Dealing with Reality: Control Flow and Iterables

Introduction to Python

Data Sciences Institute, University of Toronto
Instructor: A Mahfouz | TA: Kaylie Lau
July 2022

Contents:

Logic

Logic Operators

Earlier, we introduced the `bool` data type, which has only two possible values: `True` and `False`. There are *boolean* or *logic operators* that work with Boolean values:

- `not`: negates the truth value of the statement
- `and`: check if the statements on both sides are true
- `or`: check if at least one of the statements are true

not

x	not x
True	False
False	True

In [1]:
```python
not True
```

Out[1]:

```
False
```

In [2]:
```python
3 == 3
```

Out[2]:

```
True
```

In [3]:
```python
not (3 == 3)
```

Out[3]:

```
False
```

and

Evaluates to `True` if both statements are true.

X	Y	X and Y
True	True	True
False	True	False
True	False	False
False	False	False

```
In [4]: 7 == 7.0 and 32 > 9
```

Out[4]:

```
True
```

```
In [5]: 'Python' == 'python' and True
```

Out[5]:

```
False
```

```
In [6]: is_summer = True
        is_sunny = True
is_summer and is_sunny
```

Out[6]:

```
True
```

`or`

Evaluates to `True` if just one of the statements is true.

X	Y	X or Y
True	True	True
False	True	True
True	False	True
False	False	False

In [7]:
```
'Python' == 'python' or True
```

Out[7]:

```
True
```

In [8]:
```
not (7 % 2 == 1) or False
```

Out[8]:

```
False
```

Operator precedence

Boolean operators are evaluated after arithmetic and comparison operators.

Order	Operator	Description
1	**	Exponentiation
2	-	Negation
3	*, /, //, %	Multiplication, division, integer division, and modulo
4	+, -	Addition and subtraction
5	<, <=, >, >=, ==, !=	Less than, less than or equal to, greater than, greater than or equal to, equal, not equal
6	not	Not
7	and	And
8	or	Or

Control Flow

What is control flow?

Control flow refers to the way computer execute programs. More specifically, control flow determines the order in which functions are called and statements are executed. Up until now, the code we have written has been more or less linear.

Conditionals

With a conditional statement, Python will run different lines of code depending on whether the condition is met -- in other words, whether the condition evaluates to `True`.

`if`

Conditional statements start with `if` followed by a condition. If the condition evaluates to `True`, the indented code block below the `if` statement runs. If the condition evaluates to `False`, the code block does not run.

In [9]:
```python
year = 2022

if year >= 2000:
    print('We are in the 21st century.')
```

```
We are in the 21st century.
```

`else`

We can use an `else` statement to tell Python what code to run if the condition evaluates to `False`. An `else` statement must always be paired with an `if` statement, but as we have seen, `if` statements do not need to be paired with `else`.

In [10]:
```
year = 1999

if year >= 2000:
    print('We are in the 21st century.')
else:
    print('We are not in the 21st century.')
```

```
We are not in the 21st century.
```

`elif`

We can evaluate several conditions one after another with `elif`, which is short for "else if". Conditions are checked in the order they appear. Python will execute the code block under the first `True` condition and skip subsequent `elif` and `else` statements after without evaluating them.

In [11]:
```
year = 1867

if year >= 2000:
    print('We are in the 21st century.')
elif year >= 1900:
    print('We are in the 20th century.')
elif year >= 1800:
    print('We are in the 19th century.')
elif year >= 1700:
    print('We are in the 18th century.')
else:
    print('We have gone way back in time!')
```

```
We are in the 19th century.
```

Building more complex conditionals with logical operators

We can use logical operators to check more complex conditions.

In [12]:
```python
day_of_week = 'Thursday'

if day_of_week == 'Saturday' or day_of_week == 'Sunday':
    print('Weekend')
else:
    print('Weekday')
```

```
Weekday
```

Nested conditionals

Conditionals can be nested within one another. This offers another way to test more complex conditionals. Whether to use conditions with logical operators, nested conditionals, or both can depend on personal preference and what we're trying to check.

Conditionals in functions

We can also use conditionals in functions to return different values.

Example: Will OHIP cover an eye exam?

OHIP covers eye exams **<u>in some cases</u>**. We can translate the eligibility criteria into conditionals.

```
In [13]: def eye_exam_covered(age, qualifying_condition, time_since_last_exam):
             if time_since_last_exam >= 12:
         if age <= 19 or age >= 65:
             return True
         elif qualifying_condition:
             return True
     else:
         return False
```

```
In [14]: eye_exam_covered(19, False, 11)
```

```
Out[14]:

 False
```

```
In [15]: eye_exam_covered(27, True, 15)
```

```
Out[15]:

 True
```

Practice

Conditionals

Write a conditional that:

- prints a warning message if a bank account balance is below $100
- prints different messages if a bank account balance is below $3,000, between \$3,000 and $10,000, or over \$10,000

Write a function:

- named `different` that takes two parameters, `a` and `b`. If `a` and `b` are different values, the function should return `True`. Otherwise it should return `False`.

Collections of values

Working with multiple values

So far, we've worked with individual values: integers, floating point numbers, strings, and booleans. However, we often want to work with groups of values
Python offers built-in data types to store and work with multiple values together. They are *lists*, *tuples*, *sets*, and *dictionaries*.

Lists

Python lists let us store and work with multiple values at once. We can create a list by putting values in square brackets ([ ])

```
In [16]: vowels = ['a', 'e', 'i', 'o', 'u']
         print(f'{vowels} are vowels.')
```

```
['a', 'e', 'i', 'o', 'u'] are vowels.
```

We can create an empty list by just using square brackets with nothing in them. It is also possible to create an empty list with the `list()` function, but this is not considered best practice.

```
In [17]: # create an empty list the conventional way
         empty_list = []
print('empty_list is', type(empty_list))

# this also works
empty_list2 = list()
print('empty_list2 is', type(empty_list2))
```

```
empty_list is <class 'list'>
empty_list2 is <class 'list'>
```

The values in a list can be different types. They can also repeat.

```
In [18]: # all valid lists!
         scores = [90, 80, 82, 91, 80]
grades = ['K', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
summary_functions = [len, sum, max, min]
```

We can even store lists within lists. The list below is written out over multiple lines so that it is easier to read.

```
In [19]: mystery_solvers = [
             ['Sherlock', 'Watson'],
    ['Scooby', 'Shaggy', 'Fred', 'Velma', 'Daphne'],
    'Poirot'
]
```

Accessing items in a list

Lists are *ordered*, which means that each item in a list can be referenced by its index, or position in the list. Just like getting characters from a string, we can get items from a list by index. We can also slice lists by providing the indices to start and end at in square brackets, separated by a colon. Negative indices count backwards from the end. If we try to access an item at an index that doesn't exist, we will get an error.

```
In [20]: # get the first school grade
         grades[0]
```

```
Out[20]:

'K'
```

```
In [21]: # get middle school grades
         grades[6:9]
```

```
Out[21]:

[6, 7, 8]
```

```
In [22]: # get high school grades
         grades[-4:]
```

```
Out[22]:

[9, 10, 11, 12]
```

```
In [23]: grades[13]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Input In [23], in <cell line: 1>()
----> 1 grades[13]

IndexError: list index out of range
```

List membership

We can check if a value is in a list with the `in` operator.

In [24]:
```
# recall the vowels list
vowels
```

Out[24]:

```
['a', 'e', 'i', 'o', 'u']
```

In [25]:
```
'e' in vowels
```

Out[25]:

```
True
```

Mutating lists

Lists are *mutable*, which means they can be modified in place. In contrast, data types like strings and numbers are *immutable*. They cannot be changed. When we update a string or numeric variable, we are actually replacing the value entirely.

To *mutate*, or change a value in a list, we access it by its index and assign the new value.

In [26]:
```python
perfect_squares = [1, 4, 9, 16, 25, 37, 49]

# fix the error
perfect_squares[5] = 36
perfect_squares
```

Out[26]:

```
[1, 4, 9, 16, 25, 36, 49]
```

Mutator beware

List variables behave in ways that can be surprising.

In [27]:
```
sandwich = ['bread', 'cheese', 'bread']
```

In [28]:
```
sandwich_copy = sandwich
```

In [29]:
```
# change original sandwich filling
sandwich[1] = 'ham'
```

In [30]:
```
sandwich_copy
```

Out[30]:

```
['bread', 'ham', 'bread']
```

What just happened?

When we assign a value to a variable, the value is stored somewhere in the computer's memory. This somewhere has an *address*. Python keeps track of the addresses where different variable values can be found.

When we assigned `sandwich` to `sandwich_copy`, we did not actually tell Python that the value of `sandwich_copy` is `['bread', 'cheese', 'bread']`. We told Python that the value of `sandwich_copy` can be found at the same memory address as the value of `sandwich`.

Remember how we said that lists mutate "in place"? The "in place" refers to a place in memory. When we updated `sandwich`, we updated the value stored at the memory address linked to both `sandwich` and `sandwich_copy`. As a result, `sandwich_copy` is now also a ham sandwich.

Mutating `sandwich_copy` will similarly update the value of `sandwich`.

Why doesn't this happen with string and numeric variables?

Because strings and numeric values are immutable, they cannot be changed in place. When we update a string or numeric variable, the memory address where the value is found changes.

In [32]:
```python
# Python tracks where in memory 1 is stored
a = 1

# Python will look for b's value at the same memory address as a's value
b = a

# 2 is stored at a new address. 1 is still stored at the old address
a = 2

# Python still looks to the old address a and b shared to find b's value
b
```

Out[32]:

```
1
```

Making an independent copy of a list

To make an independent copy of a list, we can pass the list we want to copy to the `list()` function.

In [33]:
```
combo = ['burger', 'fries', 'drink']
kid_meal = list(combo)
combo[0] = 'chicken sandwich'
kid_meal
```

Out[33]:

```
['burger', 'fries', 'drink']
```

Operations on lists

There are many ways to manipulate data in a list. Some produce summary statistics about the values in the list.

In [34]:
```
len(perfect_squares)
```

Out[34]:

```
7
```

In [35]:
```
max(perfect_squares)
```

Out[35]:

```
49
```

In [36]:
```
sum(perfect_squares)
```

Out[36]:

```
140
```

The `+` and `*` operators work on lists as well. `+` concatenates two lists.

In [37]:
```
letters = ['a', 'b', 'c']
numbers = [1, 2, 3]
characters = letters + numbers
characters
```

Out[37]:

```
['a', 'b', 'c', 1, 2, 3]
```

`*` repeats the list's items `int` times.

```
In [38]: letters * 2
```

Out[38]:

```
['a', 'b', 'c', 'a', 'b', 'c']
```

```
In [39]: numbers * 2
```

Out[39]:

```
[1, 2, 3, 1, 2, 3]
```

```
In [40]: letters
```

Out[40]:

```
['a', 'b', 'c']
```

Notice that `letters` did not change. `+` and `*` do not mutate lists.

List methods

Lists, like strings, have their own methods. Remember that methods are called with the pattern `value.method(arguments)`.
Almost all list methods modify lists in place. That is, they mutate them.

Adding items

We can add items to the end of a list with `append()` and `extend()`. `append()` takes one (and only one!) argument and tacks that value on to the end of a list.

In [41]:
```
rainbow = ['red', 'orange', 'yellow', 'green', 'light blue', 'blue', 'violet']
```

In [42]:
```
rainbow.append('purple')
rainbow
```

Out[42]:

```
['red', 'orange', 'yellow', 'green', 'light blue', 'blue', 'violet', 'purple']
```

In [43]:
```
# try appending a list
rainbow.append(['purple'])
rainbow
```

Out[43]:

```
['red',
 'orange',
 'yellow',
 'green',
 'light blue',
 'blue',
 'violet',
 'purple',
 ['purple']]
```

`extend()` also adds a single argument to the end of a list. Notice the difference -- it adds the argument value in pieces.

In [44]:
```
rainbow.extend(['magenta', 'pink'])
rainbow
```

Out[44]:

```
['red',
 'orange',
 'yellow',
 'green',
 'light blue',
 'blue',
 'violet',
 'purple',
 ['purple'],
 'magenta',
 'pink']
```

Strings get broken up into single characters.

```
In [45]: rainbow.extend('pale pink')
         rainbow
```

```
Out[45]:

['red',
 'orange',
 'yellow',
 'green',
 'light blue',
 'blue',
 'violet',
 'purple',
 ['purple'],
 'magenta',
 'pink',
 'p',
 'a',
 'l',
 'e',
 ' ',
 'p',
 'i',
 'n',
 'k']
```

And numbers don't work with `extend()` at all.

```
In [46]: rainbow.extend(2.3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [46], in <cell line: 1>()
----> 1 rainbow.extend(2.3)

TypeError: 'float' object is not iterable
```

What happens if we try to append data and assign the list to a new variable?

```
In [47]: new_rainbow = rainbow.append('dark purple')
         print(new_rainbow)
```

```
None
```

List methods that only mutate a list return `None`, or no data. The data we're looking for is in the original list.

Inserting items

If we want to add an item somewhere else to a list besides the end, we can use the `insert()` method, passing in the index to insert data into and what value to put in. Like the `append()` and `extend()`, `insert()` modifies the list in place.

```
In [48]: rainbow.insert(6, 'indigo')
         rainbow
```

```
Out[48]:

['red',
 'orange',
 'yellow',
 'green',
 'light blue',
 'blue',
 'indigo',
 'violet',
 'purple',
 ['purple'],
 'magenta',
 'pink',
 'p',
 'a',
 'l',
 'e',
 ' ',
 'p',
 'i',
 'n',
 'k',
 'dark purple']
```

Removing items

We can remove items by value with the `remove()` method. Notice that `remove()` only gets rid of the first match.

```
In [49]: rainbow.remove('p')
         rainbow
```

```
Out[49]:
```

```
['red',
 'orange',
 'yellow',
 'green',
 'light blue',
 'blue',
 'indigo',
 'violet',
 'purple',
 ['purple'],
 'magenta',
 'pink',
 'a',
 'l',
 'e',
 ' ',
 'p',
 'i',
 'n',
 'k',
 'dark purple']
```

We can also remove one or more items by index with the `del` operator.

In [50]:
```
# get rid of all the stuff we appended and extended
del rainbow[-13:]
rainbow
```

Out[50]:

```
['red', 'orange', 'yellow', 'green', 'light blue', 'blue', 'indigo', 'violet']
```

To empty a list out completely, we can `clear()` it.

In [51]:
```
rainbow.clear()
rainbow
```

Out[51]:

```
[]
```

Practice

Create a list, `books`, containing the following items: 'War and Peace', 'Pride and Prejudice', 'Mockingjay', 'Three Musketeers', 'The Adventures of Robinson Crusoe', 'Yevgeniy Onegin'.

1. Using slicing or indexing, create the following:
    - An empty list
    - The last item of `books`
    - List of three items: 'Three Musketeers', 'The Adventures of Robinson Crusoe', 'Yevgeniy Onegin'.

1. Using list methods:
    - Remove 'Pride and Prejudice' from the list.
    - Insert 'Harry Potter and the Chamber of Secrets' after 'Mocking Jay'.

Sorting lists

Lists are ordered, which means that they can be sorted. There are two ways to sort lists. Which way to use depends on if we want to modify the original list in place or if we want to make a brand new list.

It can be easier to follow code that creates a brand new list. Mutating the original list, on the other hand, is more efficient for large lists.

Modifying in place

The `sort()` method reorders a list's values in place and returns `None`.

In [52]:
```
fruits = ['pineapple', 'apple', 'kiwi', 'banana']
print(f'Output of sort(): {fruits.sort()}')
print(f'Original list: {fruits}')
```

```
Output of sort(): None
Original list: ['apple', 'banana', 'kiwi', 'pineapple']
```

Make a new sorted list

The `sorted()` function takes a list as an argument and returns a new list with sorted values.

```
In [53]: veggies = ['potato', 'celery', 'cabbage', 'bell pepper', 'onion']
         print(f'Output of sorted(): {sorted(veggies)}')
print(f'Original list: {veggies}')
```

```
Output of sorted(): ['bell pepper', 'cabbage', 'celery', 'onion', 'potato']
Original list: ['potato', 'celery', 'cabbage', 'bell pepper', 'onion']
```

Defining sorting criteria

Both `sort()` and `sorted()` take an optional `key` argument. We can pass any function name without parentheses to `key` depending on how we want to sort a list.

In [54]:
```
def last_letter(text):
    return text[-1]
```

In [55]:
```
sorted(veggies, key=last_letter)
```

Out[55]:

```
['cabbage', 'onion', 'potato', 'bell pepper', 'celery']
```

We can use our own functions to even sort nested lists.

In [56]:
```
students_per_class = [['Grade 9', 20], ['Grade 10', 17], ['Grade 11', 13], ['Grade 12', 22]]
```

In [57]:
```
def second_element(item):
    return item[1]
```

In [58]:
```
students_per_class.sort(key = second_element)
students_per_class
```

Out[58]:

```
[['Grade 11', 13], ['Grade 10', 17], ['Grade 9', 20], ['Grade 12', 22]]
```

Practice

Sorting in place

Sort the `colors` list below, keeping the original list unchanged.

```
colors = ['purple', 'black', 'maroon', 'mauve', 'aquamarine']
```

Custom sorting

Given the list `people`, sort it by people's first name, last name and age. Store the sorted lists under the following names: `by_first_name`, `by_last_name`, and `by_age`, respectively.

```
people = [('Mark', 'Harrison', 56), ('Ken', 'Wolseley', 23), ('Emily',
'Robinson', 77)]
```

Tuples

Tuples are a built-in data type similar to lists. Like lists, they are ordered collections of values. We can store multiple values in them, access values by index, slice them, and do things like calculate their length.

The key difference is that tuples are *immutable*: they cannot be changed once they are created. We cannot update a tuple to add, remove, replace, or reorder items in place. This makes tuples a good choice for storing values that should be read-only.

Creating tuples

We can create a tuple by surrounding values in parentheses.

In [59]:
```
mutable_synonyms = ('changeable', 'fluctuating', 'inconstant', 'variable')
mutable_synonyms
```

Out[59]:

```
('changeable', 'fluctuating', 'inconstant', 'variable')
```

To create an empty tuple, we can use either parentheses or the `tuple()` function. We can't add things to an empty tuple later!

```
In [60]: empty = ()
         type(empty)
```

```
Out[60]:

tuple
```

```
In [61]: also_empty = tuple()
         type(also_empty)
```

```
Out[61]:

tuple
```

```
In [62]: # not an actual tuple method
         empty.append('hi')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [62], in <cell line: 2>()
      1 # not an actual tuple method
----> 2 empty.append('hi')

AttributeError: 'tuple' object has no attribute 'append'
```

Working with tuples

Functions that work on lists *and do not modify the list in place* also work on tuples.

```
In [63]: len(mutable_synonyms)
```

```
Out[63]:

4
```

```
In [64]: sorted(mutable_synonyms)
```

```
Out[64]:

['changeable', 'fluctuating', 'inconstant', 'variable']
```

```
In [65]: mutable_synonyms + ('modifiable', 'shifting')
```

```
Out[65]:

('changeable',
 'fluctuating',
 'inconstant',
 'variable',
 'modifiable',
 'shifting')
```

```
In [66]: # check that mutable_synonyms hasn't changed
         mutable_synonyms
```

```
Out[66]:

('changeable', 'fluctuating', 'inconstant', 'variable')
```

Sets

Sets are another built-in data type. Like lists, they are mutable. Unlike lists and tuples, the items in a set are **unordered and distinct**. This makes them well-suited for cases where we do not want any duplicates in our data, like when de-duping a list or comparing unique values.

Creating Sets

We can create a set by surrounding values in curly braces.

In [67]:
```
things = {'coat', 'lock', 'box', 'book', 'apple', 'hair','xylophone', 'lock', 'book'}
things
```

Out[67]:

```
{'apple', 'book', 'box', 'coat', 'hair', 'lock', 'xylophone'}
```

In [68]:
```
# turn a list into a set to remove duplicates
visitor_post_codes = ['M5R', 'M5V', 'M1M', 'M1M', 'M1T']
set(visitor_post_codes)
```

Out[68]:

```
{'M1M', 'M1T', 'M5R', 'M5V'}
```

The only way to create an empty set is with the `set()` function.

In [69]:
```
empty_set = set()
empty_set
```

Out[69]:

```
set()
```

Working with sets

Sets are mutable, so we can add and remove items, and the set will be modified in place. This also means we have to be careful when setting one set equal to another -- modifying one means modifying both!

If an item is already in a set, it won't be duplicated.

In [70]:
```
# check for membership
'lock' in things
```

Out[70]:

```
True
```

In [71]:
```
# the set will not update
things.add('lock')
things
```

Out[71]:

```
{'apple', 'book', 'box', 'coat', 'hair', 'lock', 'xylophone'}
```

In [72]:
```
# notice where mirror appears in the set
things.add('mirror')
things
```

Out[72]:

```
{'apple', 'book', 'box', 'coat', 'hair', 'lock', 'mirror', 'xylophone'}
```

In [73]:
```
things.remove('apple')
things
```

Out[73]:

```
{'book', 'box', 'coat', 'hair', 'lock', 'mirror', 'xylophone'}
```

Since sets are unordered, we cannot slice them or access items by index.

```
In [74]: things[1]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [74], in <cell line: 1>()
----> 1 things[1]

TypeError: 'set' object is not subscriptable
```

Set operations

There are some operations that are unique to sets. A *union* combines two sets to get the unique values in both. An *intersection* finds the values two sets have in common. A *symmetric difference* finds the values that are in only one of two sets. And *difference* finds the values in the first set that are not in the second set.
Each operation has a corresponding set method.

```
In [75]: rainbow = {'red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'}
         olympic_flag = {'red', 'green', 'yellow', 'blue', 'black'}
```

```
In [76]: print(f'In the rainbow but not the Olympic flag: {rainbow.difference(olympic_flag)}')
         print(f'In the Olympic flag but not the rainbow: {olympic_flag.difference(rainbow)}')
```

```
In the rainbow but not the Olympic flag: {'violet', 'indigo', 'orange'}
In the Olympic flag but not the rainbow: {'black'}
```

```
In [77]: print(f'Only in one: {rainbow.symmetric_difference(olympic_flag)}')
```

```
Only in one: {'violet', 'black', 'orange', 'indigo'}
```

```
In [78]: print(f'Common colours : {rainbow.intersection(olympic_flag)}')
```

```
Common colours : {'green', 'blue', 'yellow', 'red'}
```

```
In [79]: print(f'All colours: {rainbow.union(olympic_flag)}')
```

```
All colours: {'violet', 'yellow', 'orange', 'green', 'black', 'indigo', 'red', 'blue'}
```

Dictionaries

Dictionaries are our last collection of data. They store data in *key:value* pairs and let us look up values by key. Dictionaries, like lists, are ordered and mutable. Every key in a dictionary is unique.

Creating dictionaries

To make a dictionary, we place curly braces around key:value pairs. Keys can be any immutable data type -- strings, numbers, booleans, and tuples all work. Values can be any data type.

In [80]:
```
capitals = {'Canada': 'Ottawa',
            'United States': 'Washington, D.C.',
        'Mexico': 'Mexico City'}
capitals
```

Out[80]:

```
{'Canada': 'Ottawa',
 'United States': 'Washington, D.C.',
 'Mexico': 'Mexico City'}
```

In [81]:
```
olympics_cities = {2020: 'Tokyo', 2016: 'Rio de Janiero', 2012: 'London'}
olympics_cities
```

Out[81]:

```
{2020: 'Tokyo', 2016: 'Rio de Janiero', 2012: 'London'}
```

We can nest dictionaries within dictionaries.

In [82]:
```
all_olympics_hosts = {'summer': olympics_cities,
                      'winter': {2022: 'Beijing', 2018: 'Pyeongchang'}}
all_olympics_hosts
```

Out[82]:

```
{'summer': {2020: 'Tokyo', 2016: 'Rio de Janiero', 2012: 'London'},
 'winter': {2022: 'Beijing', 2018: 'Pyeongchang'}}
```

The preferred way to create an empty dictionary is with curly braces, but the `dict()` function also works.

In [83]:
```
empty_dictionary = {}
type(empty_dictionary)
```

Out[83]:

```
dict
```

In [84]:
```
still_empty = dict()
type(still_empty)
```

Out[84]:

```
dict
```

Accessing, adding, and updating dictionary items

We access the content of a dictionary by specifying the dictionary, then the key to look up in square brackets. To access nested values, we *chain* together bracket selections.

```
In [85]: olympics_cities[2016]
```

```
Out[85]:

'Rio de Janiero'
```

```
In [86]: all_olympics_hosts['winter'][2018]
```

```
Out[86]:

'Pyeongchang'
```

Trying to access a key that doesn't exist results in an error.

```
In [87]: olympics_cities[2014]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Input In [87], in <cell line: 1>()
----> 1 olympics_cities[2014]

KeyError: 2014
```

The `get()` method provides a safe way to work with dictionary values. It takes the name of the key to look up and the default value to create a new key:value pair if the key does not exist.

In [88]:
```
olympics_cities.get(2004, 'Athens')
olympics_cities
```

Out[88]:

```
{2020: 'Tokyo', 2016: 'Rio de Janiero', 2012: 'London'}
```

We can check to see if a key is in a dictionary with `in`.

In [89]:
```
2016 in olympics_cities
```

Out[89]:

```
True
```

In [90]:
```
# 'in' looks for matching keys
'Rio de Janiero' in olympics_cities
```

Out[90]:

```
False
```

If we assign a value to a key that doesn't exist, the key:value pair will be added to the dictionary. If we assign a value to a key that already exists, the value for that key will be updated.

In [91]:
```
olympics_cities[2008] = 'Barcelona'
olympics_cities
```

Out[91]:

```
{2020: 'Tokyo', 2016: 'Rio de Janiero', 2012: 'London', 2008: 'Barcelona'}
```

In [92]:
```
# fix 2008's city
olympics_cities[2008] = 'Beijing'
olympics_cities
```

Out[92]:

```
{2020: 'Tokyo', 2016: 'Rio de Janiero', 2012: 'London', 2008: 'Beijing'}
```

Mutations, mutations

Notice that updating a dictionary will also change other variables that reference it! Let's take a look at our `all_olympics_hosts` dictionary.

In [93]:
```
all_olympics_hosts
```

Out[93]:

```
{'summer': {2020: 'Tokyo',
  2016: 'Rio de Janiero',
  2012: 'London',
  2008: 'Beijing'},
 'winter': {2022: 'Beijing', 2018: 'Pyeongchang'}}
```

Deleting dictionary items

To remove a key:value pair from a dictionary, we can use the `del` operator.

In [94]:
```
del olympics_cities[2020]
olympics_cities
```

Out[94]:

```
{2016: 'Rio de Janiero', 2012: 'London', 2008: 'Beijing'}
```

Dictionary methods

Python dictionaries have methods for getting keys, values, and items (that is, key:value pairs). This is useful for getting all dictionary keys, checking for values in a dictionary, and, as we'll see soon, working with keys, values, and items one-by-one.

```
In [95]: all_olympics_hosts.keys()
```

```
Out[95]:

dict_keys(['summer', 'winter'])
```

```
In [96]: if 'London' in olympics_cities.values():
             print('London was a host city')
else:
    print('London was not a host city')
```

```
London was a host city
```

```
In [97]: # get keys and values for the nested winter dictionary
         all_olympics_hosts['winter'].items()
```

```
Out[97]:

dict_items([(2022, 'Beijing'), (2018, 'Pyeongchang')])
```

Practice

Write a function called `dict_intersect` that takes two dictionaries, `d1` and `d2`, as arguments and returns a set that contains only the values found in both of the original dictionaries.

Collections: a summary

(Adapted from: Table 17, Chapter 11, *Practical Programming: An Introduction to Computer Science Using Python 3.6*)

Collection	Mutable?	Ordered?	Use when...
`str`	No	Yes	You want to keep track of text.
`list`	Yes	Yes	You want to keep track of and update an ordered sequence.
`tuple`	No	Yes	You want to build an ordered sequence that you know won't change or that you want to use as a key in a dictionary or as a value in a set.
`set`	Yes	No	You want to keep track of values, but order doesn't matter, and you don't want duplicates. The values must be immutable.
`dict`	Yes	No	You want to keep a mapping of keys to values. The keys must be immutable.

Control Flow: Iteration

What are iteration and loops?

Earlier, we saw how to control the flow of a program through `if`/`elif`/`else` statements, which tell Python to run or skip blocks of code depending on whether a condition is met. We can also tell Python to repeat code in a loop for a certain number of times or until a condition is met, a technique called *iteration*. For example, we may want to manipulate every item in a list individually. Copy/pasting code for each item is inefficient and error-prone. Instead, we can use one of Python's two loops: `for` loops or `while` loops.

`for` loops

A `for` loop runs an indented block of code for every item in an *iterable* -- a data type like a list, tuple, set, dictionary, or even string. When setting up a `for` loop, we have to specify a variable name to refer to individual items by. Try to pick one that makes sense, but if you're in a rush, `i` (for index) is conventional.

In [98]:
```
for vowel in vowels:
    print(f'Give me an {vowel}!')
```

```
Give me an a!
Give me an e!
Give me an i!
Give me an o!
Give me an u!
```

If we simply want to run a block of code *n* number of times, we can use the `range()` function to create an iterable to loop over.

```
In [99]: for i in range(7):
             print(i, i*2)
```

```
0 0
1 2
2 4
3 6
4 8
5 10
6 12
```

We can use loops to build new lists (and other iterables).

```
In [100]: input_files = ['data_01.csv', 'data_02.csv', 'data_03.csv', 'data_04.csv']
          output_files = []

for i in input_files:
    output_file_name = 'processed_' + i.replace('.csv', '.xlsx')
    output_files.append(output_file_name)

output_files
```

```
Out[100]:

['processed_data_01.xlsx',
 'processed_data_02.xlsx',
 'processed_data_03.xlsx',
 'processed_data_04.xlsx']
```

Looping with multiple values

It is often useful to iterate over more than one value at once, such as when working with functions like `enumerate()` and methods like `dict.items()`, which give us index:value and key:value pairs, respectively. Because these methods give us two values at once, we need to supply two looping variables. The returned value pairs are *unpacked* to our variables.

In [101]:
```
stops = ['Sheppard-Yonge', 'Bayview', 'Bessarion', 'Leslie', 'Don Mills']
for idx, stop in enumerate(stops):
    print(f'Stop {idx + 1} is {stop}.')
```

```
Stop 1 is Sheppard-Yonge.
Stop 2 is Bayview.
Stop 3 is Bessarion.
Stop 4 is Leslie.
Stop 5 is Don Mills.
```

In [102]:

```python
# double a list in place
numbers = [1, 10, 100, 1000]
for idx, val in enumerate(numbers):
    numbers[idx] = val * 2
numbers
```

Out[102]:

```
[2, 20, 200, 2000]
```

Looping over two iterables at once

To loop over more than one iterable at the same time, we can `zip()` them up. Note that if the iterables are different lengths, we won't get the "extra" values in the longer iterable.

In [103]:
```
lats = (43.650, 45.520, 49.280)
lons = (-79.380, -73.570, -123.130)

for i, j in zip(lats, lons):
    print(f'({i}, {j})')
```

```
(43.65, -79.38)
(45.52, -73.57)
(49.28, -123.13)
```

Loops within loops

We can nest loops within each other, indenting once more each time. The variables from the higher-level loop are available at the lower levels.
One thing to keep in mind is that the number of times code runs increases very quickly with nested loops -- slightly longer iterables can mean a longer-running program than expected!

In [104]:
```
for key, value in all_olympics_hosts.items():
    for year, city in value.items():
        # this way of formatting strings is new
        print(f'The {year} {key.title()} Olympics were in {city}.')
```

```
The 2016 Summer Olympics were in Rio de Janiero.
The 2012 Summer Olympics were in London.
The 2008 Summer Olympics were in Beijing.
The 2022 Winter Olympics were in Beijing.
The 2018 Winter Olympics were in Pyeongchang.
```

```
In [105]: def print_table(n):
              """Print the multiplication table for numbers 1 through n inclusive.
    >>> print_table(3)
      1 2   3
    1 1 2   3
    2 2 4   6
    3 3 6   9
    """
    # The numbers to include in the table.
    numbers = list(range(1, n + 1))
    # Print the header row.
    for i in numbers:
        print(f'\t{i}', end='')
    # End the header row.
    print()
    # Print each row number and the contents of each row.
    for i in numbers:
        print (i, end='')
        for j in numbers:
            print(f'\t{i * j}', end='')
        # End the current row.
        print()
```

```
In [106]: print_table(5)
```

```
        1       2       3       4       5
1       1       2       3       4       5
2       2       4       6       8       10
3       3       6       9       12      15
4       4       8       12      16      20
5       5       10      15      20      25
```

`while` loops

What if we aren't sure how many times code needs to run, but we know how to tell when we're done? In that case, we can use a `while` loop, which runs an indented block of code until a condition is met.

In [107]:
```python
countdown = 4

while countdown > 0:
    print(countdown)
    countdown -= 1
```

```
4
3
2
1
```

Infinite loops

What happens if we omit the last line of code in the countdown example? The countdown never changes, so it never hits zero, and our program keeps printing "4". We've just created an *infinite loop*.
(**NOTE**: If you try this, you will need to interrupt the program. In Anaconda, press `ctrl+c` on your keyboard or go to **Kernel --> Interrupt** in the toolbar. In Colab, press `ctrl+m+i` or go to **Runtime --> Interrupt execution** in the toolbar. You may want to try this in a new notebook.)

In [108]:
```
# uncomment the lines below to run
#countdown = 4

#while countdown > 0:
#    print(countdown)
```

Infinite loops are sometimes necessary. They are used extensively in gaming or to run a connection to a server, for example. To create an intentional infinite loop, we make the `while` condition `True`.

`break`ing free

A `break` statement interrupts the execution of a loop.

In [109]:
```python
countdown = 4

while countdown > 0:
    print(countdown)
    if countdown == 3:
        print('We are breaking the loop early.')
        break
    countdown -=1

print('Done iterating.')
```

```
4
3
We are breaking the loop early.
Done iterating.
```

Even infinite loops can be exited.

```
In [110]: while True:
              password = input("What's the password? ")
    # case-insensitive comparison
    if password.lower() == 'open sesame':
        print("You're in!")
        break
```

```
What's the password? password
What's the password? let me in
What's the password? please
What's the password? open sesame
You're in!
```

Please `continue`...

Lastly, we can interrupt a loop with a `continue` statement, which tells Python to leave the current iteration of the loop and start back up at the top

```
In [111]: wishes = 3
          while wishes > 0:
    wish = input('Make a wish: ')
    if 'infinite wishes' in wish.lower():
        print('You can\'t do that!')
        continue
    else:
        print('Wish granted.')
    wishes -= 1
print('You have used all your wishes.')
```

```
Make a wish: infinite money
Wish granted.
Make a wish: never get rained on
Wish granted.
Make a wish: infinite wishes
You can't do that!
Make a wish: no more allergies
Wish granted.
You have used all your wishes.
```

Practice

Write a loop that iterates over the two lists below simultaneously. For each pair of values, print the first number divided by the second. The loop should keep running when it encounters a zero divisor.

```
dividends = [100, 37.5, -12]
divisors [8, 0, -3]
```

References

- Bostroem, Bekolay, and Staneva (eds): "Software Carpentry: Programming with Python" Version 2016.06, June 2016, **https://github.com/swcarpentry/python-novice-inflammation**, 10.5281/zenodo.57492.
- Chapter 8, 9, and 11, Gries, Campbell, and Montojo, 2017, *Practical Programming: An Introduction to Computer Science Using Python 3.6*
- "Modules", Python Software Foundation, *Python Language Reference, version 3.* Available at **https://docs.python.org/3/tutorial/modules.html**.