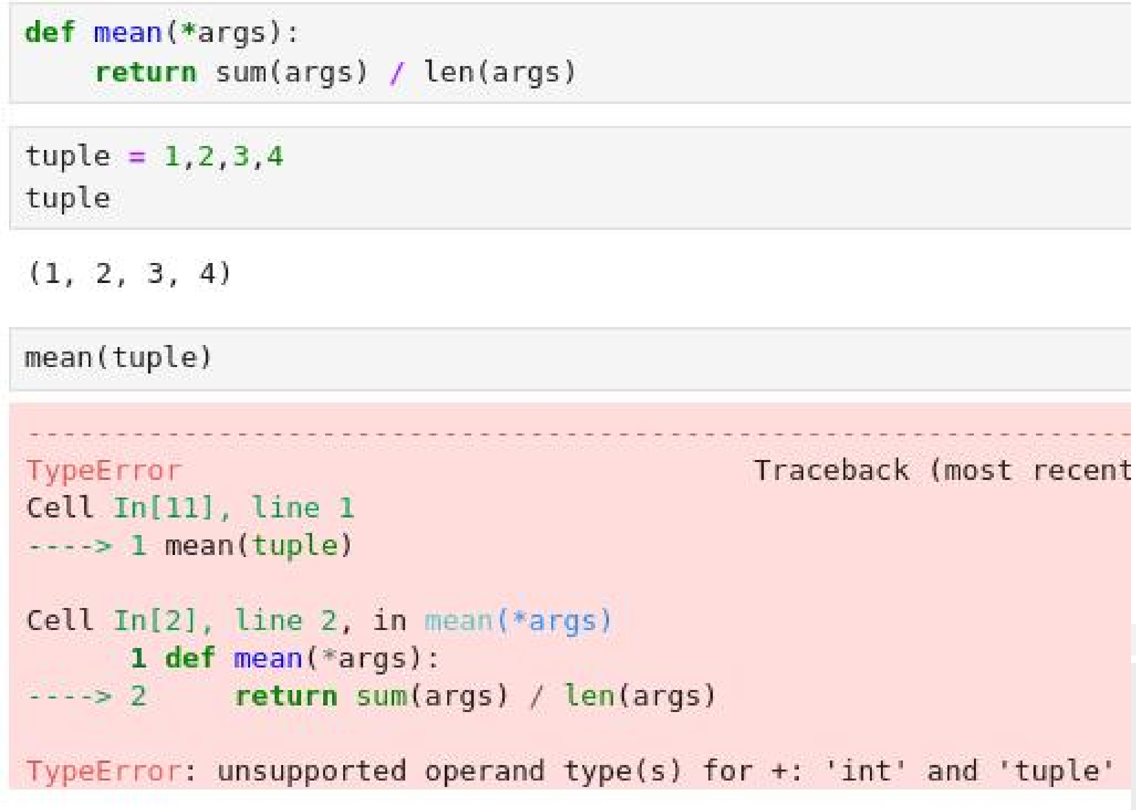


```
def mean(*args):
    return sum(args) / len(args)

mean(1, 2, 3, 4)
```

```
2.5
```

5. Argument Packing

- If you have a *sequence of values* and you want to pass them to a function as multiple arguments, you can use the * operator to **unpack** the tuple.



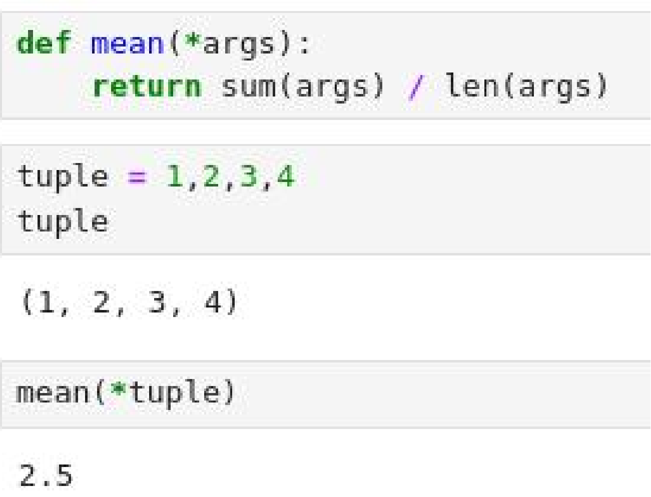
```
def mean(*args):
    return sum(args) / len(args)
```

```
tuple = 1,2,3,4
tuple
```

```
(1, 2, 3, 4)
```

```
mean(tuple)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent
Cell In[11], line 1
----> 1 mean(tuple)

Cell In[2], line 2, in mean(*args)
      1 def mean(*args):
----> 2     return sum(args) / len(args)

TypeError: unsupported operand type(s) for +: 'int' and 'tuple'
```



```
def mean(*args):
    return sum(args) / len(args)
```

```
tuple = 1,2,3,4
tuple
```

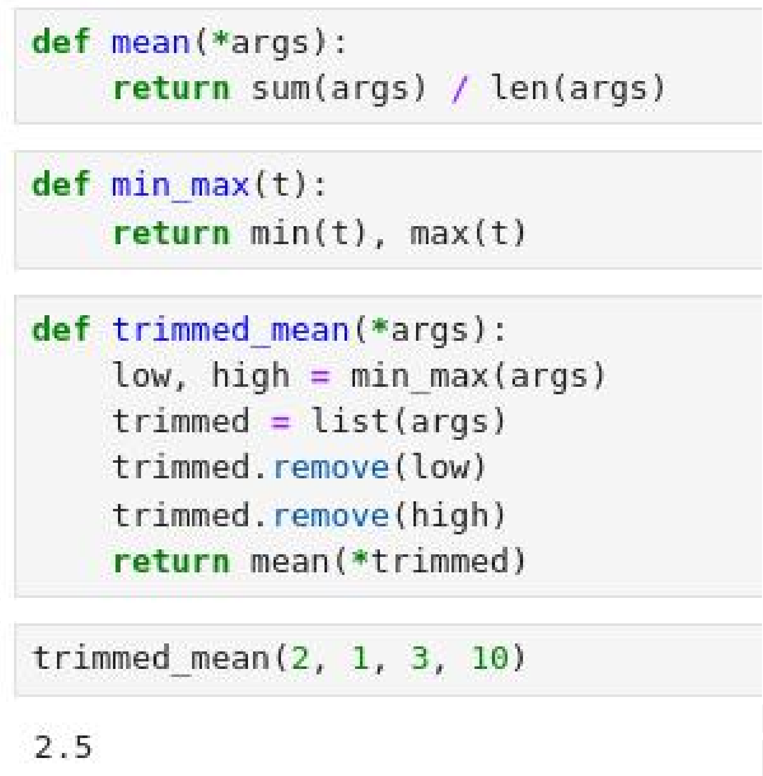
```
(1, 2, 3, 4)
```

```
mean(*tuple)
```

```
2.5
```

5. Argument Packing

- Packing and unpacking can be useful if you want to adapt the behavior of an existing function.



```python
def mean(*args):
    return sum(args) / len(args)
```

```python
def min_max(t):
    return min(t), max(t)
```

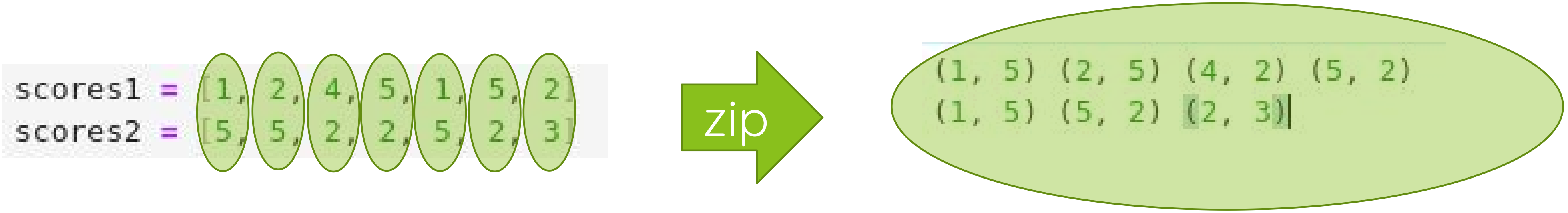
```python
def trimmed_mean(*args):
    low, high = min_max(args)
    trimmed = list(args)
    trimmed.remove(low)
    trimmed.remove(high)
    return mean(*trimmed)
```

```python
trimmed_mean(2, 1, 3, 10)
```

```
2.5
```

6. Zip

- Tuples are useful for looping through the elements of two sequences and performing operations on corresponding elements.
    - For example, suppose two teams play a series of seven games, and we record their scores in two lists, one for each team:

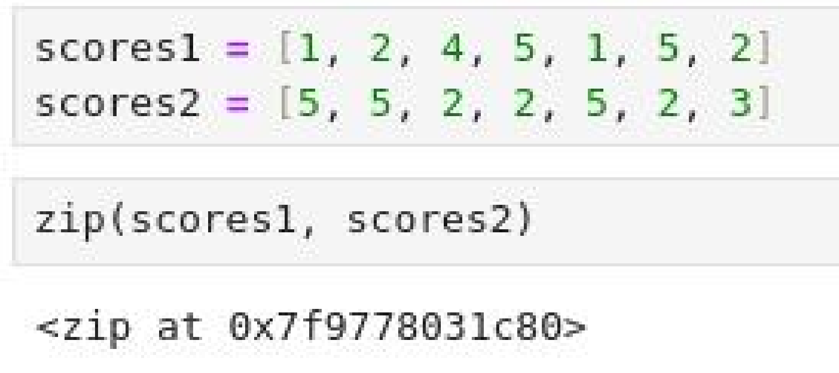


```
scores1 = [1, 2, 4, 5, 1, 5, 2]
scores2 = [5, 5, 2, 2, 5, 2, 3]
```

```
(1, 5) (2, 5) (4, 2) (5, 2)
(1, 5) (5, 2) (2, 3)
```

zip object

It pairs up the elements of the sequences like the teeth of a *zipper*.



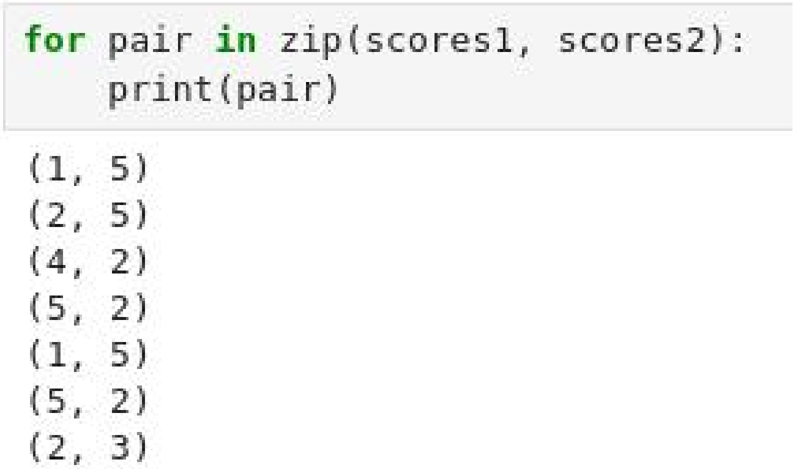
```
scores1 = [1, 2, 4, 5, 1, 5, 2]
scores2 = [5, 5, 2, 2, 5, 2, 3]
```

```
zip(scores1, scores2)
```

```
<zip at 0x7f9778031c80>
```

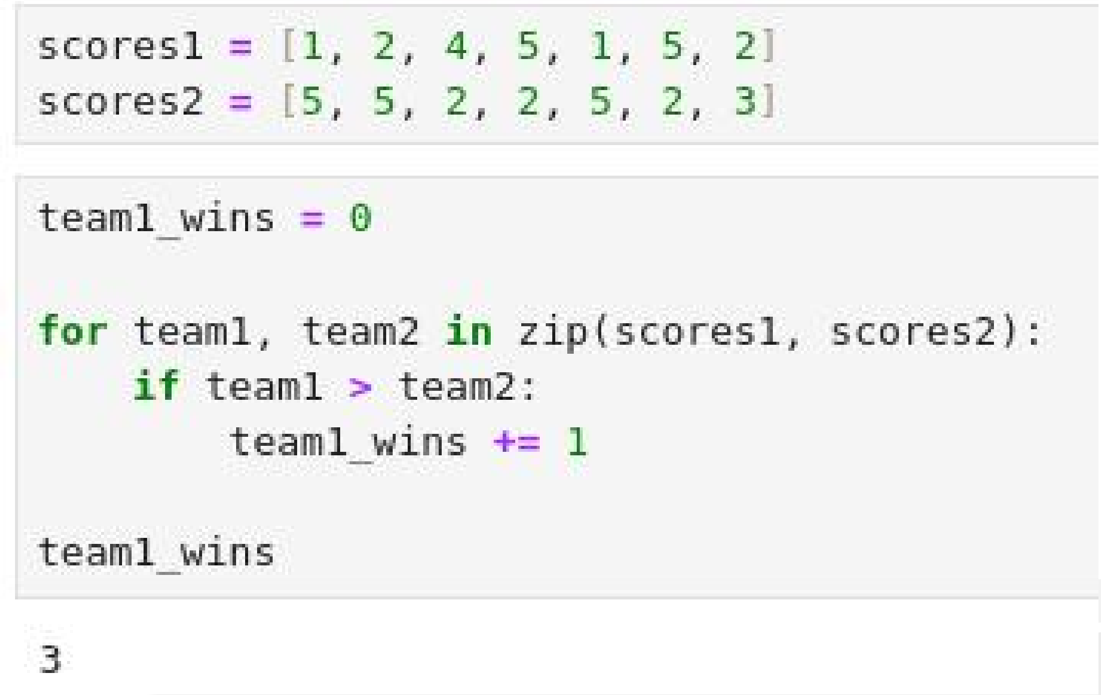
6. Zip

- We can use the `zip` object to loop through loop the values in the sequences pairwise:



```
for pair in zip(scores1, scores2):
    print(pair)
```

```
(1, 5)
(2, 5)
(4, 2)
(5, 2)
(1, 5)
(5, 2)
(2, 3)
```



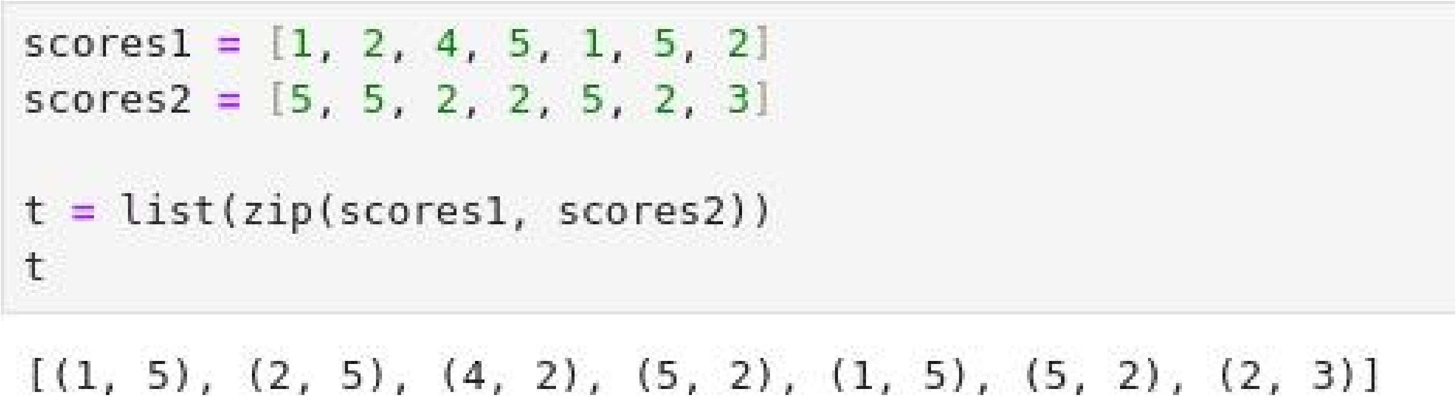
```
scores1 = [1, 2, 4, 5, 1, 5, 2]
scores2 = [5, 5, 2, 2, 5, 2, 3]
```

```
team1_wins = 0

for team1, team2 in zip(scores1, scores2):
    if team1 > team2:
        team1_wins += 1

team1_wins
```

```
3
```

6. Zip

- If you have two lists and you want a list of pairs, you can use zip and list:
  - The result is a list of tuples.

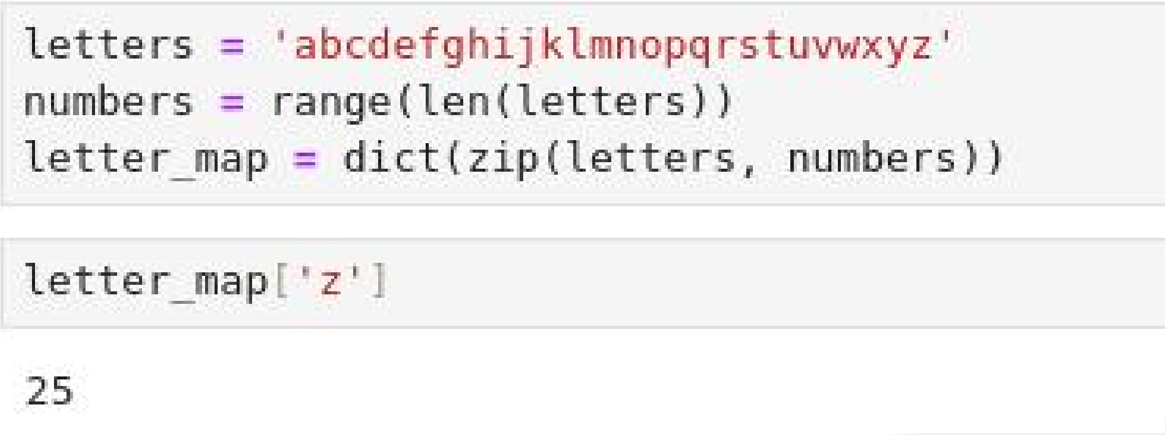


```
scores1 = [1, 2, 4, 5, 1, 5, 2]
scores2 = [5, 5, 2, 2, 5, 2, 3]

t = list(zip(scores1, scores2))
t
```

```
[(1, 5), (2, 5), (4, 2), (5, 2), (1, 5), (5, 2), (2, 3)]
```

6. Zip

- If you have a list of keys and a list of values, you can use zip and dict to make a dictionary.
  - For example, here’s how we can make a dictionary that maps from each letter to its position in the alphabet:
    - In this mapping, the index of 'a' is 0, ‘b’ is 1, ..., and the index of 'z' is 25.



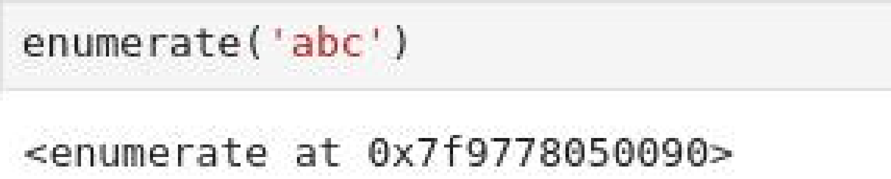
```
letters = 'abcdefghijklmnopqrstuvwxyz'
numbers = range(len(letters))
letter_map = dict(zip(letters, numbers))
```

```
letter_map['z']
```

```
25
```

6. Zip

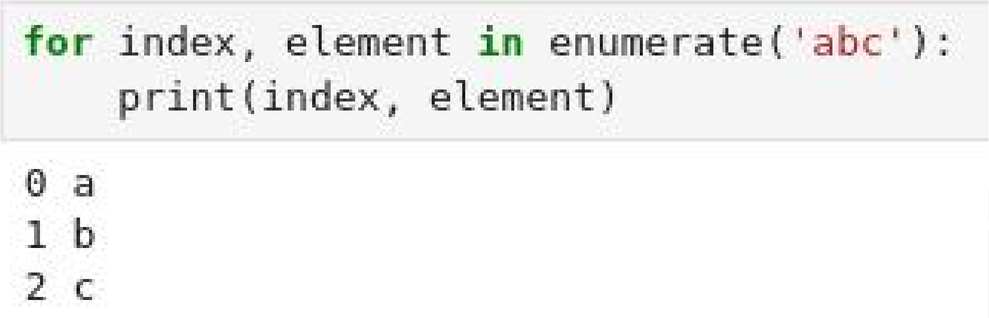
- If you need to loop through the elements of a sequence and their indices, you can use the built-in function enumerate():



```
enumerate('abc')
```

```
<enumerate at 0x7f9778050090>
```

  - The result is an enumerate object that loops through a sequence of pairs, where each pair contains an index (starting from 0) and an element from the given sequence:

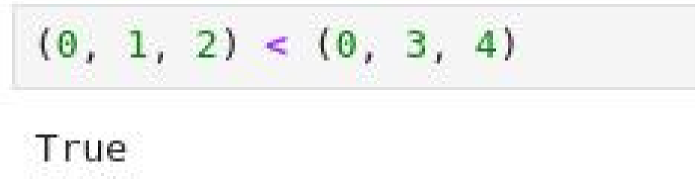


```
for index, element in enumerate('abc'):
    print(index, element)
```

```
0 a
1 b
2 c
```

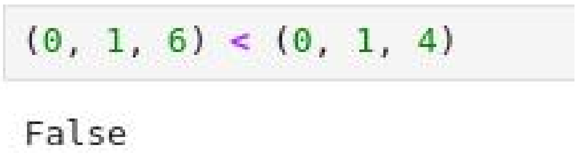
7. Comparing and Sorting

- The relational operators work with tuples and other sequences.
  - For example, if you use the < operator with tuples, it starts by comparing the first element from each sequence.
  - If they are equal, it goes on to the next pair of elements, and so on, until it finds a pair that differ:



```
(0, 1, 2) < (0, 3, 4)
```

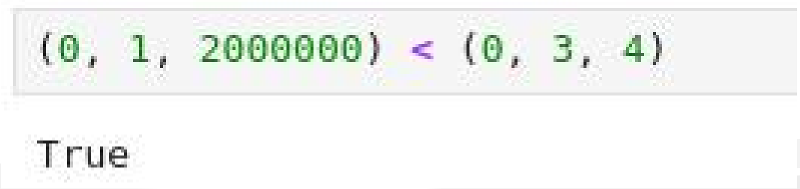
```
True
```



```
(0, 1, 6) < (0, 1, 4)
```

```
False
```

  - Subsequent elements are not considered—even if they are big:

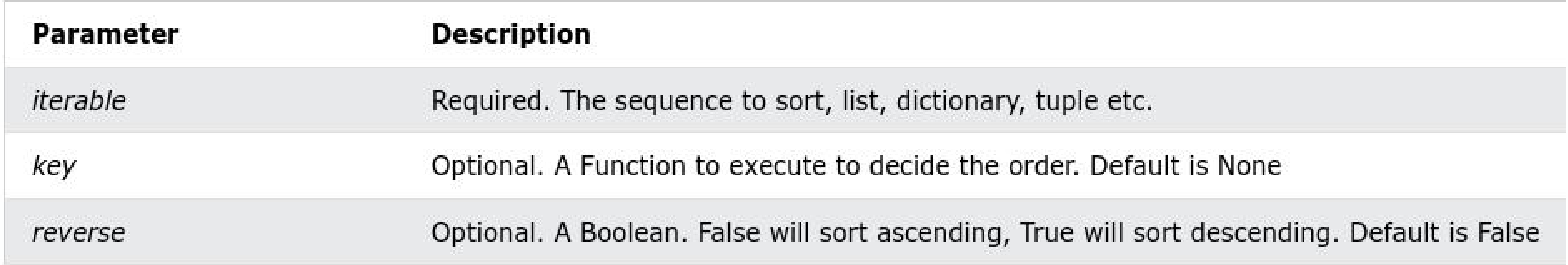


```
(0, 1, 2000000) < (0, 3, 4)
```

```
True
```

7. Comparing and Sorting

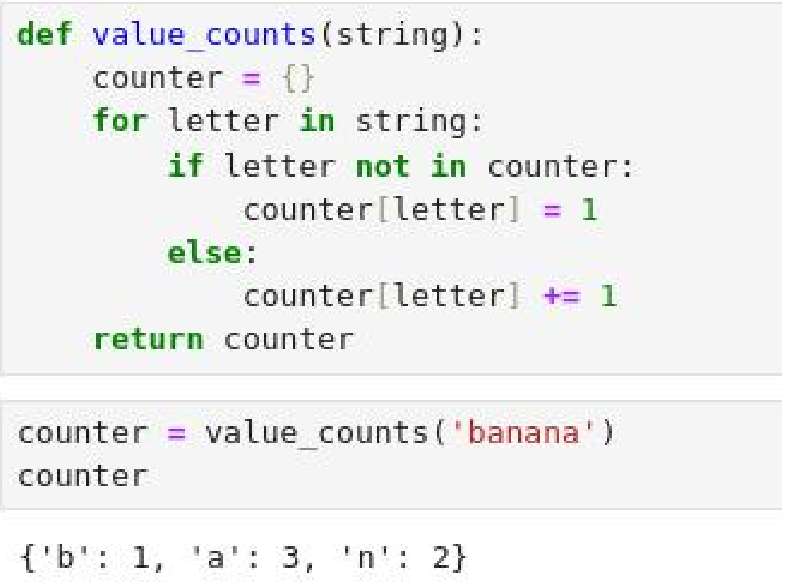
- sorted(iterable, key=*key*, reverse=*reverse*) function:



Parameter	Description
iterable	Required. The sequence to sort, list, dictionary, tuple etc.
key	Optional. A Function to execute to decide the order. Default is None
reverse	Optional. A Boolean. False will sort ascending, True will sort descending. Default is False

7. Comparing and Sorting

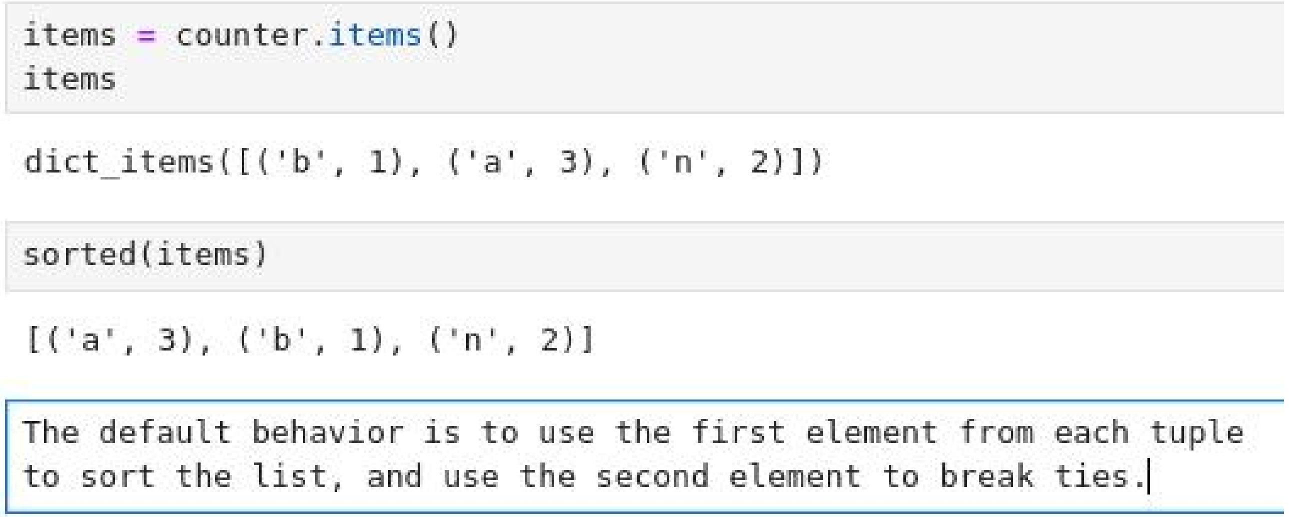
- For example: Find the most common letter in a word.



```python
def value_counts(string):
    counter = {}
    for letter in string:
        if letter not in counter:
            counter[letter] = 1
        else:
            counter[letter] += 1
    return counter
```

```python
counter = value_counts('banana')
counter
```

```
{'b': 1, 'a': 3, 'n': 2}
```



```python
items = counter.items()
items
```

```
dict_items([('b', 1), ('a', 3), ('n', 2)])
```

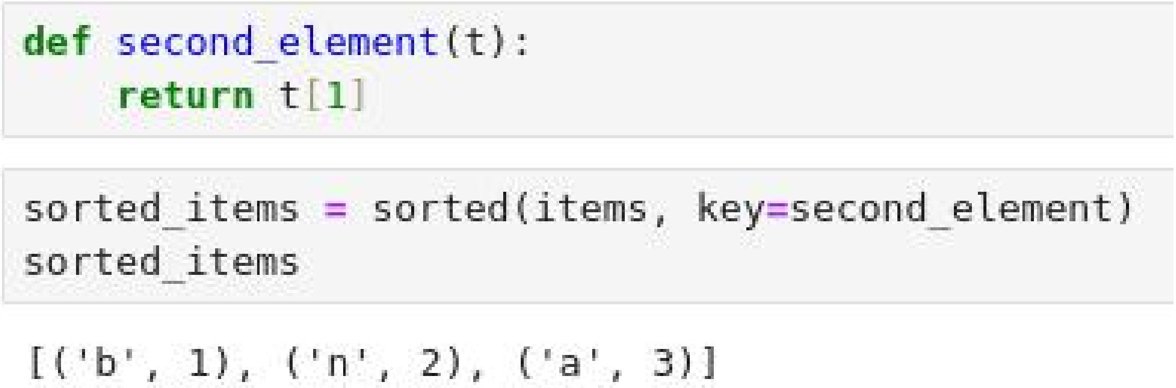
```python
sorted(items)
```

```
[('a', 3), ('b', 1), ('n', 2)]
```

The default behavior is to use the first element from each tuple to sort the list, and use the second element to break ties.

7. Comparing and Sorting

- However, to find the items with the highest counts, we want to use the second element to sort the list. We can do that by writing a function that takes a tuple and returns the second element:



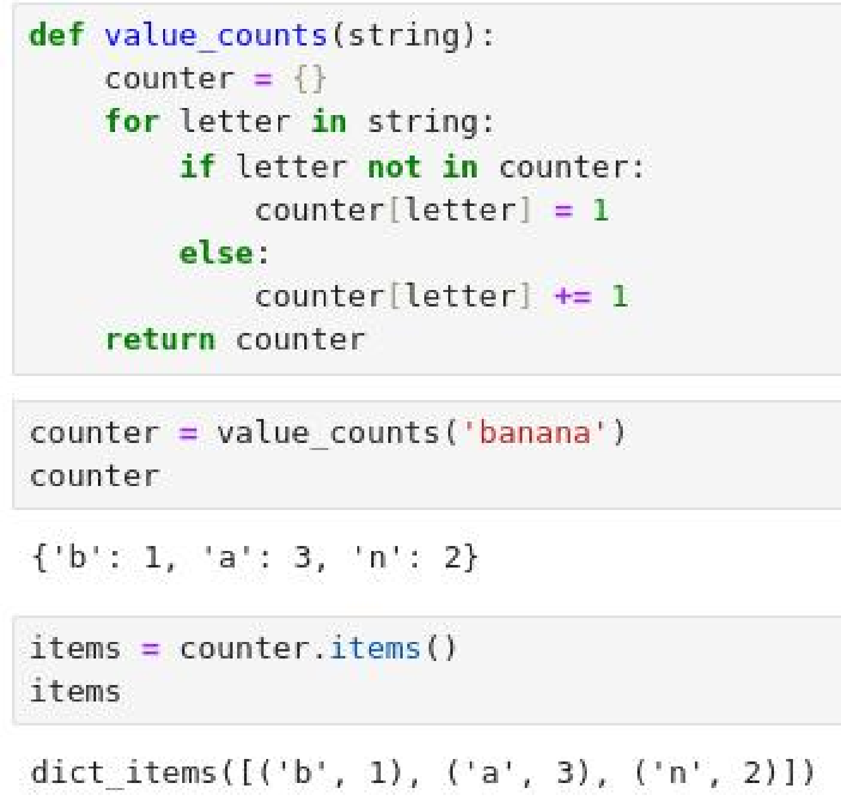
```
def second_element(t):
    return t[1]
```

```
sorted_items = sorted(items, key=second_element)
sorted_items
```

```
[('b', 1), ('n', 2), ('a', 3)]
```

7. Comparing and Sorting

- If we only want the maximum (or minimum), we don't have to sort the list. We can use max (or min), which also takes key as an optional argument:



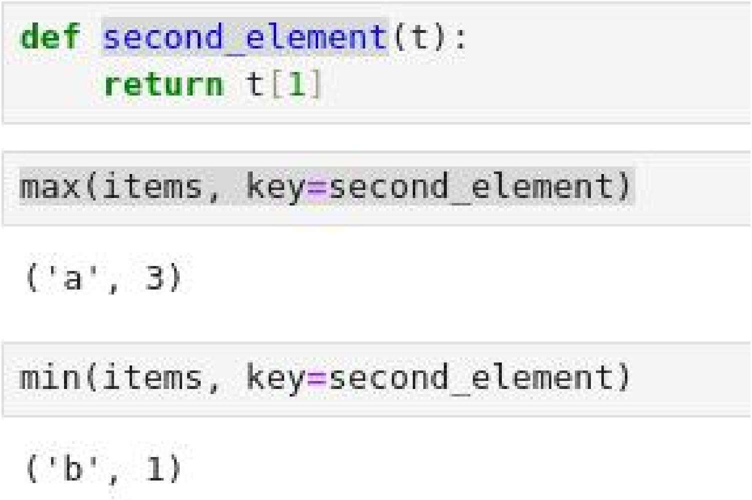
```
def value_counts(string):
    counter = {}
    for letter in string:
        if letter not in counter:
            counter[letter] = 1
        else:
            counter[letter] += 1
    return counter
```

```
counter = value_counts('banana')
counter
```

```
{'b': 1, 'a': 3, 'n': 2}
```

```
items = counter.items()
items
```

```
dict_items([('b', 1), ('a', 3), ('n', 2)])
```



```
def second_element(t):
    return t[1]
```

```
max(items, key=second_element)
```

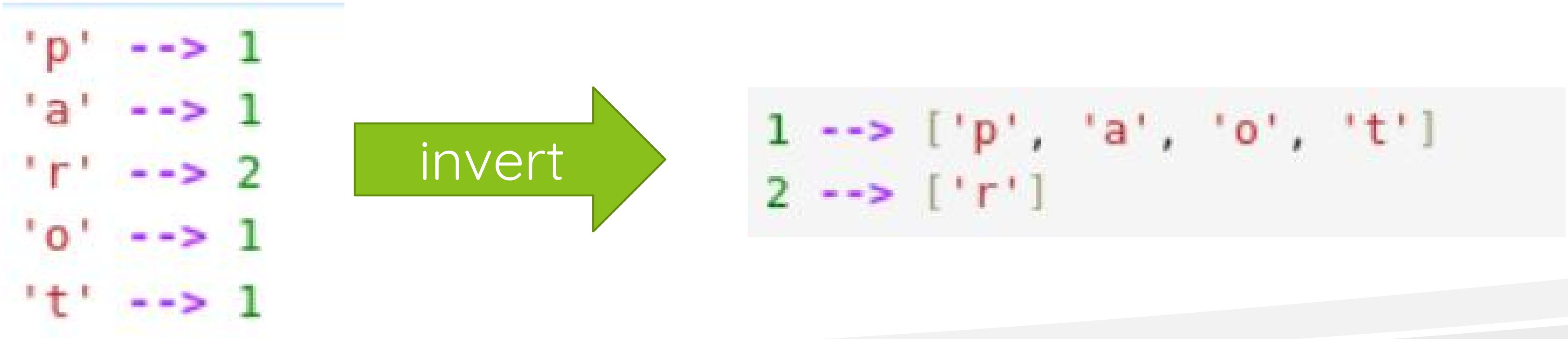
```
('a', 3)
```

```
min(items, key=second_element)
```

```
('b', 1)
```

8. Inverting a Dictionary

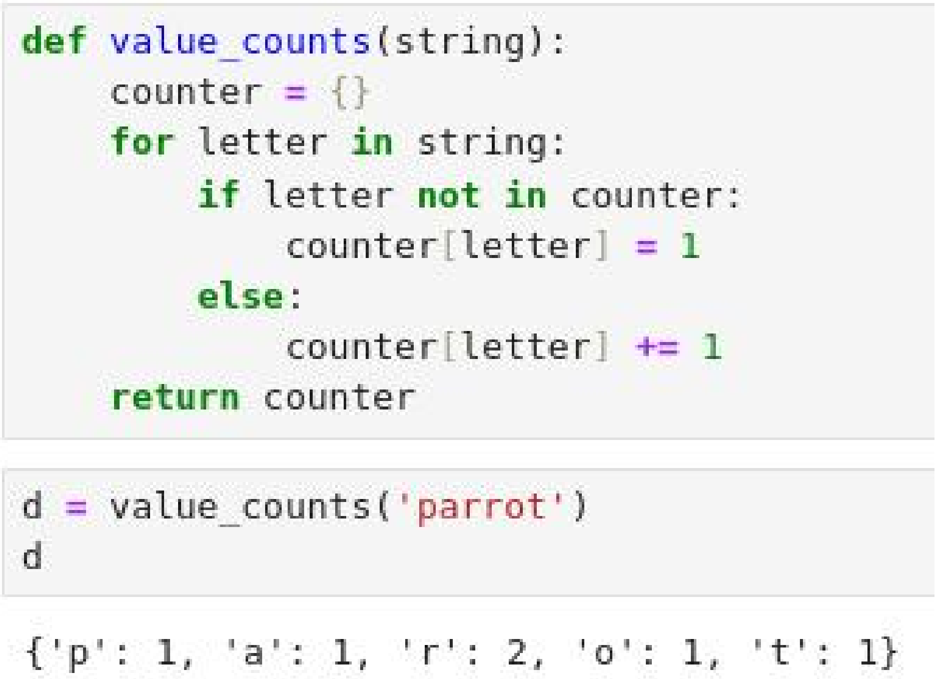
- Suppose you want to invert a dictionary so you can look up a value and get the corresponding key.
- For example, if you have a word counter that maps from each word to the number of times it appears, you could make a dictionary that maps from integers to the words that appear that number of times.



```
'p' --> 1
'a' --> 1
'r' --> 2
'o' --> 1
't' --> 1
```

```
1 --> ['p', 'a', 'o', 't']
2 --> ['r']
```

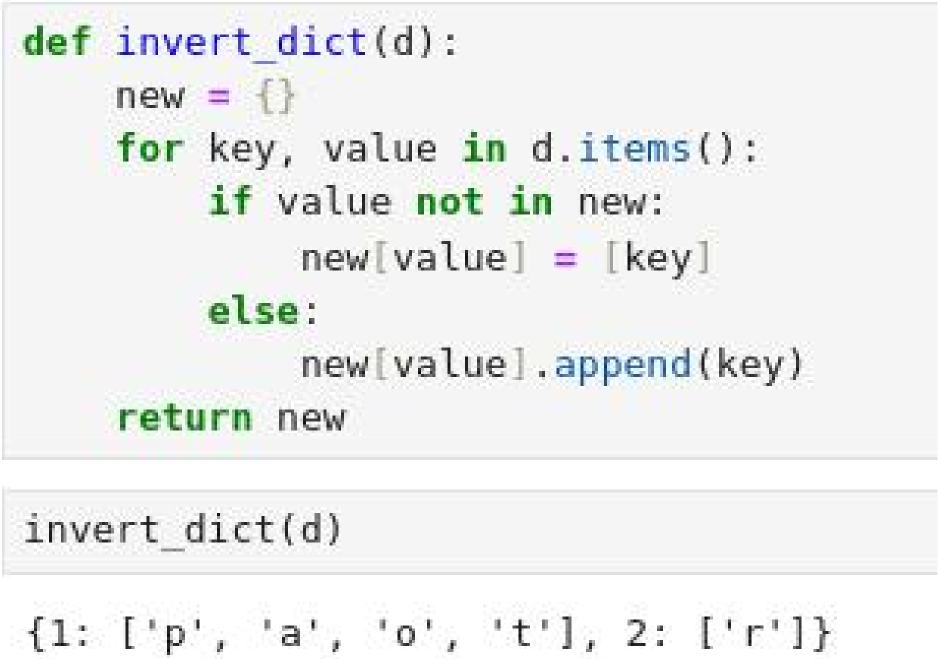
8. Inverting a Dictionary



```python
def value_counts(string):
    counter = {}
    for letter in string:
        if letter not in counter:
            counter[letter] = 1
        else:
            counter[letter] += 1
    return counter
```

```python
d = value_counts('parrot')
d
```

```
{'p': 1, 'a': 1, 'r': 2, 'o': 1, 't': 1}
```



```python
def invert_dict(d):
    new = {}
    for key, value in d.items():
        if value not in new:
            new[value] = [key]
        else:
            new[value].append(key)
    return new
```

```python
invert_dict(d)
```

```
{1: ['p', 'a', 'o', 't'], 2: ['r']}
```