Module 4: Strings and lists are iterables

Exercise

If you have not already, get prepared for class by downloading the start code:

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/module-04-start.ipynb
```

Discuss the previous module with your neighbour.

- How do we write code to do something repeatedly?
- How do we create functions that take a function as an argument?

Sequences

In mathematics, a **sequence** is a collection of items, in some order. Some examples:

- The natural numbers: $0, 1, 2, 3, 4, \ldots$
- The prime numbers: $2, 3, 5, 7, 11, 13, 17, 19, 23 \ldots$
- The Collatz sequence starting at 3 and ending at 1: $3, 10, 5, 16, 8, 4, 2, 1$

In mathematics sequences are often infinitely long, but not always.

In programming we often work with sequences of finite length like $[2, 4, 6, 0, 1]$.

Two kinds of iterables: strings and lists

A string such as `"foobar"` is a collection of characters, in some order:
the first item is `'f'`, the second is `'o'`, the third is another `'o'`, and so on.

`"foobar"` is like a sequence of letters.

Another way to have a collection of values is to make a **list**. In Python, we do this by writing the values, separated by commas, inside square brackets `[]`.

For example, `[2, 4, 6, 0, 1]` is a list that contains five integers:
the first item is `2`, the second is `4`, the third is `6`, and so on.

`[2, 4, 6, 0, 1]` is like a sequence of integers.

Indexing

We can extract a single item from either of these iterables using **indexing**.

After a value we write square brackets around an integer called an **index**.

```
word = "foobar"
word[0] ⇒ "f"
word[1] ⇒ "o"
word[2] ⇒ "o"
word[3] ⇒ "b"
word[4] ⇒ "a"
word[5] ⇒ "r"
```

```
jvj = [2, 4, 6, 0, 1]
jvj[0] ⇒ 2
jvj[1] ⇒ 4
jvj[2] ⇒ 6
jvj[3] ⇒ 0
jvj[4] ⇒ 1
```

! Notice: the first value is item 0, not item 1.
The last item is numbered 1 less than the length.
The index of a slot indicates how many slots there are **before** it.

Indexing

`rhg = ["Everything", "is", "theoretically", "impossible,", "until", "it", "is", "done."]`

What is `rhg[2]` ?
`rhg[2]` ⇒ `"theoretically"`

But notice: `rhg[2]` is a **`str`**, and we can also index a **`str`**.
`rhg[2][0]` ⇒ `"t"`

Ex. What is `rhg[3][0]` ? `rhg[0][3]` ?

Ex. How many different ways can you use indexing on `rhg` to get `"y"` ?

Evaluation Principle: if an expression evaluates to something that we could use in a certain way, we can use the expression in that way.

Evaluation Principle: if an expression evaluates to something that we could use in a certain way, we can use the expression in that way.

Exercise

```
mannie = [[12, 13, 14], [15, 16, 17], [18, 19, 20]]
```

Use indexing on `mannie` to get `17`.

Exercise

```
wyoh = [["The", "five"], ["boxing", "wizards"], ["jump", "quickly"]]
```

- Use indexing on `wyoh` to get `"x"`.
- Use indexing on `wyoh` to get `"w"`.

Type annotations for lists

To state the type of a list, we say `list`, then inside square brackets, a **single** expression indicating the type of the values that are in the list. Some examples:

- The type of `[2, 4, 6, 0, 1]` is `list[int]`, since each value is an `int`.
- The type of `[3.14, 2.718, 1.414, 2.0]` is `list[float]`, since each value is a `float`.
- The type of `["we're", "all", "fine", "here"]` is `list[str]`, since each value is a `str`.
- The type of `[[2,3], [4,5,6], [7]]` is `list[list[int]]`, since each value is a `list[int]`.

We **could** have a list that contains a mix of a few types, like `[1, "word", 4, "you"]`, which contains some `int` and some `str`. We're going to avoid this; it's usually a bad idea.

If we want to talk about a list where the values could be of **any** type, we can say `list[any]`.

Length and walking using `while`

We can use the built-in function `len` to determine how many values an iterable contains:

`len("foobar")` ⇒ `6`

`len([2, 4, 6, 0, 1])` ⇒ `5`

A `while` loop using `len` and a variable index can extract items one at a time:

```
i = 0
word = "foobar"
while i < len(word):
    print(i, word[i])
    i = i + 1
## We see:
0 f
1 o
2 o
3 b
4 a
5 r
```

```
j = 0
jvj = [2, 4, 6, 0, 1]
while j < len(jvj):
    print(j, jvj[j])
    j = j + 1
## We see:
0 2
1 4
2 6
3 0
4 1
```

Using a `for` loop

We often want to want to walk through an iterable.
Often we just need the values, not the counter.

To make it easier, Python provides the `for` loop.
It steps through the sequence, one item at a time, and sets a variable to each item.

```
word = "foobar"
for letter in word:
    print(letter)
## We see:
f
o
o
b
a
r
```

```
jvj = [2, 4, 6, 0, 1]
for number in jvj:
    print(number)
## We see:
2
4
6
0
1
```

The first time through the loop, the variable takes the first value in the iterable; the second time through, it takes the second value, and so on.

Syntax of `for` loops

```python
word = "foobar"
for letter in word:
    print(letter)
```

```python
jvj = [2, 4, 6, 0, 1]
for number in jvj:
    print(number)
```

The syntax of `for` has some similarities to the syntax of `if` and `while`, and some new parts.

We write:

1. the keyword `for`,
2. the name of a variable,
3. the keyword `in`,
4. a sequence,
5. a colon,
6. an indented block of code.

The block of code will run repeatedly, with the variable taking a value from the sequence each time.

Example: using `for` with `if`

We can use our tools together. Consider:

```
def drop_e(word: str) -> None:
    """Print all the letters in word except e."""
    for letter in word:
        if letter != 'e':
            print(letter)
```

Then `drop_e("djent")` will print:

```
d
j
n
t
```

Exercise

Write a function `count_e(word: str) -> int`, that counts how many times `'e'` appears in `word`. For example,

```
check.expect("CE1", count_e("hello"), 1)
check.expect("CE2", count_e("able was I ere I saw Elba"), 3)
```

Exercise

Write a function `count_n(target: float, vals: list[float]) -> int`, that counts how many times `target` appears in `vals`. For example,

```
check.expect("Cn1", count_n(3.1, [2.5, 6.5, 3.1, 1.0]), 1)
check.expect("Cn2", count_n(2.1, [2.5, 2.1, 3.1, 2.1, 1.0, 2.1]), 3)
```

It's common to want to check if some value is contained, somewhere, inside a `str` or `list`.

For example, does `[2,4,6,0,1]` contain the number 6? By looping through the list, one item at a time, we eventually find the target; so the list does contain a 6, and we can return `True`. We don't even need to look at the 0 and 1.

Does `[2,4,6,0,1]` contain the number 7? Again, we loop through, and reach the end of the loop, without ever finding the target. So it does not contain it; we can return `False`.

Exercise

Use a `for` loop to write a function `contains(target: int, collection: list[int]) -> bool`. The function shall return `True` if `target` appears in `collection` at least once.

```
check.expect("C6", contains(6, [2,4,6,0,1]), True)
check.expect("C7", contains(7, [2,4,6,0,1]), False)
```

Something neat: the same code works for a `str`:

```
check.expect("Cy", contains("y", "too many geese"), True)
check.expect("Cx", contains("x", "too many geese"), False)
```

The built-in operator `in` does the same thing

In the previous exercise you wrote code like this.

```
def contains(target: int, collection: list[int]) -> bool:
    """Return True if collection contains target, and False otherwise."""
    for item in collection:
        if item == target:
            return True
    return False
```

The `in` operator does the same!

`6 in [2,4,6,0,1]` ⇒ `True`

`7 in [2,4,6,0,1]` ⇒ `False`

`"y" in "too many geese"` ⇒ `True`

`"x" in "too many geese"` ⇒ `False`

To create a Boolean expression using `in`, we write:

1. a value,
2. `in`,
3. an iterable such as a `str` or `list`.

Exercise

Without using `for` or `if`, write a 1-line function `is_vowel(ch)` that takes a string of length 1, and determines if it is a vowel (one of *a*, *e*, *i*, *o*, *u*, *A*, *E*, *I*, *O*, *U*).

Note there are two ways to use the keyword `in`: by itself as above, or as part of a `for` loop.

Algorithm design: Finding the largest

Exercise

Find the largest value in this list:
`[45,27,46,27,69,48,66,49,77,75,15,84,49,53,87,61,32,72,23,37,12,80,79,58,47,19,81]`

Exercise

Now think about your thinking: how did you find it?

Any answer to this question is an **algorithm**: an explanation of how to solve a problem.

As programmers, a big part of our work is

- identifying/inventing the right algorithm, and
- turning the algorithm into working code.

Let's take our rough description and try to make it a bit more precise.

We might describe our algorithm to find the largest item in a list as:

> "Create a variable that stores the 'largest item seen so far'; set it to the first item from the list (or some other item).
> Then look at each item in turn; if the new item is larger than the largest item seen so far, update the largest item seen so far."

Now we have a detailed algorithm; let's turn it in to code.

(Note: there is a built-in function `max`. We want to understand the algorithm, so we are not going to use it.)

Exercise

Write a function `longest(items: list[str])` that returns the longest value in `items`.

```
check.expect("W", longest(["a", "bee", "was", "on", "a", "green", "leaf"]), "green")
```

Comparing the values directly with `<` doesn't work; that gives us `"was"`.

It takes only a very small change: before comparing, transform each value using `len`.

Fancy indexing: slicing

So far we have only indexed an iterable of length L using an integer $i : 0 \le i < |L|$:

```
word = "foobar"                         jvj = [2, 4, 6, 0, 1]
word[3] ⇒ "b"                           jvj[0] ⇒ 2
```

This is enough for many purposes, but a useful trick is to take a **slice** of a `str` or `list`:

- Using a **negative index** counts from the back.
  `[-1]` gives the last item, `[-2]` the second last, and so on:
  ```
  word[-1] ⇒ "r"    word[-6] ⇒ "f"    jvj[-2] ⇒ 0    jvj[-5] ⇒ 2
  ```
- Using a single **colon** like `[start:stop]` where the first item is item `start`, stopping just **before** `stop`. If either value is omitted, go to that end:
  ```
  word[1:4] ⇒ "oob"    word[4:] ⇒ "ar"    jvj[:2] ⇒ [2,4]    jvj[1:-1] ⇒ [4,6,0]
  ```
- Using two `colons` like `[start:stop:step]`, as above, but we skip values (and move backwards with negative `step`).
  ```
  word[::2] ⇒ "foa"  word[1::2] ⇒ "obr"  word[2:5:2] ⇒ "oa"  word[::-1] ⇒ "raboof"
  ```

Making iterables with +

We can "join" two numbers together using the + operator, making a new number:

`2 + 37 + 3 ⇒ 42`

We can use this same operator to join two or more strings, making a new string:

`"Glory" + "To" + "Ukraine" ⇒ "GloryToUkraine"`

Similarly, we can join lists together lists, making a new list:

`[2, 4] + [6] + [0, 1] ⇒ [2, 4, 6, 0, 1]`

Exercise

Write a function `swap_ends(L)` that takes a `list[any]` of length at least 2 and returns a new list where the first and last value have been swapped.

`swap_ends([4, 7, 5, 1, 100]) ⇒ [100, 7, 5, 1, 4]`

Hint

Hint: `L[1:-1]` gives a slice that omits the first and last.

Mutation by index

We saw that we can extract items from a list using **indexing**, like:

```
jvj = [2, 4, 6, 0, 1]
jvj[0] ⇒ 2
```

We can also assign values to an item inside a list, using the same syntax:

```
jvj[1] = 100
print(jvj)
[2, 100, 6, 0, 1]
```

Exercise

What do suppose this code prints?

```
p = [2, 3, 4]
q = p
p[0] = 10
print(q)
```

To **mutate** is to change. Above, we are changing the list `jvj`.

Carefully consider a **state diagram.** `p` and `q` are arrows pointing at the same thing!

We are not creating a new list, we are changing this one. We say `p` is an **alias** of `q`.

Mutation and non-mutation

A working solution to `swap_ends` from earlier:

```
def swap_ends(L: list[any]) -> list[any]:
    """Return a list like L but with
    first and last swapped."""
    return [L[-1]] + L[1:-1] + [L[0]]

mylist = [4, 7, 5, 1, 100]
## We call: swap_ends(mylist)
swap_ends(mylist) ⇒ [100, 7, 5, 1, 4]
mylist ⇒ [4, 7, 5, 1, 100]
```

Notice: the function returns a new list, and `mylist` is unchanged.

Compare with this:

```
def swap_ends_mutate(L: list[any]) -> None:
    """Mutate L, swapping first and last."""
    last = L[-1]
    first = L[0]
    L[0] = last
    L[-1] = first
mylist = [4, 7, 5, 1, 100]
## We call: swap_ends_mutate(mylist)
swap_ends_mutate(mylist) ⇒ None
mylist ⇒ [100, 7, 5, 1, 4]
```

Notice: the function returns `None`, and `mylist` is mutated.

! A critically important difference: **creating a new list** vs **mutating an existing list.**

On some data types there are functions called **methods** that operate on the data value itself. To use these, we write the **name of the variable**, a **dot**, then the **name of the method**, with arguments.

The method `list.append` mutates the list we call it on, adding a single value at the end:

```
mylist = [2, 3, 4]
mylist.append(5)
mylist ⇒ [2, 3, 4, 5]
```

Often we can start with an empty list, then build an answer using `list.append` in a loop:

```
def countdown(n: int) -> list[int]:
    """Return a list counting down from n to 0."""
    answer = []
    while n >= 0:
        answer.append(n)
        n = n - 1

    return answer

countdown(5) ⇒ [5, 4, 3, 2, 1, 0]
```

Exercise

Modify the following code so it returns a `list[int]` containing the values, instead of printing them.

```
def collatz(n: int) -> None:
    """Print the Collatz sequence from n to 1."""
    while n != 1:
        print(n)
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    print(n)
check.expect("C1", collatz(1), [1])
check.expect("C3", collatz(3), [3, 10, 5, 16, 8, 4, 2, 1])
```

Hint

Start with an empty list, and `append` something to it each time.

Exercise

Replace the . . . with only one line of code to make this function work:

```
def double_each(L: list[int]) -> list[int]:
    """Return a new list containing the double of each item from L."""
    answer = []
    for item in L:
        ...
    return answer

check.expect("D0", double_each([2,4,6,0,1]), [4,8,12,0,2])
```

Procedure

To make a new list containing values transformed in some way, start with a new empty list that will be the answer. Loop through the list, transform each value, and append it to the answer.

Method to mutate a list to remove an item: `list.pop`

We use the `list.pop` method to remove a single item from a list.
This function returns the value that was removed.

1. To remove the **last item** call it without an argument:
   It returns the removed value.
   ```
   jvj = [2, 4, 6, 0, 1]
   jvj.pop() ⇒ 1  # Remove last item; list now contains [2, 4, 6, 0]
   jvj.pop() ⇒ 0  # Remove last item; list now contains [2, 4, 6]
   jvj.pop() ⇒ 6  # Remove last item; list now contains [2, 4]
   ```
2. To remove an item at a particular index, call it with the index as the argument:
   ```
   jvj = [2, 4, 6, 0, 1]
   jvj.pop(4) ⇒ 1  # Remove item number 4; list now contains [2, 4, 6, 0]
   jvj.pop(2) ⇒ 6  # Remove item number 2; list now contains [2, 4, 0]
   jvj.pop(0) ⇒ 2  # Remove item number 0; list now contains [4, 0]
   ```

Method to mutate a list to remove an item: `pop`

Exercise

```
mike = [[["That"], ["Dinkum"], ["Thinkum"]],
        [["High", "Operational"], ["Logical"]],
        [["Multi", "Evaluating"], ["Supervisor"]],
        [["Mark", "IV", "Mod", "L"], ["Holmes", "Four"]]]
```

What does `mike.pop().pop().pop()` evaluate to? How is `mike` mutated?

A tuple is an immutable list

Sometimes we want to have a list-like thing that never changes, containing certain types of values.

For each person we might want to store:

- their name, as a `str`
- their year of birth, as an `int`
- their magical possessions, as a `list[str]`.

We **always** want to store exactly three things. So this is a good place to use a `tuple`.

We could use a list. But the point of a list is that we can change it—it's **mutable**. Here it's not.

```
harry = ("Potter, Harry", 1980, ["Elder Wand", "Resurrection Stone", "Invis. Cloak"])
hermione = ("Granger, Hermione", 1979, ["Time Turner"])
frodo = ("Baggins, Frodo", 2968, ["One Ring", "Sting"])
sam   = ("Gamgee, Samwise", 2980, [])
```

Tuples are immutable lists

There are a few ways to create a `tuple`:

- Write values separated by commas, inside round brackets like `(1,2,3)`.
  To create a `tuple` containing only one item, write a seemingly-useless comma after: `(3,)`
- Use the `tuple` function to convert an existing iterable:
  `tuple([2,4,6,0,1])` ⇒ `(2, 4, 6, 0, 1)`
  `tuple('foobar')` ⇒ `('f', 'o', 'o', 'b', 'a', 'r')`
- Use arithmetic, as with lists:
  `(1,2,3) + (4,5)` ⇒ `(1, 2, 3, 4, 5)`

Working with a `tuple` is like working with a list, except we cannot mutate.

Pretty much all we can do is:

- extract items by indexing/slicing:
  ```
  sam[0] ⇒ "Gamgee, Samwise"
  frodo[1:] ⇒ (2968, ['One Ring', 'Sting'])
  ```
- iterate using a `for` loop:
  ```
  for thing in hermione:
      print(thing)

  ## We see:
  Granger, Hermione
  1979
  ['Time Turner']
  ```

We use a `tuple` mostly to store a fixed group of data. Our algorithms will mostly use lists.

Generally we work with a `tuple` of some (short) fixed length.

The type of a tuple is `tuple[...]`, replacing the ... with the types of the values in the `tuple`.

Consider:

```
harry = ("Potter, Harry", 1980, ["Elder Wand", "Resurrection Stone", "Invis. Cloak"])
hermione = ("Granger, Hermione", 1979, ["Time Turner"])
frodo = ("Baggins, Frodo", 2968, ["One Ring", "Sting"])
sam   = ("Gamgee, Samwise", 2980, [])
```

Each of these contains exactly 3 values: a `str`, an `int`, and a `list[str]`.
So each of these is a `tuple[str, int, list[str]]`.

!

Notice the difference between the type of a `list` vs a `tuple`.

- `list` has only **one** argument, indicating the type of **every value** it contains.
- `tuple` has **many** arguments, one argument for each value it contains.

We can already write code to count, using a **while** loop, like so:

```
count = 0
while count < 10:
    print(count)
    count = count + 1
```

This is OK, but we want to count often. There should be an easier way, and there is: **range**.

```
>>> help(range)
Help on class range in module builtins:

class range(object)
 |  range(stop) -> range object
 |  range(start, stop[, step]) -> range object
 |
 |  Return an object that produces a sequence of integers from start (inclusive)
 |  to stop (exclusive) by step.  range(i, j) produces i, i+1, i+2, ..., j-1.
 |  start defaults to 0, and stop is omitted!  range(4) produces 0, 1, 2, 3.
 |  These are exactly the valid indices for a list of 4 elements.
 |  When step is given, it specifies the increment (or decrement).
```

Counting with `range` objects

Directly, a `range` value doesn't do anything:

```
vals = range(4)
print(vals)
## We see:
range(0,4)
```

It isn't a list. But we can convert it to a list:

`list(vals)` ⇒ `[0, 1, 2, 3]`

I can **imagine** "every second number from 1000 to 2000," without writing them all down.

That's what `range` is for: `range(1000, 2000, 2)` represents `[1000, 1002, 1004, 1006, ..., 1998]`, compactly.

Exercise

Fill in the blanks ... to create a `range` object that expands to the desired list:

- `list(range(...))` ⇒ `[5,6,7,8]`
- `list(range(...))` ⇒ `[40, 45, 50, 55, 60, 65, 70]`
- `list(range(...))` ⇒ `[30, 27, 24, 21]`

A `for` loop using `range`

We can convert a `range` object to a list... or iterate directly through it with a `for` loop.

These snippets do the same thing:

```
count = 0
while count < 10:
    print(count)
    count = count + 1
```

```
for count in range(10):
    print(count)
```

Exercise

Replace the ... with only one line of code to make this function work:

```
def mutate_double_each(L: list[int]) -> None:
    """Mutate L so each value is doubled."""
    for i in range(len(L)):
        ...
    return None

thing1 = [2,4,6,0,1]
check.expect("double-r", mutate_double_each(thing1), None)
check.expect("double-m", thing1, [4,8,12,0,2])
```

Can we generalize? What if we wanted to change the items in some other way?

Procedure

To mutate a list `L`, transforming each item while keeping the items in the same order, write `for i in range(len(L)):`. This uses `i` as an **index**. Inside the loop, transform `L[i]` as needed.

A `for` loop inside a `for` loop

Suppose I want to create a table of data, like a times table.

I can represent a single value as a `tuple[int, int, int]`; for example, `(6, 7, 42)` can represent "the product of 6 and 7 is 42".

I want to make a list of such values, something like:

```
[(1,1,1), (1,2,2), (1,3,3),
 (2,1,2), (2,2,4), (2,3,6),
 (3,1,3), (3,2,6), (3,3,9)]
```

I need to create 3 values like `(1, ...)`, then 3 values like `(2, ...)`, then 3 values like `(3, ...)`.

Solution:

```
def timestable(size: int) -> list[tuple[int, int, int]]:
    answer = []
    for row_n in range(1, size+1):
        for column_n in range(1, size+1):
            answer.append( (row_n, column_n, row_n * column_n) )
    # for the tuple:    ^ . . . . . . . . . . . . . . . . . ^
    return answer
```

A `for` loop inside a `for` loop

```
def timestable(size: int) -> list[tuple[int, int, int]]:
    answer = []
    for row_n in range(1, size+1):
        for column_n in range(1, size+1):
            answer.append( (row_n, column_n, row_n * column_n) )
    # for the tuple:        ^ . . . . . . . . . . . . . . . . ^
    return answer
```

Exercise

Using a similar pattern, write a function `even_pairs(n: int)` that returns a `list[tuple[int, int]]` containing all the pairs of integers (x, y) where $x + y$ is even.

```
check.expect("EP3", even_pairs(3), [(1,1), (1,3), (2, 2), (3,1), (3,3)])
```

(Remember, you can tell if an integer is even using the remainder operator:
`13 % 2` ⇒ `1` but `14 % 2` ⇒ `0`.

Module summary

- Use **indexing** like `thing[3]` to extract a single value from a `str` or `list[...]`, or **slicing** like `thing[3:4:2]` to get a collection of values.
- Describe types with `list[...]` and `tuple[..., ...]`.
- Use `for` loops to walk through a `str`, `list[...]`, `tuple[..., ...]`, or `range` object.
- Write code that mutates lists, and that refrains from mutating lists.
- Use loops inside loops.

Before we begin the next module:

- Read and complete the exercises in module 4 of the online textbook, at https://online.cs.uwaterloo.ca/
- Complete the module 4 Review Quiz, due on Monday.