Readings for Unit 4

Back to Top

Licensing Information

Portions of these readings were modified or copied verbatim from *Think Python 2e* by Allen Downey.

PDF of these readings also available to download: 6s090_reading4.pdf

Table of Contents

1) Introduction

In previous units, we have learned about a variety of useful types including functions, lists, tuples, strings, floats, ints, bools, and `None`. This week we're going to expand this list to include two new powerful built-in types: *dictionaries* and *sets*.

2) Dictionaries (e.g., `{'dog': 'Hund', 'cat': 'Katze'}`)

First up: dictionaries. Dictionaries are one of Python's best features; they are the building blocks of many efficient and elegant programs. In some ways, a dictionary is like a list, but more general. Lists and dictionaries are both compound objects, and they are both mutable. In a list, though, the indices have to be integers, whereas in a dictionary they can be any *hashabl*Back to Top
(completely immutable) object.

A dictionary contains a collection of unique indices, which are called *keys*, and a collection of values. Each key is associated with a single *value*. The association of a key and a value is called a *key-value pair* or sometimes an *item*.

In mathematical language, we might say that a dictionary represents a *mapping* from keys to values, so you can also say that each key "maps to" a value. As an example, let's build a dictionary that maps from English to German words, so the keys and the values are all strings.

Lists in Python are created with square brackets (e.g., `new_list = []`). Dictionaries, on the other hand, are created with squiggly brackets. For example, we could make an empty dictionary with:

```
en_de = {}
```

To add items to the dictionary, you can use square brackets:

```
en_de['dog'] = 'Hund'
```

This line creates an item that maps the key `'dog'` to the value `'Hund'`. If we print the dictionary now, we see a key-value pair with a colon between the key and value:

```
print(en_de)  # prints {'dog': 'Hund'}
```

This output format is also an input format. For example, you can create a new dictionary with three items:

```
en_de = {'dog': 'Hund', 'cat': 'Katze', 'guinea pig': 'Meerschweinchen'}
```

If you print `en_de`, you will see `{'dog': 'Hund', 'cat': 'Katze', 'guinea pig': 'Meerschweinchen'}`. [1]

To access a value in a dictionary, you can also use the square brackets:

```
print(en_de['cat'])  # prints Katze
```

If the key isn't in the dictionary, you get an error:

```
print(en_de['eagle'])  # gives us an error: KeyError: 'eagle'
```

So far, we've seen that square brackets are used to look up values in a dictionary, and also to add new items to a dictionary. It turns out that they are also used to *change* an existing mapping. For example, if we wanted our `en_de` dictionary to store the Swiss dialect German words, we could update our dictionary like so to account for this change:

```
en_de['dog'] = 'Hundli'
print(en_de)  # prints {'dog': 'Hundli', 'cat': 'Katze', 'guinea pig': 'Meerschweinchen'}
```

Back to Top

But we speak High German, so let's put it back:

```
en_de['dog'] = 'Hund'
```

2.1) Dictionary operations: `{}` `len(d)` `d[key] = value` `{1: [2]} == {1: [2]}`

In addition to creating empty dictionaries with `{}` and accessing or setting the values of a dictionary with indexing `en_de['dog'] = 'Hund'`, we can quickly determine the number of dictionary keys using `len`. For example:

```
>>> en_de = {'dog': 'Hund', 'cat': 'Katze', 'guinea pig': 'Meerschweinchen'}
>>> len(en_de)  # how many keys are in this dictionary?
3
```

We can also determine whether two dictionaries are equal with `==`. While sequences check for equality by checking that all the elements are equal and in the same order, dictionary equality checks that both dictionaries have the exact same key-value items, in any order:

```
>>> en_de == {'dog': [], 'cat': [], 'guinea pig': []}
False  # not equal because keys have different values
>>> en_de == {'guinea pig': 'Meerschweinchen', 'cat': 'Katze', 'dog': 'Hund'}
True   # equal becauase same keys with same values, even though different order
```

2.2) Hashability

Importantly, dictionaries are not limited to containing strings. They can have arbitrary (any) objects as values, and arbitrary *hashable* objects as keys. An object is *hashable* if it has an unchanging hash value. You can determine whether an object is hashable by calling the `hash` function on it, which returns an integer. For example:

```
>>> hash(5)
5
>>> hash("hello")
-3280404559733277131
>>> hash(None)
-9223363241310935348
>>> hash(5.4)
922337203685478405
>>> hash((1, 2, 3))
529344067295497451
```

Calling the `hash` function on an unhashable object will raise a `TypeError`:[2]

```
>>> hash([1,2,3])
    ...
TypeError: unhashable type: 'list'
>>> hash(([1, 2], 3))
    ...
TypeError: unhashable type: 'list'
>>> hash ({"Hund": "dog"})
    ...
TypeError: unhashable type: 'dict'
```

Back to Top

Note that the hash value of an object can change between different runs of the same program. However, while a program is running, a hashable object will have a constant hash value. Additionally, objects that are equal (meaning `object1 == object2` is True) will have the same hash value.

In general, `None` and all ints, floats, booleans, and strings are hashable because they are immutable. Lists, dictionaries, and other mutable objects that can change are not hashable because their hash value is not guaranteed to be constant. *Tuples are hashable only if all the elements inside the tuple are hashable* (which is why `((1, 2), 3)` is hashable but `([1, 2], 3)` is not.)

Python implements dictionaries as a structure called a *hash map*. The details of this kind of structure are a bit beyond the scope of this class, but it is sufficient to know that the hash value of a key must be constant because the hash value is used to find the location in memory where the key's value is stored.

2.3) Example: Letter Frequency

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
2. You could create a list with 26 elements. Then you could convert each character to a number, use the number as an index into the list, and increment the appropriate counter.
3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```
def letter_count(string):
    '''
    Given a string, create a dictionary that counts the frequency of each character.
    '''
    counter = {} # make a new dictionary
    for char in string:
        if char not in counter:
            counter[char] = 1  # initialize count of new letter
```

```
        else:
            counter[char] += 1 # increment count of old letter
    return counter
```

Back to Top

The first line of the function creates an empty dictionary `counter`. The `for` loop traverses the string. Each time through the loop, if the character `char` is not in the dictionary, we create a new item with key `char` and the initial value 1 (since we have seen this character once). If `char` is already in the dictionary, we increment `counter[char]`.

If we printed the result of `letter_count('brontosaurus')`, python would output:

```
{'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
```

This indicates that the letters `'b'` and `'n'` appear once; `'r'` appears twice, and so on.

2.4) Looping and Dictionaries

If you loop over a dictionary in a `for` statement, it traverses the keys of the dictionary. For example, we could print each key and corresponding value of a dictionary as follows:

```
dino_count = letter_count('brontosaurus')

for key in dino_count:
    val = dino_count[key]
    print(key, val)
```

and it will produce the following output:

```
b 1
r 2
o 2
n 1
t 1
s 2
a 1
u 2
```

2.5) Example: Reverse Lookup

As we've seen, given a dictionary `d` and a key `k`, it is easy to find the corresponding value `v = d[k]`. This operation is called a lookup.

But what if you have a value `v` and you want to find `k`? You have two problems: first, there might be more than one key that maps to the value `v`. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a reverse lookup; you have to search.

The `value_lookup` function below presents one possible solution:

```
def value_lookup(d, val):
    '''
    Given a dictionary, return a list of keys in the dictionary that have
    values equal to val.
    '''
    keys = []
    for key in d:
        if d[key] == val:
            keys.append(key)
    return keys


dino_count = letter_count('brontosaurus')
print(value_lookup(dino_count, 2))  # ['r', 'o', 's', 'u']
```

Back to Top

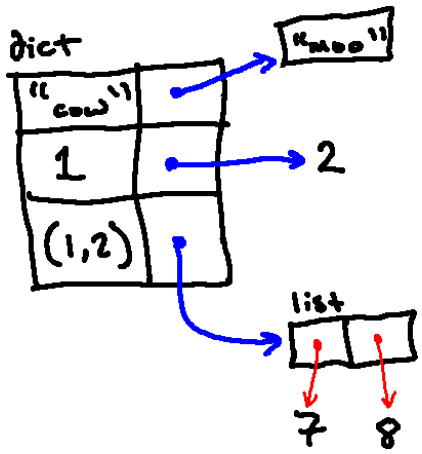
2.6) Dictionaries and Environment Diagrams

As we've done whenever we've introduced a new type of Python object, we'll learn a way to represent dictionaries in environment diagrams. Note that a dictionary is much like a frame in some ways; it is a mapping between keys and values. So our representation will look very similar: we'll put the keys in the left-hand side of this mapping, and the right-hand side will consist of pointers to the associated values.

For example, the following dictionary:

```
{'cow': "moo", 1: 2, (1, 2): [7, 8]}
```

would be represented as follows in an environment diagram:



3) Sets (e.g., `{'oranges', 'bananas', 'tomatoes'}`)

Sets can be thought of as similar to a collection of dictionary keys without values. Like dictionary keys, elements in a set are unordered, hashable, and unique.

Like dictionaries, we define sets using curly brackets:

```
>>> produce = {"tomatoes", "oranges", "bananas", "oranges"}
>>> produce
{'oranges', 'bananas', 'tomatoes'}
```

Back to Top

Displaying the value of `produce` in the python shell shows that sets have no duplicates and can shuffle the order of the original elements. Note that re-running this exact same code could produce different orderings of the elements.

Unlike dictionaries and sequences we cannot index into sets:

```
>>> produce['tomatoes']
    ...
TypeError: 'set' object is not subscriptable
```

But we can iterate over them with a for loop:

```
for item in produce:
    print(item)
'''
oranges
bananas
tomatoes
'''
```

3.1) Set operations: `set()`, `len(x)`, `x.add(item)`, `{1, 2} == {2, 1}`

We can also make an empty set and *add* elements to it. The `set()` command creates a new empty set (`{}` creates an empty dictionary). While lists add elements to the end of a list with `x.append(element)`, sets use the `x.add(element)` method (sets are unordered so elements are not added to the "end").

```
>>> nums = set()  # create empty set
>>> nums.add(5)
>>> nums.add(5)
>>> nums.add(3)
>>> nums
{3, 5}
```

Like dictionaries and sequences, we can determine how many unique elements are in a set using `len`:

```
>>> len(nums)
2
```

We can also check whether a set is equal to another object using `==`. Like dictionaries, set equality depends on both objects being sets and containing the same elements (in any order).

```
>>> [1, 2, 3] == [3, 2, 1]
False  # list equality cares about the order of elements
>>> {1, 2, 3} == {3, 2, 1}
True   # set equality does not care about order
>>> {1, 2, 3} == [1, 2, 3]
False  # but does care about type
```

Back to Top

3.2) Example: Unique Words

Say you were given a book (in the form of a list of strings representing individual words) and asked to determine how many unique words the book contained. Without dictionaries and sets, this would likely involve an inefficient process of building a list of unique words. For each word in the book, we would need to check whether it was already present in our list of unique words before appending it.

While we could use a strategy similar to our `letter_count` function to count the frequency of each word, since we only care about the number of unique words, using a set would be more appropriate:

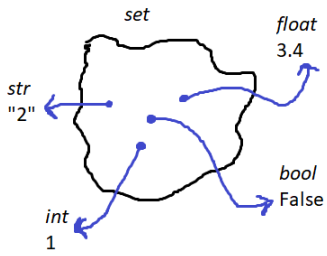
```
# A line from "Goodnight Moon" by Margaret Wise Brown
moon = ['goodnight', 'stars', 'goodnight', 'air', 'goodnight', 'noises', 'everywhere']
# The first paragraph to "The House on Mango Street" by Sandra Cisneros
mango = ['we', "didn't", 'always', 'live', 'on', 'mango', 'street', 'before', 'that', 'we', 'lived', 'on',
'loomis', 'on', 'the', 'third', 'floor', 'and', 'before', 'that', 'we', 'lived', 'on', 'keeler', 'before',
'keeler', 'it', 'was', 'paulina', 'and', 'before', 'that', 'i', "can't", 'remember', 'but', 'what', 'i',
'remember', 'most', 'is', 'moving', 'a', 'lot', 'each', 'time', 'it', 'seemed', "there'd", 'be', 'one',
'more', 'of', 'us', 'by', 'the', 'time', 'we', 'got', 'to', 'mango', 'street', 'we', 'were', 'six', 'mama',
'papa', 'carlos', 'kiki', 'my', 'sister', 'nenny', 'and', 'me']

def unique_words(book):
    """
    Given a list of words from a book, calculate and return the
    number of unique words in the book
    """
    words = set()
    for word in book:
        words.add(word)
    return len(words)

print(len(moon), unique_words(moon))    #  7 words in the line,      5  unique words
print(len(mango), unique_words(mango))  # 74 words in the paragraph, 51 unique words
```

3.3) Sets and Environment Diagrams

Sets can also mix-and-match and contain different *hashable* objects:

```
{1, "2", 3.4, False}
```

Because sets are unordered, we represent them as a blob with pointers to the values. We can represent the set above in an environment diagram as:



Back to Top

4) The `in` operator

The `in` operator is a handy Python shortcut that can be used to determine whether a collection of objects contains a particular value. Using the `in` operator results in performing a containment check on the object. For example:

```
# for lists and tuples "in" checks if any single element is == to the desired value
>>> 5 in [1, 2, 3, 4]
False
# which does not include subsequences
>>> [1, 2] in [1, 2, 3, 4]
False
# or deeply searching nested objects
>>> 5 in ([5], [4], [3])
False
# it only checks the top-layer
>>> [5] in ([5], [4], [3])
True
# sets behave similarly
>>> 4 in {1, 2, 3, 4}
True
# however, for strings "in" can check for substrings
>>> "a" in "apple"
True
>>> "app" in "apple"
True
# for dictionaries, "in" checks whether the object
# is == to one of the dictionary keys, not values
>>> 4 in {"key": 4, "other": 2}
False
>>> 'key' in {"key": 4, "other": 2}
True
# sets and dictionaries can only do containment checks for hashable objects
>>> [5] in {1, 2, 3, 4}
TypeError: unhashable type: 'list'
>>> {"key": 4} in {"key": 4, "other": 2}
TypeError: unhashable type: 'dict'
# other primitive types cannot do containment checks
>>> 5 in 5
TypeError: argument of type 'int' is not iterable
# but generally, if you can loop over something with "for x in thing"
# you can check "if x in thing"
```

We can also check if a collection of objects does *not* contain a particular value by using `not in`:

Back to Top

```
>>> 5 not in [1, 2, 3, 4]
True
>>> "a" not in "apple"
False
```

Besides having the handy property of avoiding duplicates, dictionaries and sets are also useful because they are optimized for fast containment checks. Because dictionaries and sets use an object's `hash` value to determine where to store an object, upon seeing that object again they know exactly where to look for it. In contrast, sequences are ordered by index, so to determine whether a sequence contains a particular value, Python potentially needs to look at the entire sequence. For objects with lots of elements, this can make quite a difference in runtime:

```
def speed_check(nums, val):
    # time how long it takes to check whether val is in nums 1000 times
    import time
    count = 0
    start = time.time()
    for i in range(1000):
        count += val in nums
    end = time.time()
    return {"exists": count, "time (s)": end-start}


# create a list and a set of numbers from 1 to 1 million
num_set = set()
num_list = []
for i in range(1, 1_000_001):
    num_set.add(i)
    num_list.append(i)



print("How long does it take to check that 1 exists in the set 1000 times?")
print(speed_check(num_set, 1))

print("What about the list?")
print(speed_check(num_list, 1))

print("How long does it take to check that 1,000,000 exists in the set 1000 times?")
print(speed_check(num_set, 1_000_000))

print("What about the list?")
print(speed_check(num_list, 1_000_000))

print("How long does it take to check that 1,000,001 does not exist in the set 1000 times?")
print(speed_check(num_set, 1_000_001))

print("What about the list?")
print(speed_check(num_list, 1_000_001))
```

On my machine, running this program outputs:

Back to Top

```
How long does it take to check that 1 exists in the set 1000 times?
{'exists': 1000, 'time (s)': 5.626678466796875e-05}
What about the list?
{'exists': 1000, 'time (s)': 8.821487426757812e-05}
How long does it take to check that 1,000,000 exists in the set 1000 times?
{'exists': 1000, 'time (s)': 0.00010418891906738281}
What about the list?
{'exists': 1000, 'time (s)': 6.894873857498169}
How long does it take to check that 1,000,001 does not exist in the set 1000 times?
{'exists': 0, 'time (s)': 3.838539123535156e-05}
What about the list?
{'exists': 0, 'time (s)': 10.412419319152832}
```

Checking if `1 in num_set` and `1 in num_list` a thousand times takes roughly the same amount of time (0.00006 vs 0.00009 seconds) because 1 is quickly found at the beginning of the list. Checking `1000000 in nums` takes .0001 seconds for the set but almost 7 seconds for the list because 1,000,000 is all the way at the end of the list. Computers are fast, but checking 1,000,000 memory locations a thousand times is a lot of work! Checking that `1000001 in nums` is similarly fast for sets at 0.00004 seconds, but takes over 10 seconds for the list because Python needs to check that all million numbers in the list are not equal to 1,000,001 a thousand times.

5) Syntatic Sugar

While the previous sections contain all the information you need to know to complete the assignments for this unit, the following sections contain additional useful operations.

5.1) Other Dictionary Operations:

The Python documentation for dictionaries describes a number of additional dictionary operations. The sections below contain some highlights, including easier ways to combine dictionaries and access values (even for non-existent keys.)

5.1.1) Concatenating dictionaries `{1: 2} | {3: []}`

Unlike previous data types, we cannot concatenate dictionaries using `+`:

```
>>> {1: 2} + {3: []}
...
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

Instead, we use `|` (which is supposed to represent the mathematical union operator) to combine the keys and values from two dictionaries into a new dictionary. Note that if the dictionaries contain overlapping keys, the values in the second dictionary will overwrite the values in the first dictionary:

```
>>> {1: 2} | {3: []}
{1: 2, 3: []}
```

```
>>> {1: 2, "hi": "there"} | {3: [], 1: [4, 5]}
{1: [4, 5], 'hi': 'there', 3: []}
```

Back to Top

5.1.2) Accessing keys and values with `d.keys()` `d.values()` `d.items()`

Sometimes instead of accessing a single value in a dictionary, we wish to access all the keys or all the values in a dictionary. Luckily, dictionaries have the `keys()`, `values()` and `items()` methods to do just that!

```
>>> en_de = {'dog': 'Hund', 'cat': 'Katze', 'guinea pig': 'Meerschweinchen'}
>>> en_de.keys()
dict_keys(['dog', 'cat', 'guinea pig'])
>>> en_de.values()
dict_values(['Hund', 'Katze', 'Meerschweinchen'])
>>> en_de.items()
dict_items([('dog', 'Hund'), ('cat', 'Katze'), ('guinea pig', 'Meerschweinchen')])
```

These methods return custom dictionary objects, but you can easily cast them to a list or a tuple as needed:

```
>>> list(en_de.keys())
['dog', 'cat', 'guinea pig']
>>> tuple(en_de.values())
('Hund', 'Katze', 'Meerschweinchen')
>>> list(en_de.items())
[('dog', 'Hund'), ('cat', 'Katze'), ('guinea pig', 'Meerschweinchen')]
```

Looping over a dictionary like in the program below:

```
dino_count = {'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
for char in dino_count:
    print(char, dino_count[char])
```

is equivalent to using the dictionary's built-in function `.keys()`:

```
for char in dino_count.keys():
    print(char, dino_count[char])
```

Again, you should generally not assume that the keys are in a particular order. To traverse the keys in sorted order, you can use the built-in function `sorted`:

```
for key in sorted(dino_count):
    print(key, dino_count[key])
```

produces the following output:

```
a 1
b 1
n 1
o 2
r 2
s 2
t 1
u 2
```

Back to Top

If your loop only uses the values in the dictionary, you can loop over the dictionary's `.values()`:

```
for val in dino_count.values():
    print(val) # prints the values
```

This is equivalent to the below:

```
for key in dino_count:
    print(dino_count[key])
```

You can also loop over both the keys and values at the same time, via `.items()`:

```
for item in dino_count.items():
    print(item) # prints a tuple (key, value)
    # can access the key via item[0]; the value via item[1]
```

You can also *unpack* this tuple of values into two separate variables:

```
for key, value in dino_count.items():
    print(key, value)
```

This is equivalent to below:

```
for key in dino_count:
    print(key, dino_count[key])
```

5.1.3) Avoiding KeyErrors with `d.get(key)`

Code that checks whether a key exists in a dictionary, and use a default value if it does not, is a very common pattern. For example:

```
>>> dino_count = {'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
>>> dino_count['z']
...
```

```
KeyError: 'z'
# Oh no!
>>> if 'z' not in dino_count:
        print(0)
0
# Avoids key error
```

Back to Top

Because of this, Python provides an easier way to accomplish it: `get`.

To use `get`, you write it with a key and a default value. If the key appears in the dictionary, `get` returns the corresponding value; otherwise, it returns the default value. For example:

```
print(dino_count.get('a', 0))  # 1
print(dino_count.get('z', 0))  # 0
```

Try Now:

Use `get` to write the letter counter code segment more concisely. You should be able to eliminate the `if / else` statement.

Show/Hide

Here is one solution:

```
def letter_count(string):
    '''
    Given a string, create a dictionary that counts the frequency of each character.
    '''
    counter = {} # make a new dictionary
    for char in string:
        counter[char] = counter.get(char, 0) + 1
    return counter

print(letter_count('brontosaurus'))
```

5.2) Additional set operations: `set(x) s1 | s2 s1 - s2 s1 & s2`

The official Python documentation for sets also has a number of useful operations. For example, we can also convert any collection of *entirely hashable* values to a set (and sets can be converted to a list, tuple, or string) using type casting:

```
>>> set([1, 1, 2, 2, 3, 3])
{1, 2, 3}
>>> set(('my', 'favorite', 'tuple', 'my', ('inner', 'tuple')))
{('inner', 'tuple'), 'favorite', 'my', 'tuple'}
```

```
>>> set("hello")
{'l', 'h', 'o', 'e'}
>>> set(["good", ("fine",), (['uh oh'])])
    ...
TypeError: unhashable type: 'list'
```

Back to Top

Additional operations can quickly determine the equality, intersection, difference, and union of sets:

```
>>> nums1 = {1, 2, 3}
>>> nums2 = {3, 4, 5}
>>> nums1 & nums2 # intersection: all elements that appear in both sets
{3}
>>> nums1 - nums2 # difference: all elements from first set that do not appear in second set
{1, 2}
>>> nums1 | nums2 # union: all elements from both sets combined
{1, 2, 3, 4, 5}
```

5.3) Tips and Tricks for Writing Concise Code

As we've discussed before, most people, programmers included, love shortcuts. After all, why take the long route if the short route is faster and produces the same result? Well it turns out when learning a new language, it is quite difficult to memorize all possible routes, so we have structured the readings to emphasize the basics first so that the fancier bells and whistles don't get in the way. However, if you feel comfortable using the operations described above, you may find the following sections on concise conditional statements and loops useful.

5.3.1) Ternary Statements `TRUE_EXPRESSION if CONDITION else FALSE_EXPRESSION`

One example of alternate syntax that can help make common operations that take a few lines of code only take a single line of code is a *ternary statement*, or one-line if statement:

```
# print whether x is even or odd
x = 5
if x % 2 == 0:
    print("even")
else:
    print("odd")

# can also be written as
if x % 2 == 0:  print("even")
else: print("odd")
```

Note that this pattern should only be used when there is only a single line of code in the body of the conditional statement.

However, we can write this if / else statement even more concisely in one line as:

```
print("even" if x % 2 == 0 else "odd")
```

This is known as a ternary statement. Ternary statements have the pattern of `TRUE_EXPRESSION if CONDITION else FALSE_EXPRESSION` and can be used to make a four-line if / else statement into a single line, especially when it involves making a decision between two values. Back to Top

Here's another example:

```
def positive_double(num):
    if num > 0:
        return num * 2
    else:
        return num * -2

def positive_double(num):
    return num * 2 if num > 0 else num * -2

print(positive_double(42))  # 84
print(positive_double(-5))  # 10
```

It's important to note that making code more concise does not necessarily make it better. Ternary statements can make the code harder to debug, and especially when it makes a single long line the code can become much less readable.

For example:

```
x = 5
y = 'This really long string that says x is positive' if x > 0 else 'A slightly different really long
string that says x is not positive'

# this can be made slightly easier to read if you surround the statement
# with round brackets () and split it up over multiple lines
y = ('This really long string that says x is positive' if x > 0 else
      'A slightly different really long string that says x is not positive')

# but at this point it might be better to use a regular if/else so you
# can clearly see the conditional
if x > 0:
    y = 'This really long string that says x is positive'
else:
    y = 'A slightly different really long string that says x is not positive'
```

5.3.2) Variable Unpacking `a, b = 1, 2`

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap `a` and `b`:

```
temp = a
a = b
b = temp
```

This solution is cumbersome; *tuple assignment* is more elegant:

Back to Top

```
a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right have to be the same:

```
a, b = 1, 2, 3  # this produces: ValueError: too many values to unpack
```

This error message might seem weird, but this idea of assigning each element in a sequence to a different variable name is often referred to as *unpacking* that sequence.

For example, consider the following three pieces of code to compute the distance between points (where points are represented as `(x, y)` tuples.)

```
def distance(pt1, pt2):
    return ((pt1[0] - pt2[0])**2 + (pt1[1] - pt2[1])**2)**0.5

def distance(pt1, pt2):
    x1 = pt1[0]
    y1 = pt1[1]
    x2 = pt2[0]
    y2 = pt2[1]
    return ((x1 - x2)**2 + (y1 - y2)**2)**0.5

def distance(pt1, pt2):
    x1, y1 = pt1
    x2, y2 = pt2
    return ((x1 - x2)**2 + (y1 - y2)**2)**0.5
```

The first function definition is nice and to-the-point, but it is a bit hard to read because the indexing operations are all collapsed together with the mathematical operation to compute the distance. The second makes the last line easier to read, but at the expense of having to write 4 extra lines of code. The last, I think, strikes a nice balance between the two.

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into its username and domain name, we could do:

```
address = 'a_clever_username@mit.edu'
uname, domain = address.split('@')
```

The return value from `split` is a list with two elements; the first element is assigned to `uname`, the second to `domain`.

```
print(uname)  # prints a_clever_username
print(domain)  # prints mit.edu
```

Unpacking is also commonly used in for-loops, like we have seen above with iterating over dictionary key-value pairs:

Back to Top

```
en_de = {'dog': 'Hund', 'cat': 'Katze', 'guinea pig': 'Meerschweinchen'}

for key, value in en_de.items():
    print(key)
    print(value)

# equivalent to
for key in en_de:
    print(key)
    print(en_de[value])
```

5.3.3) Loop Comprehensions `[EXPRESSION for x in thing]`

In the previous units, we have seen a common pattern related to **creating a list of elements based on another sequence**. For example, this function takes a list of strings and returns a new list of strings with an `"s"` added to each:

```
def add_s_to_all(words):
    res = []
    for word in words:
        res.append(word + "s")
    return res
print(add_s_to_all(["cake", "pie", "cookie"]))  # ["cakes", "pies", "cookies"]
```

We can write this more concisely using a *list comprehension*:

```
def add_s_to_all(words):
    return [word + "s" for word in words]
```

You can almost read the code like English, but here's what this means:

- The square brackets indicate that we are constructing a new list.
- The first expression inside the brackets (`word + "s"`) specifies the elements of the new list.
- The `for` clause indicates what sequence (the `words` list) we are traversing.

The syntax of a list comprehension is a little awkward because the loop variable, `word` in this example, appears in the expression before we get to the definition.

You can also use this syntax to construct lists from *other* sequences, e.g., strings or tuples.

This function takes a string and returns a list containing all the letters in upper case:

```
def uppercase_list(word):
    return [letter.upper() for letter in word]

    # this list comprehension is equivalent to:
    result = []
```

```
    for letter in word:
      result.append(letter.upper())
    return result

    # or
    return list(word.upper())

print(uppercase_list('tomatoes')) # this prints ['T', 'O', 'M', 'A', 'T', 'O', 'E', 'S']
```

Back to Top

This function adds the index of an element to the number:

```
def add_index(nums):
    return [nums[i] + i for i in range(len(nums))]

    # equivalent to:
    result = []
    for i in range(len(nums)):
        result.append(nums[i] + i)
    return result

print(add_index([1,2,3,4])) # this prints [1, 3, 5, 7]
print(add_index((1,1,1,1))) # this prints [1, 2, 3, 4]
```

We can also use ternary statements inside list comprehensions. For example, this function takes a word and returns a list of letters in AlTeRnAtInG cAsE:

```
def alternating_case(word):
    return [(word[i].upper() if (i % 2) == 0 else word[i].lower())
                for i in range(len(word))]

    # is equivalent to:
    result = []
    for i in range(len(word)):
        if (i % 2) == 0:
            result.append(word[i].upper())
        else:
            result.append(word[i].lower())
    return result

print(alternating_case('tomatoes'))
# this prints ['T', 'o', 'M', 'a', 'T', 'o', 'E', 's']
print(alternating_case('other_case'))
# this prints ['O', 't', 'H', 'e', 'R', '_', 'C', 'a', 'S', 'e']
```

An if statement at the end of a list comprehension filters what gets added to the new list. This function takes a list of numbers and returns a list with only even numbers:

```
def evens_only(nums):
    return [num for num in nums if num % 2 == 0]
```

```
    # this is equivalent to:
    result = []
    for num in nums:
        if num %2 == 0:
            result.append(num)
    return result

print(evens_only([1,2,3,4,8,7])) # this prints [2, 4, 8]
```

Back to Top

This function takes a dictionary and returns a new list containing all its values:

```
def list_of_values(d):
    res = []
    for key in d:
        res.append(d[key])
    return res
```

There are many other ways you could write this function:

```
def list_of_values(d):
    res = []
    for value in d.values(): # iterate over the values explicitly
        res.append(value)
    return res
```

```
def list_of_values(d):
    return list(d.values()) # get the values and make them into a list
```

You could also rewrite it using a list comprehension, either of these ways:

```
def list_of_values(d):
    return [d[key] for key in d]
```

```
def list_of_values(d):
    return [value for value in d.values()]
```

Compare these two to the above functions. Again, the square brackets indicate the creation of a new list, the first expression in the square brackets describes what elements to add to the list, and the `for` clause indicates what sequence to traverse.

You can also use comprehensions to create tuples, dictionaries, and sets:

```
>>> tuple(i**2 for i in range(5))  # note including tuple is important
(0, 1, 4, 9, 16)
```

```
>>> {i**2 for i in range(5)}
{0, 1, 4, 9, 16}
>>> {i: i**2 for i in range(5)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Back to Top

Note how the syntax for set and tuple comprehensions is the same as list comprehensions, with the exception of what kind of brackets we use to surround the expression. For dictionary comprehensions, we need to include a `key: value` pair instead of a single element.

We can even make nested list comprehensions:

```
x = []
for outer in range(2):
    for inner in range(3):
        x.append((inner, outer))

print(x)  # [(0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1)]
y = [(inner, outer) for outer in range(2) for inner in range(3)]
print(y)  # [(0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1)]

# this may seem confusing that the loops are reversed, but think of it
# as keeping the same loop structure, and moving the element we're appending
# to the front:

y = [(inner, outer)
        for outer in range(2)
            for inner in range(3)]
```

We can even use nested list comprehensions to create nested structures:

```
x = []
for i in range(3):
    row = []
    for letter in "abcd":
        row.append(letter + str(i))
    x.append(row)
print(x) # [['a0', 'b0', 'c0', 'd0'], ['a1', 'b1', 'c1', 'd1'], ['a2', 'b2', 'c2', 'd2']]

# can also be written as
y = [[letter + str(i) for letter in "abcd"] for i in range(3)]
print(y)

# think of simplifying row into a list comprehension first:
x = []
for i in range(3):
    row = [letter + str(i) for letter in "abcd"]
    x.append(row)

# we can think of row as the value we want to create for each element in
# our list x [row for i in range(3)]
```

```
    # now substitute the actual list comprehension for row and you get
    # the one line version in y
```

Back to Top

I'd recommend being careful with nested list comprehensions, because like ternary statements they tend to be more difficult to debug and can make code harder to read. However, they're a tool commonly used by experienced Python programmers, so you're likely to see code written like this in the 'wild'. It's also good to think of different ways to write the same code, so we'll get some practice with writing code more concisely in this week's exercises. Additionally, this should be helpful for completing some of your 15.066 assignments that use Python.

6) Summary

In this reading, we introduced dictionaries and sets and saw a few sample programs involving them. We also learned some extra syntax that enables us to write code more concisely.

In this set of exercises, you'll get some practice with dictionaries and sets. In the next set of readings and exercises, we will dive even further into functions and importing other modules into our programs.

Back to exercises

Footnotes

[1] However, in earlier versions of Python than those we use in this class, you may have seen `{'cat': 'Katze', 'dog': 'Hund', 'guinea pig': 'Meerschweinchen'}` or `{'guinea pig': 'Meerschweinchen', 'dog': 'Hund', 'cat': 'Katze'}` — something with the key-value pairs in a different order than specified! In general, you should assume that the order of items in a dictionary is unpredictable. If you're using Python 3.6, for example, the preservation of order in dictionaries is merely a coincidental byproduct of how Python is implemented, and Python specifies that it should not be relied upon. Admittedly, newer versions of Python are moving toward stronger guarantees about the order of dictionaries. Python version 3.7 and above guarantees preservation of insertion order. Still, we think it's good practice to assume dictionaries are unordered. Or, if you are relying upon their order, to make that reliance explicit.

[2] Note that we've truncated the error messages to only include the last line that describes the error.