Info1 Python Tutorial - Part II

Carol V. Alexandru-Funakoshi

October 3, 2023

1 Quick recap from last week

1.0.1 Functions:

- A function takes zero or more parameters and returns something.
- You can call a function by it's name and `(...)`.
- To define a new function, you use the `def` keyword and accompanying syntax.

[1]:
```
def hypothenuse(a, b):
    return (a**2 + b**2)**(1/2)

def get_first_three_characters_as_upper(word):
    return word[:3].upper()

print(hypothenuse(3, 4))
print(get_first_three_characters_as_upper("France"))
```

```
5.0
FRA
```

1.0.2 Basic numbers: `int`, `float`

[2]:
```
1 + 2
```

[2]: 3

[3]:
```
1.0 + 2
```

[3]: 3.0

[4]:
```
1.1 + 1.1 + 1.1
```

[4]: 3.3000000000000003

1.0.3 Booleans, `and`/`or` and shortcircuiting

[5]:
```
False and␣
↪some_really_expensive_function_call_that_would_take_1_hour_to_complete(123)
```

[5]: False

1.0.4 Check the type of a value using the `type(...)` function

[6]:
```
type(3.5)
```

[6]: float

[7]:
```
type(hypothenuse)
```

[7]: function

1.0.5 Imports

[8]:
```
import datetime
datetime.datetime.now()
```

[8]: datetime.datetime(2023, 10, 3, 9, 19, 8, 640384)

[9]:
```
from os.path import splitext
splitext("picture.png")
```

[9]: ('picture', '.png')

1.0.6 How to declare and operate on strings

[10]:
```
"Pineapple"[-5:].capitalize()
```

[10]: 'Apple'

1.0.7 Variables

[11]:
```
name = "Alice"
book = name + " in Wonderland"
name = "Bob"
book
```

[11]: 'Alice in Wonderland'

1.0.8 String interpolation (*f-strings*) and formatting

[12]:
```
f'{name} is reading "{book}"'
```

[12]: 'Bob is reading "Alice in Wonderland"'

[13]:
```
from math import pi
f"Pi to the first 4 digits is {pi:.4f}"
```

[13]: 'Pi to the first 4 digits is 3.1416'

```
[14]: digits = 6
      f"Pi to the first {digits} digits is {pi:.{digits}f}"
```

[14]: 'Pi to the first 6 digits is 3.141593'

1.0.9 Tuples and Lists

```
[15]: stuff = (1, "hello", 2)
      print(stuff[1])
      print(stuff[-1])
```

```
hello
2
```

```
[16]: stuff = [1, "hello", 2]
      print(stuff[1])
      del(stuff[2])
      stuff.append(3)
      stuff.extend([4, 5])
      print(stuff)
```

```
hello
[1, 'hello', 3, 4, 5]
```

1.0.10 Sets

```
[17]: my_set = {1, 2, 3, 4, 4, 4, 4, "hello"}
      print(my_set)
      my_set.remove(3)
      my_set.add(4)
      my_set.add(10)
      print(my_set)
```

```
{1, 2, 3, 4, 'hello'}
{1, 2, 4, 'hello', 10}
```

1.0.11 Mutable vs. Immutable

- lists are *mutable*
- tuples and strings are *immutable*

```
[18]: mutable_list = [1, 2, 3, 3, 4]
      mutable_list.remove(3)
      print(list)
      mutable_list.append(100)
      mutable_list.extend([-1, -2, "hello"])
      mutable_list
```

```
<class 'list'>
```

[18]:
```
[1, 2, 3, 4, 100, -1, -2, 'hello']
```

[19]:
```python
immutable_tuple = (1, 2, 3, 4)
# not possible:
# immutable_tuple.append(5)
changed_tuple = immutable_tuple + (5,)
changed_tuple
```

[19]:
```
(1, 2, 3, 4, 5)
```

2 Useful built-in functionality

2.0.1 range

`range` is a collection that represents sequences of numbers (for example, `range(10)` represents `0` through `9`). Note that just calling range will not really *do* much. You just get a ``range'' object.

[20]:
```python
range(10)
```

[20]:
```
range(0, 10)
```

[21]:
```python
type(range(10))
```

[21]:
```
range
```

Only when you *need* the numbers will they be produced. For example, if we want a list of numbers:

[22]:
```python
list(range(10))
```

[22]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This is done because only the necessary number of elements will be produced one by one. This is called *lazy evaluation*.

[23]:
```python
super_big_range = range(10000000000000000000000000000000000000000000)
super_big_range[5:20]
```

[23]:
```
range(5, 20)
```

[24]:
```python
list(super_big_range[5:20])
```

[24]:
```
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

`range` also takes additional parameters to determine the start, end, step size, etc. See https://docs.python.org/3/library/stdtypes.html#typesseq-range

[25]:
```python
set(range(5, 20, 2))
```

[25]: {5, 7, 9, 11, 13, 15, 17, 19}

[26]:
```python
tuple(range(21, 0, -3))
```

[26]: (21, 18, 15, 12, 9, 6, 3)

(don't convert ranges to lists, tuples or whatever unless necessary)

2.0.2 enumerate

Use `enumerate` on a collection to create a sequence of tuples, where each tuple has two elements: * an index * an element from the collection

[27]:
```python
list(enumerate(["These", "are", "words", "in", "a", "list"]))
```

[27]: [(0, 'These'), (1, 'are'), (2, 'words'), (3, 'in'), (4, 'a'), (5, 'list')]

Just like `range`, `enumerate` on it's own doesn't *do* much and only evaluates when needed:

[28]:
```python
enumerate(["These", "are", "words", "in", "a", "list"])
```

[28]: <enumerate at 0x7f93d5549490>

[29]:
```python
enumerate(super_big_range)
```

[29]: <enumerate at 0x7f93d5549c10>

Note that `enumerate` also works on collections that don't really have an inherent ordering, like sets:

[30]:
```python
list(enumerate({1, 100, 40, 88}))
```

[30]: [(0, 40), (1, 1), (2, 88), (3, 100)]

A few more examples:

[31]:
```python
list(enumerate(range(5)))
```

[31]: [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]

[32]:
```python
tuple(enumerate("this is a sentence".split()))
```

[32]: ((0, 'this'), (1, 'is'), (2, 'a'), (3, 'sentence'))

`enumerate` also takes an additional parameter to change the starting index:

[33]:
```python
list(enumerate([10, 333, 30], 250))
```

[33]: [(250, 10), (251, 333), (252, 30)]

2.0.3 `sum`, `min`, and `max`

These functions can be used on collections of numbers to calculate their sum, minimum value or maximum value, respectively:

[34]:
```
numbers = [12, 41, 2024, 3, -301, 0]
sum(numbers)
```

[34]: 1779

[35]:
```
min(numbers)
```

[35]: -301

[36]:
```
max(numbers)
```

[36]: 2024

Exercise

1. Write an expression that calculates the sum of all numbers between 5 and 200.
2. Write an expression that creates a list of tuples, given a list of words `words`. Each tuple should contain an index and a word. The index for the first word should be `1`

[37]:
```
# exercise 1 solution
```

[38]:
```
# exercise 2 solution
words = "The original machine had a base-plate of prefabulated aluminite".
↪split()
```

3 Conditional statements for control flow: `if`, `elif`, and `else`

Oftentimes, you want your program behavior to vary depending on what data it receives. One mechanism to do so is *control flow* by means of `if`, `elif`, and `else` to decide whether to execute a given block of code.

[39]:
```
number = 10
if number > 0:
    print("Greater than zero")
else:
    print("Not greater than zero")
```

```
Greater than zero
```

The following variations of using if/elif/else are permitted: * Just an `if` block, alone. The code block gets executed if the condition is true, otherwise not. Here are two examples:

[40]:
```
if 1 + 1 == 2:
    print("Math is easy")
```

```python
if True == False:
    print("The universe is broken")
```

```
Math is easy
```

- An `if` block followed by an `else` block. At least **one** of those two blocks will **always** be executed (unless the program crashes…), but **not the other**:

[41]:
```python
if True == False:
    print("The universe is broken")
else:
    print("Everything is OK")
```

```
Everything is OK
```

- An `if` block followed by any number of `elif` blocks, which just add more possible conditions and code blocks. Only the **first** block with a `True` condition will execute. If none of the conditions are true, none of the code blocks are executed:

[42]:
```python
number = 5
if number > 0:
    print("Number is greater than 0")
elif number > 3:
    print("Number is greater than 3")
```

```
Number is greater than 0
```

If you wanted both blocks to execute, you could write the following. Note that these two `if` statements are entirely separate and have nothing to do with each other.

[43]:
```python
number = 5
if number > 0:
    print("Number is greater than 0")
if number > 3:
    print("Number is greater than 3")
```

```
Number is greater than 0
Number is greater than 3
```

- An `if` block followed by zero or more `elif` blocks, followed by a final `else` block. Exactly **one** of these blocks will execute.

[44]:
```python
number = -5
if number > 0:
    print("Number is greater than 0")
elif number > 3:
    print("Number is greater than 3")
elif number > 5:
    print("Number is greater than 5")
else:
    print("None of the above")
```

```
None of the above
```

Of course, this means, there can never be an `else` or `elif` block standing alone, only together with `if`. Here's another example. Observe that here, we have two separate pieces of code: * One `if` block (without anything else) * One `if` block together with an accompanying `else` block

[45]:
```python
number = -3
if number < 0:
    print("Less than zero")
if number < 10:
    print("Less than ten")
else:
    print("Greater than ten")
```

```
Less than zero
Less than ten
```

Generally speaking, keep your if/elif/else expressions short and obvious. You don't want to be debugging a christmas tree of conditions.

By the way… if you do this, you're embarrassing yourself:

[46]:
```python
def find_bob(names):
    if "Bob" in names:
        return True
    else:
        return False
names = ["Alice", "Bob", "Peter"]
find_bob(names)
```

[46]:
```
True
```

A few more examples:

[47]:
```python
haystack = range(10)
needle = 30
if needle in haystack:
    print("30 found")
else:
    print("30 not found")
```

```
30 not found
```

[48]:
```python
# A kiosk with a very limited selection...
menu = ["Banana", "Peach", 'Potato']
prices = [1.25, 1.75, 1.20]
selection = 1
if 0 <= selection < len(menu):
    print(f"A {menu[selection].lower()} costs {prices[selection]:.2f}")
else:
    print("Invalid selection")
```

```
A peach costs 1.75
```

3.0.1 ``Truthyness''

In many languages, certain values are automatically converted to a boolean if used in a boolean expression. **Remember**: In Python, any value is considered `True`, except the following: * `False` and `None` * `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)` * `''`, `()`, `[]`, `{}`, `set()`, `range(0)` * A few other values that are not so important right now

See the Official documentation for all the details, or this StackOverflow post for a casual list.

You can easily check which boolean something evaluates to by using the `bool` function:

[49]:
```
bool(0)
```

[49]:
```
False
```

[50]:
```
bool(25)
```

[50]:
```
True
```

[51]:
```
bool(["not", "empty", "list"])
```

[51]:
```
True
```

[52]:
```
bool([])
```

[52]:
```
False
```

[53]:
```
bool("Just a string")
```

[53]:
```
True
```

[54]:
```
bool("")
```

[54]:
```
False
```

That's why you can use an arbitrary value as the condition in an if statement, even if that value is not a boolean:

[55]:
```
if 3+5:
    print("Banana")
if 3-3:
    print("Not going to happen")
```

```
Banana
```

This is most commonly done for differentiating between empty and populated collections:

[56]:
```
names = []
if not names:
```

```
    print("No names!")
else:
    print(names)
```

```
No names!
```

Exercise

Write a function `sign` that takes a single integer as a parameter `number` and returns a string. If `number` is negative, the function should return `"negative"`, if `number` is positive, it should return `"positive"`, and otherwise it should return `"zero"`

[57]:
```
# exercise solution
```

Exercise

Write a function `within` that takes two parameters: 1. a single integer `target` 2. a list of numbers `numbers`

The function should return `True` if `target` is (strictly) between the smallest and largest values in `target`, otherwise it should return `False`

[58]:
```
# exercise solution
```

4 Maps (Dictionaries)

Let's look at the previous example with the mediocre kiosk:

[59]:
```
# A kiosk with a very limited selection...
menu = ["Banana", "Peach", 'Potato']
prices = [1.25, 1.75, 1.20]
selection = 2
if 0 <= selection < len(menu):
    print(f"A {menu[selection].lower()} costs {prices[selection]:.2f}")
else:
    print("Invalid selection")
```

```
A potato costs 1.20
```

First of all, let's * Move the conditional into a function, so we can call it with different selections over and over again * Replace the `print` statements with `return` statements, because maybe we want to further process the result

[60]:
```
menu = ["Banana", "Peach", 'Potato']
prices = [1.25, 1.75, 1.20]
def select(selection):
    if 0 <= selection < len(menu):
        return f"A {menu[selection].lower()} costs {prices[selection]:.2f}"
    else:
        return "Invalid selection"
```

So now we can call it and see what it evaluates to

```
[61]: select(2)
```

```
[61]: 'A potato costs 1.20'
```

In this code, we're managing two separate lists, `menu` and `prices`, and address individual elements by their index. If the kiosk changes its menu, we have to *very carefully* edit both lists to avoid a mess:

```
[62]: # change the price of a product
prices[2] = 1.80                # what is product number 2???
# remove a product
del(menu[1])
del(prices[1])                  # better use the same index or it'll be a mess
# add a product
menu.append("Water")
prices.append(1.50)
# replace a product
menu[0] = "Gummybears"          # which product did we replace???
prices[0] = 0.95
```

```
[63]: select(2)
```

```
[63]: 'A water costs 1.50'
```

Wouldn't it be convenient if we had some way of clearly associating a product with a price?

This is where *maps* come in. In software engineering, a *map* (which Python calls a ``dictionary'', we're going to use those terms interchangeably) has nothing to do with geography. A map is a data structure where keys are associated with values. Let's store the menu in a dictionary:

```
[64]: menu = {"Banana": 1.25, "Peach": 1.75, 'Potato': 1.20}
menu
```

```
[64]: {'Banana': 1.25, 'Peach': 1.75, 'Potato': 1.2}
```

Note that `"Banana"` is a *key*, while `1.25` is a *value*.

The dictionary allows us to retrieve values by their key. This is done with a notation similar to accessing list elements via their index, but instead we use a key. For example:

```
[65]: menu["Potato"]
```

```
[65]: 1.2
```

Now we can rewrite and call our selection function like so:

```
[66]: menu = {"Banana": 1.25, "Peach": 1.75, 'Potato': 1.20}
def select(selection):
    #if 0 <= selection < len(menu):
    if selection in menu:
        #return f"A {menu[selection].lower()} costs {prices[selection]:.2f}"
```

```
        return f"A {selection.lower()} costs {menu[selection]:.2f}"
    else:
        return "Invalid selection"
select("Banana")
```

[66]: 'A banana costs 1.25'

Note that to check whether a *key* exists in a dictionary, you can use the `in` operator. You cannot directly check whether a *value* exists in a dictionary.

[67]:
```
"Peach" in menu
```

[67]: True

And if we want to change the menu, that's now easy. To change a price, we just assign a new value to an existing key:

[68]:
```
menu["Banana"] = 6.55       # "Because of inflation!"
select("Banana")
```

[68]: 'A banana costs 6.55'

Adding a new product looks exactly the same, just with a new key:

[69]:
```
menu["Water"] = 1.50
menu
```

[69]: {'Banana': 6.55, 'Peach': 1.75, 'Potato': 1.2, 'Water': 1.5}

To remove a product, we just delete the key (the value also gets deleted):

[70]:
```
del(menu["Banana"])
menu
```

[70]: {'Peach': 1.75, 'Potato': 1.2, 'Water': 1.5}

A few things to know about dictionaries in Python: * key/value pairs in a dictionary do not have any particular ordering (just like values in a set) * `{}` is the empty dictionary (you can add elements later) * any *key* can only appear once in a dictionary (that's the point), but multiple keys can reference the same *value* * the *values* in a dictionary can be of any arbitrary type * the *keys*, however, must be **immutable**

[71]:
```
things = {}                         # empty dictionary
things[0] = "nil"                   # 0 is the key, not an index
things[0] = "zero"                  # overwrites the value "nil"
things["two"] = 2
things[2] = 2                       # same value can appear multiple times
things["a list"] = [1, 2, 3]        # values can be of mutable types
things[('a', 'tuple', 123)] = 13    # even a tuple can be a key, as long all the␣
 ↪contents are immutable
```

```
# NOT possible because lists are mutable; they cannot be dictionary keys:
#things[['a', 'list']] = "invalid"

print(things[('a', 'tuple', 123)]) # that tuple really works as a key!
print(things)
```

```
13
{0: 'zero', 'two': 2, 2: 2, 'a list': [1, 2, 3], ('a', 'tuple', 123): 13}
```

You'll often want to only deal with the keys or the values of the dictionary. For this, simply call `.keys()` or `.values()`, respectively.

[72]:
```
menu = {"Banana": 1.25, "Peach": 1.75, 'Potato': 1.20}
menu.keys()
```

[72]:
```
dict_keys(['Banana', 'Peach', 'Potato'])
```

[73]:
```
menu.values()
```

[73]:
```
dict_values([1.25, 1.75, 1.2])
```

You can more or less ignore that this gives you `dict_keys` and `dict_values`, but you could convert them to lists:

[74]:
```
list(menu.keys())
```

[74]:
```
['Banana', 'Peach', 'Potato']
```

Finally, you may want to get the key/value-pairs as a list of tuples. You can do this using `items()`:

[75]:
```
menu.items()
```

[75]:
```
dict_items([('Banana', 1.25), ('Peach', 1.75), ('Potato', 1.2)])
```

[76]:
```
list(menu.items())
```

[76]:
```
[('Banana', 1.25), ('Peach', 1.75), ('Potato', 1.2)]
```

Exercise

Implement a phonebook. Note the following: * The implementation should assume that contacts are stored in a dictionary where keys are the names of people and values are their phone numbers (as strings). * Implement a function `add_contact(phonebook, name, number)` which adds a new entry to `phonebook`, but only if `name` is not already in the phonebook. In the latter case, it should print `"Already in phonebook"` * Implement a function `delete_contact(phonebook, name)` which removes an entry from `phonebook` and ignores the case where the given name is not in `phonebook`. * Implement a function `find_contact(phonebook, name)` which *returns* the number of the given contact. If the contact is not in `phonebook`, it should *print* `"Not found"`.

[77]:
```python
# exercise solution
my_contacts = {
    "Joe": "+41440002341",
    "Anne": "+41441112341",
}

def add_contact(phonebook, name, number):
    pass
def delete_contact(phonebook, name):
    pass
def find_contact(phonebook, name):
    pass

#add_contact(my_contacts, "Bob", "+41449999912")
#add_contact(my_contacts, "Alice", "+41440000012")
#delete_contact(my_contacts, "Bob")
#find_contact(my_contacts, "Alice")
```

5 List comprehensions

Sometimes, you have a collection of values, and you want to do the same thing to each of the values to create a new list. This is where Python uses *list comprehensions*.

[78]:
```python
names = ["bob", "alice", "tony"]
[n.capitalize() for n in names]
```

[78]:
```
['Bob', 'Alice', 'Tony']
```

Note that `names` has **not** been changed! The list comprehension creates a *new* list with the transformed values:

[79]:
```python
capitalized_names = [n.capitalize() for n in names]
print(names)
print(capitalized_names)
```

```
['bob', 'alice', 'tony']
['Bob', 'Alice', 'Tony']
```

A few more examples:

[80]:
```python
[n*2 for n in range(10, 2, -2)]
```

[80]:
```
[20, 16, 12, 8]
```

[81]:
```python
[f"{n} squared is {n**2}" for n in range(5)]
```

[81]:
```
['0 squared is 0',
 '1 squared is 1',
 '2 squared is 4',
```

```
 '3 squared is 9',
 '4 squared is 16']
```

[82]:
```
sum([i**2 for i in range(3,7)])
```

[82]:
```
86
```

When using a list comprehension, you can conveniently filter the input collection by appending an `if` condition at the end, for example:

[83]:
```
[f"{n} squared is {n**2}" for n in range(10) if n % 2 == 0]
```

[83]:
```
['0 squared is 0',
 '2 squared is 4',
 '4 squared is 16',
 '6 squared is 36',
 '8 squared is 64']
```

[84]:
```
[character for character in "Hello, World!" if character.isalpha()]
```

[84]:
```
['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd']
```

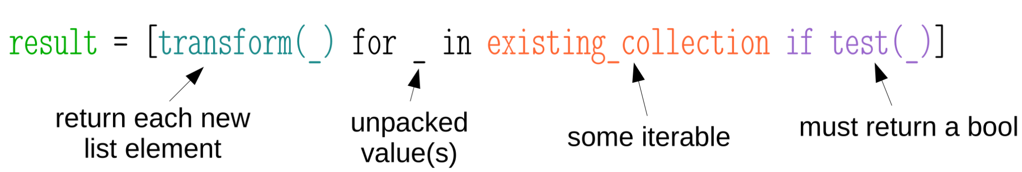
[85]:
```
[character for character in "Hello, World!" if character.lower() in "aeiou"]
```

[85]:
```
['e', 'o', 'o']
```

[86]:
```
sum([int(character) for character in "c1h291hgf93e" if character.isdigit()])
```

[86]:
```
25
```

This illustrates the general syntax:



[87]:
```
def cleanup(name):
    return name.strip().capitalize()
def is_string(name):
    return type(name) == str
[cleanup(name) for name in ["   alice   ", 123, False, "bob"] if is_string(name)]
```

[87]:
```
['Alice', 'Bob']
```

Exercise

Write an expression that determines the length of the longest word in a list of words.

[88]:
```
# exercise solution
words = "The latter consisted simply of six hydrocoptic marzlevanes".split()
```

6 Dict comprehensions

The exact same concept exists for dictionaries as well. If you have a collection of values and would like to create a new dictionary from them, use a similar syntax. The important thing to understand is that instead of producing *one* value at a time, like for a list comprehensions, you're producing *key*: *value* pairs (`key: value`).

Instead of just creating one number at a time for the resulting list…

[89]:
```
[n**2 for n in range(10)]
```

[89]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

…this example creates a key: value pair for the resulting dictionary:

[90]:
```
{n: n**2 for n in range(10)}
```

[90]:
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Here's a dictionary where the keys are the characters that appear in a string and the values are how many times that character appears:

[91]:
```
sentence = "Hello, world!"
{char: sentence.count(char) for char in set(sentence)}
```

[91]:
```
{'o': 2,
 '!': 1,
 'l': 3,
 'w': 1,
 ' ': 1,
 'H': 1,
 'r': 1,
 'e': 1,
 ',': 1,
 'd': 1}
```

Again, you can add a filter at the end. Here, we're ignoring vowels:

[92]:
```
{char: sentence.count(char) for char in set(sentence) if char not in "aeiou"}
```

[92]:
```
{'!': 1, 'l': 3, 'w': 1, ' ': 1, 'H': 1, 'r': 1, ',': 1, 'd': 1}
```

Here's our menu dictionary again:

[93]:
```
menu = {"Banana": 1.25, "Peach": 1.75, 'Potato': 1.20}
```

Let's create a list of all products which start with a ``P'', using a list comprehension, using only the dictionary keys as input:

[94]:
```
[product for product in menu.keys() if product.startswith("P")]
```

[94]: ['Peach', 'Potato']

Or, if we want to also keep the prices, then we use a dict comprehension with the same filter:

[95]:
```
{product: price for product, price in menu.items() if product.startswith("P")}
```

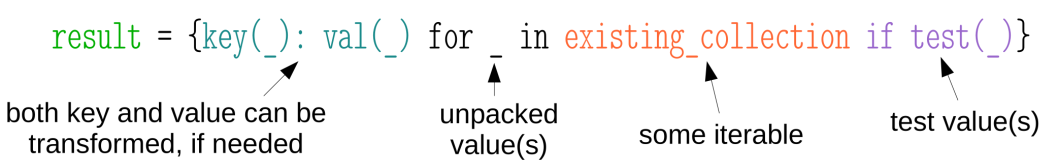
[95]: {'Peach': 1.75, 'Potato': 1.2}

Note that we state `for product, price in menu.items()`, meaning each individual element that's being transformed will be a tuple like `("Peach", 1.75)`, which is why we can assign the two tuple values to the local variables product and price. Remember that you can assign multiple values on the left side of an expression, e.g.:

[96]:
```
name, age = "Alice", 37
```

So `for product, price in menu.items()` is essentially the same thing, for each individual key-value pair in `menu.items()`

This illustrates the general syntax:



[97]:
```
def cleanup_key(name):
    return name.strip().capitalize()
def number_as_binary(number):
    return bin(number)
def is_integer(value):
    return type(value) == int
{cleanup(name): number_as_binary(number) for name, number in {"bob": 3.5,␣
↪"alice": 3}.items() if is_integer(number)}
```

[97]: {'Alice': '0b11'}

Exercise

Write a dict comprehension that takes in a list of words and produces a dictionary mapping words to their length. All keys should be lower-case.

[98]:
```
# exercise solution
words = "How much wood would a woodchuck chuck if a woodchuck could chuck wood".
↪split()
```