Dictionaries and Sets, Types

CSE/IT 107L

NMT Department of Computer Science and Engineering

"Vague and nebulous is the beginning of all things, but not their end."

— K. Gibran

"The danger that computers will become like humans is not as big as the danger that humans will become like computers." ("Die Gefahr, dass der Computer so wird wie der Mensch ist nicht so groß, wie die Gefahr, dass der Mensch so wird wie der Computer.")

— Konrad Zuse

"I don't need to waste my time with a computer just because I am a computer scientist."

— Edsger W. Dijkstra

Introduction

This lab concludes the introduction to Python's data structures. Dictionaries are one of Python's fundamental data structures. They behave like lists, but you can use strings, tuples, floats or any set of integers as indices to Python values. Dictionaries are useful for storing elements that are best identified by a description or name rather than by a strict order. Sets are collections of unique (non-repeated) values. They support fast combination operations, like finding the elements that appear in two sets. We will define the *types* of Python values and describe how to convert between different types.

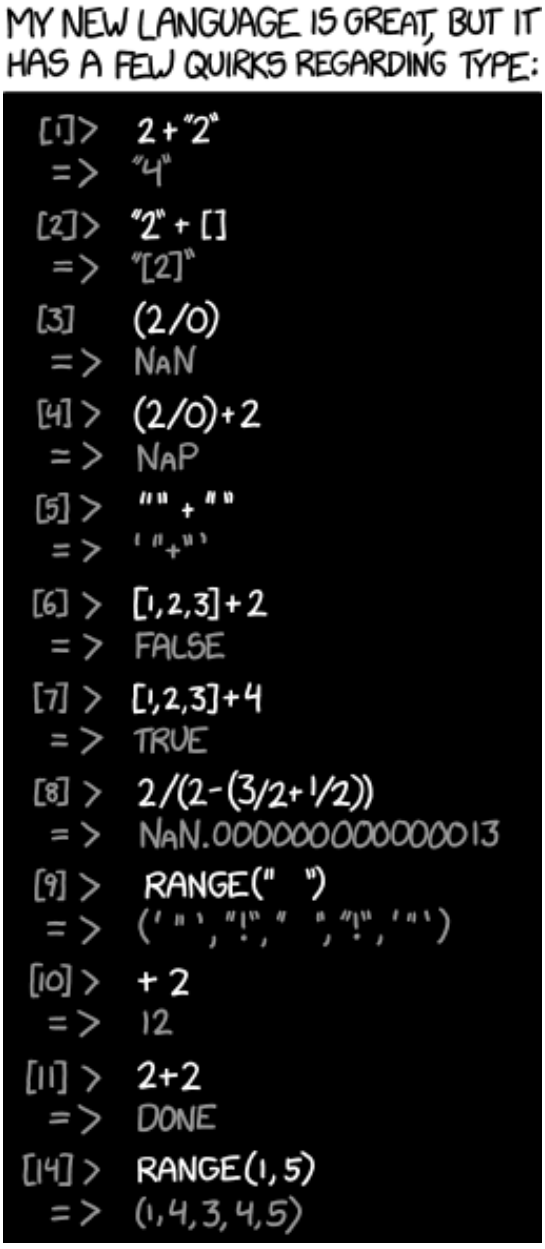

Figure 1: `http://xkcd.com/1537`

Contents

1 Represent Relationships Between Values Using Dictionares

You will often need to define relationships between pairs of Python values. For example, a dataset of names of movies and their ratings could be represented using strings (movie names) that are related to floats (movie ratings). As another example, you could have time measurements (floats) related to another measurement (a float or integer). One of this lab's exercises involves counting the number of letters in the user's input. In this case, you could relate one-letter strings to integers.

1.1 Creating Dictionaries

How do you represent relationships between the two groups of values? Python offers dictionaries to help you perform this task. They are a fundamental data structure and are also known as associative arrays, hash maps, or key-value pairs. Some examples:

```
1 movie_ratings = {"Sharknado 4": 2.0, "Spirited Away": 4.0, "The Big Lebowski": 4.0}
2 temperatures = {0.0: 75.2, 0.5: 79.2, 1.0: 80.3, 1.5: 81.0, 2.0: 88.5, 2.5: 98.6}
3 letter_count = {"e": 99, "t": 92, "a": 90, "q": 2}
4 moons = {"venus": [], "earth": ["moon"], "mars": ["phobos", "deimos"]}
```

Listing 1: Example dictionaries.

A dictionary is written by surrounding a list of comma-separated key-value pairs with curly braces. Key-value pairs are a Python value that represents the key, then a colon, and a Python value that represents the value.

```
1 dictionary = {key1: value, key2: value}
```

The values in a dictionary can be any Python value: strings, integers, booleans, lists of any Python value, other dictionaries, etc.. The keys in a dictionary must not be repeated and must be immutable Python values: these include strings, ints, floats, tuples of any of these, and some other values.

Figure 2 visualizes the dictionaries given in the previous code sample. Notice that the order of key-value pairs is not preserved. Different keys can be related to the same value. For example, there are two keys in the `movie_ratings` dictionary that have the value 4.0. Values can be lists, but keys cannot be values.

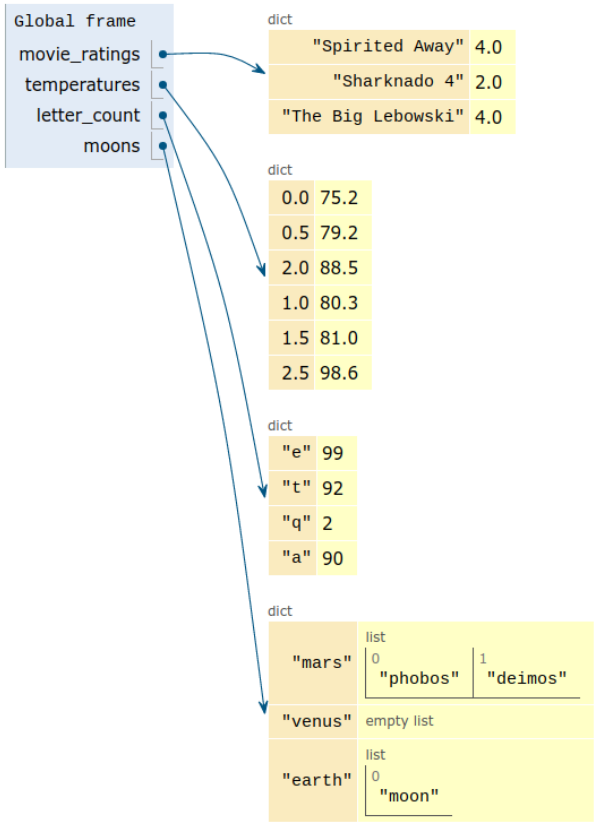

Figure 2: A visualization of the dictionaries in listing 1 created using `pythontutor.com`

Python raises a `TypeError` if there's a dictionary with mutable key. Keys can't be repeated: in this case, Python ignored all the key-value pairs with key set 1, except for the last one. The following session from a Python interactive shell demonstrates the definition of both valid and invalid dictionaries:

```
>>> {1: 'a', 2: 'm', 3: 'z'}
{1: 'a', 2: 'm', 3: 'z'}
>>> {'a': 1, 'm': 2, 'z': 3}  # order is not preserved
{'a': 1, 'z': 3, 'm': 2}
>>> {1: [1, 2, 3], 4: [9, 10]}
{1: [1, 2, 3], 4: [9, 10]}
>>> {[1, 2, 3]: 1, [9, 10]: 4}  # lists can't be keys
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> {1: 1, 1: 2, 1: 3}  # only the last is kept
{1: 3}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

1.2 Indexing by Key to Access, Modify, and Add Dictionary Values

Similar to a list, you may access values in the dictionary by indexing on the dictionary's keys to access elements. For example:

```
>>> letter_count = {"e": 99, "t": 92, "a": 90, "q": 2}
>>> "the letter {} occurs {} times".format("e", letter_count["e"])
'the letter e occurs 99 times'
>>> moons = {"venus": [], "earth": ["moon"], "mars": ["phobos", "deimos"]}
>>> moons["venus"]
[]
>>> moons["mars"]
['phobos', 'deimos']
>>> moons["jupiter"]
Traceback (most recent call last):
  File "stdin", line 1, in <module>
    moons["jupiter"]
KeyError: 'jupiter'
Traceback (most recent call last):
  File "stdin", line 1, in <module>
    moons["jupiter"]
KeyError: 'jupiter'
```

Notice, Python raises an `IndexError` whenever you try to access a key that doesn't appear in the dictionary.

Just like lists, the values to a dictionary key can be reassigned by using indexing with assignment. In the following example, the value of the key `'breakfast'` is changed to `'toast'`. You can also add key-value pairs by assigning to a key that isn't in the dictionary. In the example, the key `'brunch'` is added and it has value `'biscuits'`.

```
>>> food = {'breakfast': 'burrito', 'lunch': 'burger', 'dinner': 'tacos'}
>>> print(food)
{'lunch': 'burger', 'breakfast': 'burrito', 'dinner': 'tacos'}
>>> food['breakfast'] = 'toast'  # add a new key!
>>> print(food)
{'dinner': 'tacos', 'lunch': 'burger', 'breakfast': 'toast'}
>>> food['brunch'] = 'biscuits'
>>> food
{'dinner': 'tacos', 'brunch': 'biscuits', 'lunch': 'burger', 'breakfast': 'toast'}
```

1.3 Safely Removing Key-Value Pairs Using `dict.pop(key, defaultvalue)`

To remove a key-value pair, use the `dict.pop(key, defaultvalue)` function. This function removes the given key from the dictionary if it exists, if it does not, then it returns the given default value. For example:

```
>>> letter_count = {"e": 99, "t": 92, "a": 90, "q": 2}
>>> letter_count["e"]
99
>>> removed_value = letter_count.pop("e", 0)
>>> removed_value
99
>>> letter_count
{'t': 92, 'a': 90, 'q': 2}
>>> removed_value2 = letter_count.pop("z", 0)  # key 'z' not in dictionary
>>> removed_value2
0
```

1.4 Checking Dictionary Properties Using `len`, `in`, `==`

The `in` keyword tests whether an item is a key in a dictionary. To find the number of key-value pairs in a dictionary, use the `len` function. Dictionary equality can be tested using the boolean operator `==`.

```
>>> states = {'NM' : 'New Mexico', 'TX' : 'Texas', 'KS' : 'Kansas'}
>>> states2 = {'NM' : 'New Mexico', 'TX' : 'Texas', 'KS' : 'Kansas'}
>>> states3 = {'NY' : 'New York', 'TX' : 'Texas', 'KS' : 'Kansas'}
>>> 'NM' in states
True
>>> 'NY' in states
False
>>> len(states)
3
>>> states == states2
True
>>> states == states3
False
```

1.5 Safe Access Using `dict.get(key, defaultvalue)`

The `dict.get(key, defaultvalue)` function is used to access a dictionary without the risk of Python raising a `KeyError` if the element does not exist. If the given key is not present in the dictionary, the function will return the given default value instead of causing an error. A sample:

```
>>> moons = {"venus": [], "earth": ["moon"], "mars": ["phobos", "deimos"]}
>>> moons.get("earth", [])
['moon']
>>> moons["jupiter"]
Traceback (most recent call last):
  File "stdin", line 1, in <module>
    moons["jupiter"]
KeyError: 'jupiter'
>>> moons.get("jupiter", [])
[]
Traceback (most recent call last):
  File "stdin", line 1, in <module>
    moons["jupiter"]
KeyError: 'jupiter'
```

1.6 Traversing Dictionaries using `dict.items()`, `dict.keys()`, `dict.values()`

You can use a for-loop to iterate over the keys, values, or key-value pairs of a dictionary.

The `dict.items()` function returns something similar to a list of tuples where the first element of each tuple is the key of a dictionary item and the second element is the corresponding value. The `dict.values()` and `dict.keys()` functions return a list of the dictionary's values or a list of its keys. Any of these three functions can be used to iterate through a dictionary. For example,

```
>>> states = {'NM' : 'New Mexico', 'TX' : 'Texas', 'KS' : 'Kansas'}
>>> states.keys()
dict_keys(['TX', 'KS', 'NM'])
>>> states.values()
dict_values(['Texas', 'Kansas', 'New Mexico']
>>> states.items()
dict_items([('KS', 'Kansas'), ('TX', 'Texas'), ('NM', 'New Mexico')])
>>> for state_short, state_name in states.items():
...     print(state_short, state_name)
KS Kansas
TX Texas
NM New Mexico

>>> for state_short in states.keys():
...     print(state_short, states[state_short])
KS Kansas
TX Texas
NM New Mexico

>>> for state_name in states.values():
...     print(state_name)
```

```
Texas
Kansas
New Mexico
```

If you need the iteration to happen in order, you can sort dictionary keys and then iterate through the dictionary using `for key in sorted(d.keys())`. Otherwise, iteration order is not enforced.

1.7 Sample Program

```
states = {'NM' : 'New Mexico', 'TX' : 'Texas', 'KS' : 'Kansas'}
capitals = { # multiline dictionaries are easier to read
  'NM' : 'Santa Fe',
  'TX' : 'Austin',
  'KS' : 'Kansas City',
}

state = input('Enter a state >>> ')
if state in states:
    print('You selected {}. The state capital is {}.'.format(states[state],
        capitals[state]))
else:
    print('The state you selected is not known to this program.')
```

Notice the dictionary `capitals` was written on multiple lines for readability.

2 Sets: Collections With Unique Elements

2.1 Creating Sets

Sets are a lot like lists, but *no element can appear twice* and a set's elements are *unordered*. Additionally, a set cannot contain mutable Python values (like lists). Sets are mutable.

A set is made using curly braces by converting a list:

```
>>> a = {5, 5, 4, 3, 2} # duplicates ignored
>>> print(a)
{2, 3, 4, 5}
>>> b = set([5, 5, 4, 3])
>>> print(b)
{3, 4, 5}
```

Because sets are unordered, they cannot be indexed. That means you cannot index a set using the `[]` operator.

```
>>> a = {5, 4, 3, 2}
>>> a[1]
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

2.2 Combining Sets

The | operator can be used to join two sets together. This creates a new set that contains the elements in each of the two sets. This is called the *union* of the two sets. Since the result is also a set, duplicate values will only appear once.

```
>>> a = {1, 2, 4, 8, 16}
>>> b = {1, 3, 9, 27}
>>> a | b
{1, 2, 3, 4, 8, 9, 16, 27}
```

The & operator can be used to find the *intersection* of two sets. This is the set containing all elements that are in both sets.

```
>>> a = {2, 4, 6, 8, 10, 12}
>>> b = {3, 6, 9, 12, 15, 18}
>>> a & b
{12, 6}
```

The - operator can be used to find the *difference* of two sets. This set contains all of the elements of the first set excluding the elements in the second set.

```
>>> a = {2, 4, 6, 8, 10, 12}
>>> b = {3, 6, 9, 12, 15, 18}
>>> a - b
{8, 2, 10, 4}
```

The ^ operator can be used to find the *symmetric difference* of two sets. This is the set containing all elements in either of the original sets, but not in both.

```
>>> a = {'one', 'eight'}
>>> b = {'two', 'four', 'six', 'eight'}
>>> a ^ b
{'one', 'six', 'two', 'four'}
```

Operator	Meaning	Definition
`a | b`	in either	union of `a` and `b`
`a & b`	in both	intersection of `a` and `b`
`a - b`	all not in `b`	difference of `a` and `b`
`a ^ b`	all elements not in common	symmetric difference of `a` and `b`

2.3 Conversions Using `set()`, `list()`

A list can be converted into a set using the `set()` function:

```
>>> a = [1, 5, 2, 2.3, 5, 2]
>>> set(a) # duplicate removed!
{1, 2, 2.3, 5}
```

Similarly, a set can be converted into a list using the `list()` function:

```
>>> a = {1, 2, 2, 5, 3, 2}
>>> a
{1, 2, 5, 3}
>>> list(a)
[1, 2, 3, 5]
```

2.4 Adding Elements Using `set.add()`

To add an element to a set, use the `set.add(element)` function. A sample:

```
>>> a = {1, 2, 5, 3}
>>> a.add(5)
>>> a
{1, 2, 3, 5}
>>> a.add(7)
>>> a
{1, 2, 3, 5, 7}
```

2.5 Removing Elements Using `set.remove()` or `set.pop()`

To remove an element that is present in the set, use the `set.remove(element)` function. To remove an arbitrary element from the set, use the `set.pop()` function. Python raises a `KeyError` if the set is empty or the element is not in the set. A sample:

```
>>> a = {1, 2, 5, 3}
>>> an_element = a.pop()
>>> an_element, a
(1, {2, 3, 5})
>>> a.remove(5)
>>> a
{2, 3}
>>> a.remove(10)
Traceback (most recent call last):
  File "<doctest lab7.tex[76]>", line 1, in <module>
    a.remove(10)
KeyError: 10
Traceback (most recent call last):
  File "<doctest lab7.tex[76]>", line 1, in <module>
```

```
    a.remove(10)
KeyError: 10
```

2.6 Traversing Sets With For Loops

You may also iterate over a set using a for-loop. Order of iteration is unspecified.

```
>>> companions_nine = {'rose', 'jack'}
>>> companions_ten = {'rose', 'mickey', 'jack', 'donna', 'martha', 'wilf'}
>>> for companion in companions_nine:
...     print(companion)
jack
rose
```

2.7 Creating an Empty Set and an Empty Dictionary

Note that empty curly braces create an empty dictionary. To create an empty set, use the `set()` function:

```
>>> type({})
<class 'dict'>
>>> type(set())
<class 'set'>
```

3 Ordered Access With Stacks

Imagine a stack of plates. You can only add plates to the top and remove plates from the top. This a "Last-In-First-Out" data structure: If you add three plates to the stack, the last one you added will be the first one you remove. *Stacks* are a specialized data structure. They are at the heart of every operating system and used in many pieces of software.

A stack is similar to a Python list where you can only add elements to the end or remove elements from the end. This is accomplished using the `list.pop()` and `list.append(element)` methods on lists. When given no arguments, the pop method will remove the last element of the list and return it. The append method will add an element to the end of the list. The array index `list[-1]` is used to inspect the top of the stack; or the last element of the list. For example:

```
>>> stack = [1, 2, 3]
>>> a = stack.pop()
>>> print(a)
3
>>> print(stack)
[1, 2]
>>> stack.append(4)
>>> print(stack)
[1, 2, 4]
```

```
>>> stack[-1]  # top!
4
```

Interacting with the elements of the list in any other way violates the idea of the list being a stack. By adhering to this restriction, we can effectively invent a new data structure based on existing Python data structures. There are more effective ways of creating data structures for organizing data according to specific needs. For example, you could compose two data structures. If you needed a dictionary with ordered key-value pairs, you could use a list of tuples where the first element in the tuple is the key and the second is the value.

4 List Comprehensions: Concise Iteration

4.1 Modifying Collections

List comprehensions can be used to create new lists by doing something to each element of a currently existing list or range. For example, if we want to make every string in a list of strings lowercase we could use a for-loop to call `.lower()` on each string.

```
>>> strings = ['Python', 'N.M.']
>>> result = []
>>> for string in strings:
...     result.append(string.lower())
>>> result
['python', 'n.m.']
```

List comprehensions are a more concise way of doing this.

```
>>> strings = ['Python', 'N.M.']
>>> result = [string.lower() for string in strings]
>>> print(result)
['python', 'n.m.']
```

We can process more complicated elements with list comprehensions. If we have a list of dictionaries containing lists, we can find the maximum element in each sublist by using this list comprehension:

```
>>> data = [{'d': [1,2,3]}, {'d': [4,5]},
...         {'d': [0,0,1]}]
...
>>> [max(dictionary['d']) for dictionary in data]
[3, 5, 1]
```

4.2 Filtering Elements

What if we want to exclude elements? In this code, we take a list of words and keep all words of length less than 2.

```
>>> strings = ['a', 'ab', 'abb', 'aab', 'aaa']
>>> result = []
>>> for string in strings:
...     if len(string) <= 2:
...         result.append(string)
...
>>> result
['a', 'ab']
```

This can be written more concisely by using an `if` statement within a list comprehension.

```
>>> strings = ['a', 'ab', 'abb', 'aab', 'aaa']
>>> [x for x in strings if len(x) <= 2]
['a', 'ab']
```

4.3 Nested Comprehensions

We can also write multiple `for` statements and `if` statements in a list comprehension. This example creates a list of floating point numbers by dividing two integers.

```
>>> result = [y / x for x in range(3) if x != 0 for y in range(3)]
>>> print(result)
[0.0, 1.0, 2.0, 0.0, 0.5, 1.0]
```

These statements should be read from right to left. So the previous list comprehension is the same as these nested statements:

```
result = []
for x in range(10):
    if x != 0:
        for y in range(10):
            result.append(y / x)
```

5 Types

In Python, every value has a type. We have already seen a few types in action: integers, floating-point numbers, strings, and boolean values, lists, tuples, sets, and dictionaries. The type of a value or variable restricts what it can represent. Here is a list of some types and example values.

Name	Description	Examples	Convert y
Int	Integer, whole number	`1`, `123`, `-12`	`int(y)`
Float	Floating-point number	`1.0`, `3.1415`, `-0.01`	`float(y)`
String	Sequence of characters	`""`, `"1"`, `"abc 123\n"`	`str(y)`
Boolean	True or false	`True`, `False`	`bool(y)`

Most programming languages have types for a good reason: for one, operations (such as +, -, ...) have different effects on different types. For example, an integer * an integer results in an integer (the multiplication of the two *operands*), but an integer * a string results in the string repeated. However, a string * a string results in an error:

```
>>> 5*3
15
>>> 5*'hi'
'hihihihihi'
>>> 'hi'*'hi'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

Also, Python can speed up your programs by picking efficient ways of dealing with different types. Multiplying integers and multiplying floating point numbers require very different techniques. Python may optimize code if the type of the number is known.

5.1 Types of Collections

In the previous labs, we have described and demonstrated several data structures. Each of them has its own type.

Name	Description	Examples	Convert y
List	Sequence of values	`[]`, `[0, 1]`, `[True, True]`	`list(y)`
Tuple	Immutable sequence of values	`(0, 1)`, `('a', 'b')`	`(y)`
Dictionary	Relationship between values	`{}`, `{1:2}`, `{'a':True}`	`dict(y)`
Set	Collection of unique values	`{0,1}`, `{True}`	`set(y)`

5.2 Checking Types with the `type()` Function

The `type()` function returns the type of a value.

```
>>> type(5)
<class 'int'>
>>> x = 10
>>> type(x)
<class 'int'>
>>> type('Allons-y!')
<class 'str'>
>>> type(True)
<class 'bool'>
```

5.3 Converting Values to Different Types

There are several functions that can convert the types of Python values. Only values have types, variables can store values of any type.

```
>>> float("5.5") # convert string with 5.5 to a float
5.5
>>> int("5") # integer containing 5
5
>>> str(5) # string containing 5
'5'
>>> # Note that an impossible conversion will throw an error
>>> int("55a")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '55a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '55a'
```

There are several functions that work on different types of collections. The `len()` function can be used on any collection object in order to find the number of elements. All collections can be iterated over with `for` loops, All collections can use the `in` keyword to test if an element is in that collection. For dictionaries, this will compare against the list of keys, not the values.

Data Structure	Mutable	Mutable elements	Indexing	Ordered	Other properties
List []	yes	yes	by whole numbers	yes	can contain elements more than once
Sets {}	yes	no	not indexable	no	no element can appear more than once
Tuples ()	no	yes	by whole numbers	yes	can contain elements more than once
Dictionary {}	yes	yes	by anything "hashable" (immutable collections or basic types)	no	

Table 1: Summary of Data Structures in Python

Tuples, sets, and lists can all be freely converted from one to another using the `tuple()`, `set()`, `list()`, `str()`, and `dict()` functions.

```
>>> food = ('nothing', 'cereal', 'lemonheads')
>>> list(food)
['nothing', 'cereal', 'lemonheads']
>>> str(food) # not particularly useful
"('nothing', 'cereal', 'lemonheads')"
>>> s = "The cat in the"
```

```
>>> list(s)
['T', 'h', 'e', ' ', 'c', 'a', 't', ' ', 'i', 'n', ' ', 't', 'h', 'e']
>>> tuple(s)
('T', 'h', 'e', ' ', 'c', 'a', 't', ' ', 'i', 'n', ' ', 't', 'h', 'e')
```

5.4 Converting Sequences to Strings With `str.join()`

Note `str()` cannot undo the action of `list()` on a string. That is, `str(list("ABC"))` is not equal to `"ABC"`. Use `"".join(list("ABC"))` for that.

To add spaces between the elements of a list, change the string at the beginning.

```
>>> "".join(['c', 'a', 't'])
'cat'
>>> " ".join(['c', 'a', 't'])
'c a t'
```

The elements of the sequence can be of any type.

```
>>> ' '.join(("dinosaurs", "roamed the", "earth"))
'dinosaurs roamed the earth'
>>> print('\n> '.join(("dinosaurs", "roamed the", "earth")))
dinosaurs
> roamed the
> earth
```

6 Sample Program

```
import turtle

# All units in kilometers.
planets8 = [
    {"name": "mercury", "radius": 2439.7, "color": "lightgrey", "moons": []},
    {"name": "venus", "radius": 6051.8, "color": "orange", "moons": []},
    {"name": "earth", "radius": 6371, "color": "blue",
     "moons": [{"name": "moon", "radius": 1737.4, "color": "lightgrey"}]},
    {"name": "mars", "radius": 3389.5, "color": "red",
     "moons": [{"name": "phobos", "radius": 11.1, "color": "grey"},
               {"name": "deimos", "radius": 6.3, "color": "grey"}]},
    {"name": "jupiter", "radius": 69911, "color": "orange",
     "moons": [{"name": "ganymede", "radius": 2631.2, "color": "grey"},
               {"name": "callisto", "radius": 2410.3, "color": "darkgrey"},
               {"name": "io", "radius": 1830, "color": "lightgreen"},
               {"name": "europa", "radius": 1560.8, "color": "green"}]},
    {"name": "saturn", "radius": 58232, "color": "yellow",
     "moons": [{"name": "titan", "radius": 2574, "color": "orange"},
               {"name": "rhea", "radius": 763.5, "color": "grey"},
               {"name": "iapets", "radius": 734.3, "color": "darkgrey"},
```

```
                {"name": "dione", "radius": 561.4, "color": "grey"}]},
    {"name": "uranus", "radius": 25362, "color": "lightblue",
     "moons": [{"name": "titania", "radius": 788.4, "color": "grey"},
               {"name": "oberon", "radius": 761.4, "color": "grey"},
               {"name": "umbriel", "radius": 584.7, "color": "darkgrey"},
               {"name": "ariel", "radius": 578.9, "color": "grey"}]},
    {"name": "neptune", "radius": 24622, "color": "blue",
     "moons": [{"name": "triton", "radius": 1352.6, "color": "salmon"},
               {"name": "proteus", "radius": 210, "color": "grey"},
               {"name": "nereid", "radius": 170, "color": "grey"},
               {"name": "larissa", "radius": 97, "color": "grey"}]},  # :(
]

def draw_body(body, zoom):
    """This function draws a filled-in circle based on the 'radius' and
    'color' keys of the dictionary `body`. The drawing is scaled by the
    factor `zoom`."""
    turtle.dot(2 * body["radius"] * zoom, body["color"])

def separate_bodies(body, prev_body_radius, zoom, sep):
    """This function moves the turtle the appropriate distance to avoid
    overlap between two drawings. Minimal separation given by `sep`."""
    distance = (body["radius"] + prev_body_radius) * zoom + sep
    turtle.forward(distance)
    return distance

def draw_planets(system, zoom):
    separation = 20  # Distance between planets.
    moon_separation = 10  # Distance between moons and their planets.

    prev_planet_radius = 0
    for index, planet in enumerate(system):
        if index > 0:
            prev_planet_radius = system[index - 1]["radius"]
        separate_bodies(planet, prev_planet_radius, zoom, separation)
        draw_body(planet, zoom)

        # Draw moons
        turtle.left(90)
        distance = 0  # How far has the turtle moved to draw the moons?
        prev_radius = planet["radius"]
        for index2, moon in enumerate(planet["moons"]):
            if index2 > 0:
                prev_radius = planet["moons"][index2 - 1]["radius"]
            distance += separate_bodies(moon, prev_radius, zoom, moon_separation)
            draw_body(moon, zoom)
        turtle.forward(-distance)
        turtle.right(90)

def draw_system(system, zoom):
    # Program inputs.
    print('Welcome to the 107L2 Radially-Precise Planetarium!')
    turtle.tracer(False)

```

```
    turtle.bgcolor('black')  # Black background
    turtle.penup()  # Hide pen

    # Draw (2 * decoration) ** 2 stars.
    decoration = 15
    scale = max(turtle.window_height(), turtle.window_width())
    for col in range(-decoration, decoration):
        for row in range(-decoration, decoration):
            # row / decoration gives is a number between -1 and 1
            # the `scale` variable determines the space between stars
            y = row / decoration * scale
            # shift every other row of stars by 50%
            x = ((col + 0.5 * (row % 2)) / decoration) * scale

            turtle.goto(x, y)

            # make every other star's radius larger
            turtle.dot(1 + col % 2, 'white')
    turtle.goto(0, 0)

    # Draw planets starting near the screen's edge.
    turtle.forward(-turtle.window_width() / 2 * 0.9)
    draw_planets(system, zoom)

    turtle.done()

if __name__ == "__main__":
    draw_system(planets8, 0.003)
```

7 Exercises

Exercise 7.1 (lettercount.py).
Write program that reads in a string on the command line and returns a table of the letters of the alphabet in alphabetical order which occur in the string together with the number of times each letter occurs. Case should be ignored. A sample run of the program would look this:

```
Enter some letters >>> The cat in the hat
a 2
c 1
e 2
h 3
i 1
n 1
t 4
```

This should involve writing a function called `count_letters` that takes in a string and returns a dictionary with these letters and counts.

Supplementary Files

The file `test_letter_count.py` is available on Canvas. Open it to see more examples.

Exercise 7.2 (date2.py, horoscope2.py).
You will use dictionaries to write more readable and compact versions of `date.py` and `horoscope.py`. The requirements for both files have changed, please read the following text.

Write a program `date2.py`, which should take in a month and day of the month, and convert it to the corresponding day of the year. Assume that leap years exist. For incorrect dates the conversion function should return `-1`, and you should print an error from `main`.

For example:

```
python3 date2.py
Type stop or enter a month, day, and year.
> March, 14, 2015
Day of the year: 73
> December, 14, 2016
Day of the leap year: 349
> Oktober, 91, 2015
Invalid date entered.
> stop
```

Use the following boolean expression to check if a year is a leap year or not:

```
is_leap = (year % 4 == 0 and year % 100 != 0) or year % 400 == 0
```

Next, the `horoscope2.py` program should take in a month and a day. It should print an error message if the date is invalid, otherwise it should print the horoscope of someone who was

born on that day. You should use the `import` statement to make use of the code you wrote in `date.py` for determining the user's astrological sign. The horoscopes themselves are completely up to you, as long as they start with the correct sign.

For example:

```
python3 horoscope2.py
Type stop or enter a month, day, and year.
> March, 14, 2015
Pisces: Tui and La, push and pull, commit and rebase...
> May, 5, 1848
Taurus: today you must choose between earth and sea, C and miniKanren...
> February, 29, 1847
Invalid date entered.
> stop
```

For a reference on the zodiac dates, see the "Tropical zodiac" column from this table: `https://en.wikipedia.org/wiki/Zodiac#Table_of_dates`. Watch out for Capricorn! That zodiac spans from December 22nd to January 20th, which for this program means that it will have two different date ranges, 1 to 20 and 356 to 365.

Exercise 7.3 (rpn_calculator.py).
Write a reverse Polish notation calculator. In reverse Polish notation (also known as HP calculator notation), mathematical expressions are written with the operator following its operands. For example, $3 + 4$ becomes 3 4 +.

Order of operations is entirely based on the ordering of the operators and operands. To write $3 + (4 * 2)$, we would write 4 2 $*$ 3 + in RPN. The expressions are evaluated from left to right.

A longer example of an expression is this:

$$5\ 1\ 2\ /\ 4\ *\ +\ 3\ -$$

which translates to

$$5 + ((1/2) * 4) - 3$$

If you were to try to "parse" the RPN expression from left to right, you would probably "remember" values until you hit an operator, at which point you take the last two values and use the operator on them. In the example expression above, you would store 5, then 1, then 2, then see the division operator (/) and take the last two values you stored (1 and 2) to do the division. Then, you would store the result of that (0.5) and encounter 4, which you store. When you encounter the multiplication sign (*), you would take the last two values stored and do the operation $(4 * 0.5)$ and store that.

Following this through step by step, the steps would look something like this (the bold number is the most recently computed value):

1. 5 1 2 / 4 * + 3 -
2. 5 **0.5** 4 * + 3 -
3. 5 **2** + 3 -

4. **7** 3 -
5. **4**

Writing this algorithm for evaluating RPN in pseudo code, we get:

1. Read next value in expression.
2. If number, store.
3. If operator:
   (a) Remove last two numbers stored.
   (b) Do operation with these last two numbers.
   (c) Store the result of the operation as last number.

If you keep repeating this algorithm, you will eventually just end up with one number stored unless the RPN expression was invalid.
Your task is to write an RPN calculator which asks the user for an RPN expression and prints the result of that expression. You *must* use a stack (see Section **??**). The RPN algorithm has to be in a separate function (not main). You need to support the four basic operators (+, -, *, and /).
You should detect and display messages for the following errors:

- Operand is used when not enough numbers are stored.
- More or less than one number stored after the expression is evaluated.

Please see the example input and output below for expressions you can test with.

RPN Expression	Output
`5 1 2 / 4 * + 3 -`	4
`312 999 +`	1311
`4 2 + 1 5 + * +`	Not enough operands for +.
`2 100 3 * 5 + 2 2 2 + + * *`	3660

Note that you need to support multi-digit numbers. You cannot expect all input to be single digits.

Index of New Functions and Methods

Submitting

You should submit your code as a tarball that contains all the exercise files for this lab. The submitted file should be named

`cse107_firstname_lastname_lab7.tar.gz`

Upload your tarball to Canvas.

List of Files to Submit

Exercises start on page 16.