# Loops and Sequences Review

## Python Lists

- **Introduction**: In Python, the list data type is an ordered sequence of elements that can be composed of strings, numbers or even other lists. Lists are mutable and zero based indexed.

▼ Example Code

```
cities = ['Los Angeles', 'London', 'Tokyo']
```

- **Accessing Elements in a List**: To access an element from the `cities` list, you can reference its index number in the sequence:

▼ Example Code

```
cities = ['Los Angeles', 'London', 'Tokyo']
cities[0] # Los Angeles
```

- **Accessing Elements Using Negative Indexing**: To access the last element of any list, you can use `-1` as the index number:

▼ Example Code

```
cities = ['Los Angeles', 'London', 'Tokyo']
cities[-1] # Tokyo
```

- Negative indexing is used to access elements starting from the end of the list instead of the beginning at index `0`.
- **Creating Lists Using the `list()` constructor**: Lists can also be created using the `list()` constructor. The `list()` constructor used to convert an iterable into a list:

▼ Example Code

```
developer = 'Jessica'

print(list(developer))
# Result: ['J', 'e', 's', 's', 'i', 'c', 'a']
```

- **Finding the Length of a List**: You can use the `len()` function to get the length of a list:

▼ Example Code

```
numbers = [1, 2, 3, 4, 5]
len(numbers) # 5
```

- **List Mutability**: Lists are mutable, meaning you can update any element in the list as long as you pass in a valid index number. To update lists at a particular index, you can assign a new value to that index:

▼ Example Code

```
programming_languages = ['Python', 'Java', 'C++', 'Rust']
programming_languages[0] = 'JavaScript'
print(programming_languages) # ['JavaScript', 'Java', 'C++', 'Rust']
```

- **Index Out of Range Error**: If you pass in an index (either positive or negative) that is out of bounds for the list, then you will receive an `IndexError`:

▼ Example Code

```
"""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
"""
```

- **Removing Elements from a List**: Elements can be removed from a list using the `del` keyword:

▼ Example Code

```
developer = ['Jane Doe', 23, 'Python Developer']
del developer[1]
print(developer) # ['Jane Doe', 'Python Developer']
```

- **Checking if an Element Exists in a List**: The `in` keyword can be used to check if an element exists in a list:

▼ Example Code

```
programming_languages = ['Python', 'Java', 'C++', 'Rust']

'Rust' in programming_languages # True
'JavaScript' in programming_languages # False
```

- **Nesting Lists**: Lists can be nested inside other lists:

▼ Example Code

```
developer = ['Alice', 25, ['Python', 'Rust', 'C++']]
```

- To access the nested list, you will need to access it using index `2` since lists are zero-based indexed.

▼ Example Code

```
developer = ['Alice', 25, ['Python', 'Rust', 'C++']]
developer[2] # ['Python', 'Rust', 'C++']
```

- To further access the second language from that nested list, you will need to access it using index `1`:

▼ Example Code

```
developer = ['Alice', 25, ['Python', 'Rust', 'C++']]
developer[2][1] # Rust
```

- **Unpacking Values from a List:** Unpacking values from a list is a technique used to assign values from a list to new variables. Here is an example to unpack the `developer` list into new variables called `name`, `age` and `job` like this:

▼ Example Code

```
developer = ['Alice', 34, 'Rust Developer']
name, age, job = developer
```

- **Collecting Remaining Items From a List**: To collect any remaining elements from a list, you can use the asterisk (`*`) operator like this:

▼ Example Code

```
developer = ['Alice', 34, 'Rust Developer']
name, *rest = developer
```

and ends at index `3`, you can use the following syntax:

▼ **Example Code**

```
desserts = ['Cake', 'Cookies', 'Ice Cream', 'Pie']
desserts[1:3] # ['Cookies', 'Ice Cream']
```

- **Step Intervals**: It is also possible to specify a step interval which determines how much to increment between the indices. Here is a example if you want to extract a list of just even numbers using slicing:

▼ **Example Code**

```
numbers = [1, 2, 3, 4, 5, 6]
numbers[1::2] # [2, 4, 6]
```

## List Methods

- **append()**: Used to add an item to the end of the list. Here is an example of using the `append()` method to add the number `6` to th `numbers` list:

▼ **Example Code**

```
numbers = [1, 2, 3, 4, 5]
numbers.append(6)
print(numbers) # [1, 2, 3, 4, 5, 6]
```

- **Appending lists**: The `append()` method can also be used to add one list at the end of another:

▼ **Example Code**

```
numbers = [1, 2, 3, 4, 5]
even_numbers = [6, 8, 10]

numbers.append(even_numbers)
print(numbers) # [1, 2, 3, 4, 5, [6, 8, 10]]
```

- **extend()**: Used to add multiple items to the end of a list. Here is an example of adding the numbers `6`, `8`, and `10` to the end of the `numbers` list:

▼ **Example Code**

```
numbers = [1, 2, 3, 4, 5]
even_numbers = [6, 8, 10]

numbers.extend(even_numbers)
print(numbers) # [1, 2, 3, 4, 5, 6, 8, 10]
```

- **insert()**: Used to insert an item at a specific index in the list. Here is an example of using the `insert()` method:

▼ **Example Code**

```
numbers = [1, 2, 3, 4, 5]
numbers.insert(2, 2.5)

print(numbers) # [1, 2, 2.5, 3, 4, 5]
```

- **remove()**: Used to remove an item from the list. The `remove()` method will only remove the first occurrence of an item in the list:

▼ **Example Code**

```
print(numbers) # [1, 2, 3, 4, 5, 5]
```

- **pop()**: Used to remove a specific item from the list and return it:

▼ **Example Code**

```
numbers = [1, 2, 3, 4, 5]
numbers.pop(1) # The number 2 is returned
```

- If you don't specify an element for the `pop` method, then the last element is removed.

▼ **Example Code**

```
numbers = [1, 2, 3, 4, 5]
numbers.pop() # The number 5 is returned
```

- **clear()**: Used to remove all items from the list:

▼ **Example Code**

```
numbers = [1, 2, 3, 4, 5]
numbers.clear()

print(numbers) # []
```

- **sort()**: The `sort()` method is used to sort the elements in place. Here is an example of sorting a random list of `numbers` in place:

▼ **Example Code**

```
numbers = [19, 2, 35, 1, 67, 41]
numbers.sort()

print(numbers) # [1, 2, 19, 35, 41, 67]
```

- **sorted()**: Used to sort the elements in a list and return a new sorted list instead of modifying the original list.
- **reverse()**: Used to reverse the order of the elements in a list:

▼ **Example Code**

```
numbers = [6, 5, 4, 3, 2, 1]
numbers.reverse()

print(numbers) # [1, 2, 3, 4, 5, 6]
```

- **index()**: Used to find the first index where an element can be found in a list:

▼ **Example Code**

```
programming_languages = ['Rust', 'Java', 'Python', 'C++']
programming_languages.index('Java') # 1
```

- If the element cannot be found using the `index()` method, then the result will be a `ValueError`.

## Tuples in Python

- **Definition**: A tuple is a Python data type used to create an ordered sequence of values. Tuples can contain a mixed set of data type

▼ **Example Code**

- Tuples are immutable, meaning the elements in the tuple cannot be changed once created. If you try to update one of the items in the tuple, you will get a `TypeError`:

▼ Example Code

```
programming_languages = ('Python', 'Java', 'C++', 'Rust')
programming_languages[0] = 'JavaScript'

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

- **Accessing Elements from a Tuple**: To access an element from a tuple, use bracket notation and the index number:

▼ Example Code

```
developer = ('Alice', 34, 'Rust Developer')
developer[1] # 34
```

- Negative indexing can be used to access elements starting from the end of the tuple:

▼ Example Code

```
numbers = (1, 2, 3, 4, 5)
numbers[-2] # 4
```

- If you try to pass in an index number that exceeds or equals the length of the tuple, then you will receive an `IndexError`:

▼ Example Code

```
numbers = (1, 2, 3, 4, 5)
numbers[7]

"""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
"""
```

- A tuple can also be created using the `tuple()` constructor. Within the constructor, you can pass in different iterables like strings, lists and even other tuples.

▼ Example Code

```
developer = 'Jessica'

print(tuple(developer))
# Result: ('J', 'e', 's', 's', 'i', 'c', 'a')
```

- **Verifying Items in a Tuple**: To check if an item is in a tuple, you can use the `in` keyword like this:

▼ Example Code

```
programming_languages = ('Python', 'Java', 'C++', 'Rust')
```

- **Unpacking Tuples**: Items can be unpacked from a tuple like this:

▼ Example Code

```
developer = ('Alice', 34, 'Rust Developer')
name, age, job = developer
```

- If you need to collect any remaining elements from a tuple, you can use the asterisk (`*`) operator like this:

▼ Example Code

```
developer = ('Alice', 34, 'Rust Developer')
name, *rest = developer
```

- **Slicing Tuples**: Slicing can be used to extract a portion of a tuple. For example, the items `pie` and `cookies` can be sliced into a separate tuple:

▼ Example Code

```
desserts = ('cake', 'pie', 'cookies', 'ice cream')
desserts[1:3] # ('pie', 'cookies')
```

- **Removing Items from Tuples**: Removing an item from a tuple will raise a `TypeError` as tuples are immutable:

▼ Example Code

```
developer = ('Jane Doe', 23, 'Python Developer')
del developer[1]

"""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: "tuple" object doesn't support item deletion
"""
```

- **When to use a Tuple vs a List?**: If you need a dynamic collection of elements where you can add, remove and update elements, ther you should use a list. If you know that you are working with a fixed and immutable collection of data, then you should use a tuple.

## Common Tuple Methods

- **`count()`**: Used to determine how many times an item appears in a tuple. For example, you can check how many times the languag `'Rust'` appears in the tuple:

▼ Example Code

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust')
programming_languages.count('Rust') # 2
```

- If the specified item in the `count()` function is not present at all in the tuple, then the return value will be `0`:

▼ Example Code

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust')
programming_languages.count('JavaScript') # 0
```

- If no arguments are passed to the `count()` function, then Python will return a `TypeError`.
- **index()**: Used to find the index where a particular item is present in the tuple. Here is an example of using the `index()` method to find the index for the language `'Java'`:

▼ Example Code

- If the specified item cannot be found, then Python will return a `ValueError`.
- You can pass an optional start index to the `index()` method to specify where to start searching for the item in the tuple:

▼ **Example Code**

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust', 'Python')
programming_languages.index('Python', 3) # 5
```

- You can also pass in an optional end index to the `index()` method to specify where to stop searching for the item in the tuple:

▼ **Example Code**

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust', 'Python', 'JavaScript', 'Pytho
programming_languages.index('Python', 2, 5) # 2
```

- **`sorted()`**: Used to sort the elements in any iterable and return a new sorted list. Here is an example of creating a new list of numbers using the `sorted()` function:

▼ **Example Code**

```
numbers = (13, 2, 78, 3, 45, 67, 18, 7)
sorted(numbers) # [2, 3, 7, 13, 18, 45, 67, 78]
```

- **Modifying Sorting Behavior**: You can customize the sorting behavior for an iterable using the optional `reverse` and `key` arguments. Here is an example of using the `key` argument to sort items in a tuple by length:

▼ **Example Code**

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust', 'Python')
sorted(programming_languages, key=len)

# Result
# ['C++', 'Rust', 'Java', 'Rust', 'Python', 'Python']
```

- You can create a new list of values in reverse order, using the `reverse` argument like this:

▼ **Example Code**

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust', 'Python')

print(sorted(programming_languages, reverse=True))

# Result
# ['Rust', 'Rust', 'Python', 'Python', 'Java', 'C++']
```

## Loops in Python

- **Definition**: Loops are used to repeat a block of code for a set number of times.
- **`for` loop**: Used to iterate over a sequence (like a list, tuple or string) and execute a block of code for each item in that sequence. Here is an example of using a `for` loop to iterate through a list and print each language to the console:

▼ **Example Code**

```
programming_languages = ['Rust', 'Java', 'Python', 'C++']

for language in programming_languages:
    print(language)
```

```
Rust
Java
Python
C++
"""
```

- Here is an example of using a `for` loop to loop through the string `code` and print out each character:

▼ Example Code

```python
for char in 'code':
    print(char)

"""
Result

c
o
d
e
"""
```

- `for` loops can be nested. Here is an example of using a nested `for` loop:

▼ Example Code

```python
categories = ['Fruit', 'Vegetable']
foods = ['Apple', 'Carrot', 'Banana']

for category in categories:
    for food in foods:
        print(category, food)

"""
Result

Fruit Apple
Fruit Carrot
Fruit Banana
Vegetable Apple
Vegetable Carrot
Vegetable Banana
"""
```

- **`while` loop**: Repeats a block of code until the condition is `False`. Here is an example of using a `while` loop for a guessing game:

▼ Example Code

```python
secret_number = 3
guess = 0

while guess != secret_number:
    guess = int(input('Guess the number (1-5): '))
    if guess != secret_number:
        print('Wrong! Try again.')
```

```
Result

Guess the number (1-5): 2
Wrong! Try again.
Guess the number (1-5): 1
Wrong! Try again.
Guess the number (1-5): 3
You got it!
"""
```

- **`break` and `continue` statements**: Used in loops to modify the execution of a loop.
- The `break` statement is used to exit the loop immediately when a certain condition is met. Here is an example of using the `break` statement for a list of `developer_names`:

▼ Example Code

```python
developer_names = ['Jess', 'Naomi', 'Tom']

for developer in developer_names:
    if developer == 'Naomi':
        break
    print(developer)
```

- The `continue` statement is used to skip that current iteration and move onto the next iteration of the loop. Here is an example to use the `continue` statement instead of a `break` statement:

▼ Example Code

```python
developer_names = ['Jess', 'Naomi', 'Tom']

for developer in developer_names:
    if developer == 'Naomi':
        continue
    print(developer)
```

- Both `for` and `while` loops can be combined with an `else` clause, which is executed only when the loop was not terminated by a `break`:

▼ Example Code

```python
words = ['sky', 'apple', 'rhythm', 'fly', 'orange']

for word in words:
    for letter in word:
        if letter.lower() in 'aeiou':
            print(f"'{word}' contains the vowel '{letter}'")
            break
    else:
        print(f"'{word}' has no vowels")
```

## Ranges and Their Use in Loops

- **The `range()` function**: Used to generate a sequence of integers.

▼ Example Code

- The required `stop` argument is an integer(non-inclusive) that represents the end point for the sequence of numbers being generated. Here is an example of using the `range()` function:

▼ Example Code

```python
for num in range(3):
    print(num)
```

- If a `start` argument is not specified, then the default will be `0`. By default the sequence of integers will increment by `1`. You can use the optional `step` argument to change the default increment value. Here is an example of generating a sequence of even integers from 2 up to but not including 11 (i.e., includes 10)

▼ Example Code

```python
for num in range(2, 11, 2):
    print(num)
```

- If you don't provide any arguments to the `range()` function, then you will get a `TypeError`.
- The `range()` function only accepts integers for arguments and not floats. Using floats will also result in a `TypeError`:

▼ Example Code

```
ERROR!
Traceback (most recent call last):
  File "<main.py>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```

- You can use a negative integer for the `step` argument to generate a sequence of integers in decrementing order:

▼ Example Code

```python
for num in range(40, 0, -10):
    print(num)
```

- The `range()` function can also be used to create a list of integers by using it with the `list` constructor. The `list` constructor is used to convert an iterable into a list. Here is an example of generating a list of even integers between 2 and 10 inclusive:

▼ Example Code

```python
numbers = list(range(2, 11, 2))
print(numbers) # [2, 4, 6, 8, 10]
```

## `enumerate()` and `zip()` functions in Python

- **`enumerate()`**: used to iterate over a sequence and keep track of the index for each item in that sequence. The `enumerate()` function takes an iterable as an argument and returns an `enumerate` object that consist of the index and value of each item in the iterable.

▼ Example Code

```python
languages = ['Spanish', 'English', 'Russian', 'Chinese']

for index, language in enumerate(languages):
    print(f'Index {index} and language {language}')

# Result
# Index 0 and language Spanish
# Index 1 and language English
# Index 2 and language Russian
```

- The `enumerate()` function can also be used outside of a `for` loop:

▼ **Example Code**

```
languages = ['Spanish', 'English', 'Russian', 'Chinese']

print(list(enumerate(languages)))
# [(0, 'Spanish'), (1, 'English'), (2, 'Russian'), (3, 'Chinese')]
```

- The `enumerate()` function also accepts an optional `start` argument that specifies the starting value for the count. If this argument is omitted, then the count will begin at `0`.
- **`zip()`**: Used to iterate over multiple iterables in parallel. Here's an example using the `zip()` function to iterate over `developers` and `ids`:

▼ **Example Code**

```
developers = ['Naomi', 'Dario', 'Jessica', 'Tom']
ids = [1, 2, 3, 4]

for name, id in zip(developers, ids):
    print(f'Name: {name}')
    print(f'ID: {id}')


"""
Result

Name: Naomi
ID: 1
Name: Dario
ID: 2
Name: Jessica
ID: 3
Name: Tom
ID: 4
"""
```

## List comprehensions in Python

- **Definition**: List comprehension allows you to create a new list in a single line by combining the loop and the condition directly within square brackets. This makes the code shorter and often easier to read.

▼ **Example Code**

```
even_numbers = [num for num in range(21) if num % 2 == 0]
print(even_numbers)
```

## Iterable methods

- **`filter()`**: Used to filter elements from an iterable based on a condition. It returns an iterator that contains only the elements tha satisfy the condition. Here is an example of creating a new list of just words longer than four characters:

▼ **Example Code**

```
words = ['tree', 'sky', 'mountain', 'river', 'cloud', 'sun']

def is_long_word(word):
```

```python
print(long_words) # ['mountain', 'river', 'cloud']
```

- `map()`: Used to apply a function to each item in an iterable and return a new iterable with the results. Here is an example of using the `map()` function to convert a list of celsius temperatures to fahrenheit:

▼ Example Code

```python
celsius = [0, 10, 20, 30, 40]

def to_fahrenheit(temp):
    return (temp * 9/5) + 32

fahrenheit = list(map(to_fahrenheit, celsius))
print(fahrenheit) # [32.0, 50.0, 68.0, 86.0, 104.0]
```

- `sum()`: Used to get the sum from an iterable like a list or tuple. Here is an example of using the `sum()` function:

▼ Example Code

```python
numbers = [5, 10, 15, 20]
total = sum(numbers)
print(total) # Result: 50
```

- You can also pass in an optional `start` argument which sets the initial value for the summation. Here is an updated example using the `start` argument as a positional argument:

▼ Example Code

```python
numbers = [5, 10, 15, 20]
total = sum(numbers, 10) # positional argument
print(total) # 60
```

- You can also choose to use the `start` argument as a keyword argument like this instead:

▼ Example Code

```python
numbers = [5, 10, 15, 20]
total = sum(numbers, start=10) # keyword argument
print(total) # 60
```

## Lambda functions

- **Definition**: A lambda function in Python is a concise way to create a function without a name (an anonymous function).
- Lambda functions are often used as an argument to another function. Here is an example of a lambda function:

▼ Example Code

```python
numbers = [1, 2, 3, 4, 5]

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # [2, 4]
```

- Best practices for using lambda functions include not assigning them to a variable, keeping them simple and readable, and using them for short, one-off functions.

**Assignment**

Please complete the assignment

Submit

Ask for Help