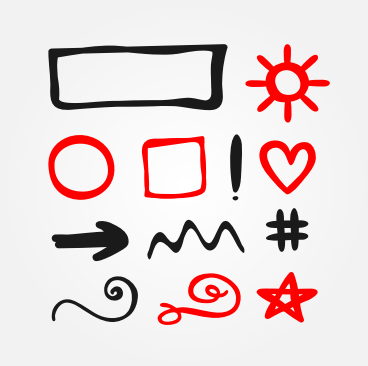

CPSC 5061—Programming I for Business

Worksheet 11

Lists

Motivation

Tuples were so successful in Python—a collection datatype that supports:

- elements of any datatype,
- indexing,
- slicing,
- for-loop iteration,
- membership with the `in` operator,
- knowing its length with the `len` built-in,
- literal constructions using parentheses, and
- concatenation with the **+** and ***** operators.

Powerful!

Just one problem. Tuples are *immutable*. That means you can create one, but once created it can never change. We can keep creating new and bigger ones by concatenating smaller ones. And we can create smaller ones by slicing. But what we cannot do is change one. There are many circumstances where we would like to.

Just to drive this point home, look at the following code from our loop in Worksheet 10, rewritten out with the **+=** expanded so we can see what's happening clearly:

```
children = children + (child,)
```

Python executes this statement by first creating the one-element tuple with the string in the variable **`child`**. Then it concatenates the tuple currently in **`children`** with this new one-element tuple. The resulting new tuple with one more element than what **`children`** had before now replaces the **`children`** variable. So, to append a single element, since we can only use concatenation, we created a brand-new tuple one bigger and discarded the old tuple.

So, for example, in the course of building up a 1000-element tuple this way, we'd have to create 1000 one-element tuples and 1000 various-sized tuples (each of 0-, 1-, 2-, …, 1000-elements) just to get our desired data structure which is a sequence of 1000 elements. This is very inefficient and a bit convoluted when all we wanted to do was to repeatedly append an extra element to the already constructed sequence. For this reason, the Python designers invented a new datatype like tuples which is mutable—a *list*. (We still want the immutable tuples for other reasons, so they couldn't just "fix" tuples to be mutable.)

Concept

The generalization of the `tuple` datatype that is *mutable* is the `list` datatype. Lists do everything that tuples do, but you can *also* modify them by:

- appending an element with the **append** method,
- removing an element with the **pop** method, or
- changing an element by using an index expression on the left-hand side of the = sign.

List Literals

List literals are just like tuple literals except we use the square bracket characters to enclose the elements instead of the parentheses.

```
t = (1, 3.8, 'girl')  # literal tuple with three elements
v = [1, 3.8, 'girl']  # literal list with three elements
```

Like a tuple, a list can have any object and so in a list we can also use any Python expressions to calculate each element's value.

```
n = 2
v = [n, n * n, n**3]  # the list [2, 4, 8]
```

Python evaluates these expressions and then constructs the list.
Just like we can have an empty string or tuple, we can have an empty list with no elements. The literal for this is just an empty pair of square brackets.

```
v = []  # v is now the empty list
```

Since there is no possible confusion between using square brackets for indexing/slicing and list construction, there is no need for a *nonsensical comma* with lists. The one-element list is just the expression for the element enclosed in square brackets.

```
v = [68]  # v is a now a list with a single element, 68
```

Indexing, Slicing, and Length

Like strings, ranges, and tuples, lists can be indexed and sliced using the square bracket notation you are already familiar with. They can also reveal their element count with the built-in `len`.

```
>>> v = [1, 99, -4, 16, 491]
>>> v[0]
1
>>> v[-1]
491
```

```
>>> v[:2]
[1, 99]
>>> v[2::2]
[-4, 491]
>>> len(v)
5
>>> [len(v), len(v[2::2]), len(v[:2]), len([]), len([1])]
[5, 2, 2, 0, 1]
```

Modifications

To add an element onto the end of an existing list, use the **append** method:

```
>>> v = [1, 2, 'hello']
>>> v.append(4.3)
>>> v
[1, 2, 'hello', 4.3]
```

To replace an existing element, use indexing on the *left-hand side* of the = sign.

```
>>> v[2] = 'George'
>>> v
[1, 2, 'George', 4.3]
```

Looping

Of course, the looping patterns of tuples work equally well for lists, but to avoid the inefficiency of repeated concatenation and discarding in a tuple-building loop, we want to use the **append** method:

```
print("What are your children's names?")
children = []  ### start with an empty list
next_child = input()
while next_child != "":
    children.append(next_child)  ### modify existing list
    next_child = input()
```

There is only ever one object referred to by children and that one object is repeatedly extended with the **append** method. There is never any use of concatenation.

Reading a list in a for loop is just like a tuple. It is the exact same code.

```
print("Your", len(children), "children's names are:")
for child in children:
    print(child)
```

Aliasing

When two variables refer to the same object, it is called *aliasing*. All variables in Python when assigned merely refer to the object on the right of the = sign. And when the right-hand side of the assignment is just another variable, after the assignment both variables refer to the same object.

For example, given these two assignments:

```
a = 'abcdef'
b = a
```

Both **a** and **b** refer to the same object, the string **'abcdef'**.

So far, we've only dealt with immutable datatypes and so aliasing, while it happened wasn't particularly interesting. For example, here using tuples:

```
>>> a = (1, 2, 3)
>>> b = a         # now a and b both refer to the same tuple, (1, 2, 3)
>>> b += (4, 5)
>>> b             # now b refers to a different tuple, (1, 2, 3, 4, 5)
(1, 2, 3, 4, 5)
>>> a             # but a still refers to the old tuple, (1, 2, 3)
(1, 2, 3)
```

This is suddenly more interesting when an object can be modified. If the object is aliased through other variables, their values change, too, since they are all referring to the exact same object.

```
>>> a = [1, 2, 3]
>>> b = a         # now a and b both refer to the same list, [1, 2, 3]
>>> b.append(4)   # add a four onto the end of list reffered to by b
>>> b[0] = 19     # modify the first element of that object to 19
>>> b             # b still refers to the same (now modified) object
[19, 2, 3, 4]
>>> a             # and a still refers to it, too!
[19, 2, 3, 4]
```

In fact, we can even ask Python if two things are the same object with the **is** operator.

```
>>> a is b
True
```

We put aliasing to powerful use by using mutable datatypes for function arguments and by placing mutable datatypes as elements of other data structures like tuples and lists.

Mutable Objects as Arguments

Imagine we had a sequence of numbers and we wanted to tack on the sum of the last two numbers as the new last element in the sequence (like the Fibonacci sequence).

```
def append_sum_of_last_two(seq):
    """Append sum of last two numbers in seq to seq."""
    new_tail = seq[-2] + seq[-1]
    seq.append(new_tail)

>>> numbers = [1, 1, 2, 3, 5]
>>> append_sum_of_last_two(numbers)
>>> numbers
[1, 2, 2, 3, 5, 8]
>>> append_sum_of_last_two(numbers)
>>> numbers
[1, 2, 2, 3, 5, 8, 13]
>>> append_sum_of_last_two(numbers)
>>> numbers
[1, 2, 2, 3, 5, 8, 13, 21]
```

Exercise

Note that this would be impossible with tuples.

```
def try_with_tuples(tup):
    """This doesn't work!"""
    new_tail = tup[-2] + tup[-1]
    tup += (new_tail,)  # this just changes tup to another object
```

Fill in below what would happen:

```
>>> numbers = (1, 1, 2, 3, 5)
>>> try_with_tuples(numbers)
>>> numbers


>>> try_with_tuples(numbers)
>>> numbers


>>> try_with_tuples(numbers)
>>> numbers
```

Exercises

1. Write a function, **floatify**, that takes a list as an argument. It modifies every **int** element to a **float** and returns **True** if all the elements are now of type **float**, **False** if there are elements of other datatypes. Remember, you can check if a number is an **int** like so:

```
if type(number) == int:
    print(number, 'is an integer')
```

Here's an example run:

```
>>> a = [1, 2, 3]
>>> floatify(a)
True
>>> a
[1.0, 2.0, 3.0]
```

2. Write another function, **purify**, that takes a sequence, **seq**, as an argument (say, either a **list** or a **tuple**) and returns a **tuple** with all the numbers from **seq** as integers.

```
>>> a = [1, 2, 'abc', 3.7]
>>> purify(a)
(1, 2, 3)
>>> a  # should be unchanged
[1, 2, 'abc', 3.7]
```