# Python Programming: Sets and Tuples

## Learning Objectives

*After this lesson, you will be able to:*

- Perform common actions with sets.
- Perform common actions with tuples.
- Know when to use different data structures.

## Discussion: Lists

Here are some lists:

```
unique_colors = ["red", "yellow", "red", "green", "red", "yellow"]
subscribed_emails = ["mary@gmail.com", "opal@gmail.com", "mary@gmail.com", "
```

What could be a problem here?

## Introducing Sets

Lists:

```
unique_colors_list = ["red", "yellow", "red", "green", "red", "yellow"]
subscribed_emails_list = ["mary@gmail.com", "opal@gmail.com", "mary@gmail.cc
```

Sets: Lists without duplicates!

```
unique_colors_set = {"green", "yellow", "red"}
subscribed_emails_set = {"mary@gmail.com", "opal@gmail.com", "sayed@gmail.cc
```

- Notice the `{}` versus the `[]`.

## How Can We Make a Set?

If we make a set via a list, Python removes duplicates automatically.

```
my_set = set(a_list_to_convert)


# In action:
unique_colors_list = ["red", "yellow", "red", "green", "red", "yellow"]
unique_colors_set = set(unique_colors_list)
# => {"green", "yellow", "red"}


# In action:
unique_colors_set_2 = set(["red", "yellow", "red", "green", "red", "yellow"]
# => {"green", "yellow", "red"}
```

We can make a set directly using curly braces:

```
colors = {"red", "orange", "yellow", "green", "blue", "indigo", "violet"}
```

## Important Note: Sets

Lists are always in the same order:

- `my_list = ["green", "yellow", "red"]` is always going to be `["green", "yellow", "red"]`
- `my_list[0]` is always `"green"`; `my_list[1]` is always `"yellow"`; `my_list[2]` is always `"red"`.

Sets are not like this! Like dictionaries, they're not ordered.

- `my_set = {"green", "yellow", "red"}` could later be `{"red", "yellow", "green"}`!
- `my_set[0]` could be `"green"`, `"red"`, or `"yellow"` - we don't know!

We **cannot** do: `print(my_set[0])` - it could be anything! Python won't let us.

## We Do: Creating a Set from a List

Let’s pull up a new `set_practice.py` file and make some sets!

- Make a list `clothing_list` containing the main color of your classmates’ clothing.
- Using `clothing_list`, make a set named `clothing_set`.
- Use a `for` loop to print out both `clothing_list` and `clothing_set`.
- Try to print an index!

## We Do: Adding to a Set

How do we add more to a set?

```
# In a list:
clothing_list.append("red")


# In a set
clothing_set.add("red")
```

`add` vs `append` - this is because we can't guarantee it's going at the end!

Let's a few colors to `clothing_list` and `clothing_set` and then print them.

- What happens if you add a duplicate?

## We Do: Removing from a List and a Set

Remember, lists are always the same order and sets are not!

- With the set `{"green", "yellow", "red"}`, `my_set[0]` could be `green`, `red`, or `yellow`.

So thus, we need to be careful about removal:

```
# In a list:
clothing_list.pop() # Removes and returns the last item in the list.
clothing_list.pop(0) # Removes and returns a specific (here, the first) item


# In a set
clothing_set.pop() # No! This is unreliable! The order is arbitrary.
clothing_set.pop(0) # No! Python throws an error! You can't index sets.
clothing_set.remove('red') # Do this! Call the element directly!
```

## We Do: Set Intersection

- One thing that sets are *really* good at is *relational algebra*. This is a fancy word for an SQL join, or comparing elements between two sets.
- Sets are really good at this because they use the same hashing trick under the hood that dictionaries use and makes them so fast.

Let’s start by making two sets:

```
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}
```

## We Do: Set Intersection

Now let’s use the `.intersection()` set method to find out *what elements these two sets have in common*:

```
set1.intersection(set2)
```

Yields the result:

```
{4, 5}
```

- This makes sense, the two sets share the elements `4` and `5`.
- Note that this is commutative, meaning we could also write `set2.intersection(set1)` and receive the same result.

## We Do: Set Differences

- Instead of finding *elements in common* between two sets, we can also find *their differences*.
- To do this, we use `.difference()`.
- Note that this method *is not commutative*, meaning *order matters*.

```
set1.difference(set2)
```

Yields the result:

```
{1, 2, 3}
```

```
set2.difference(set1)
```

Yields the result:

```
{6, 7, 8}
```

## Quick Review: Sets vs. Lists

**Lists**:

- The original, normal object.
- Created with `[]`.
- `append()`, `insert(index)`, `pop()`, `pop(index)`.
- Duplicates and mutable.

**Sets**:

- Lists without duplicates.
- Created with `{}` or with `set(my_list)`.
- `add()` and `remove(element)`.

## Quick Review: Sets vs. Lists

```
### Creation ###

# List

my_list = ["red",  "yellow", "green", "red"]

# Sets

my_set = {"red",  "yellow", "green"}

my_set2 = set(my_list)

my_set = set(a_list_to_convert)


### Appending a New Value ###

my_list.append("blue")

my_set.add("blue")
```

## Discussion: Immutability Thoughts

A set is a type of list which doesn't allow duplicates.

What if, instead, we have a list we don't want to change?

```
rainbow_colors = ["red", "orange", "yellow", "green", "blue", "indigo", "vic
```

We **don't** want:

```
rainbow_colors[0] = "gray"
## Gray's not in the rainbow!
rainbow_colors.pop()
## We can't lose violet!
rainbow_colors.append("pink")
# Pink's not in the rainbow!
```

We want `rainbow_colors` to be **immutable** - the list *cannot* be changed.

How we do that in Python?

## Introducing: Tuples

Sets are one specific type of list.

- No duplicates, but mutable.

**Tuples** are another specific type of list.

- Duplicates, but immutable.
- A list that *cannot* be changed.

```
rainbow_colors_tuple = ("red", "orange", "yellow", "green", "blue", "indigo"
```

When should you use a tuple?

- When you need data protection through immutability.
- When you never want to change the list.
- Tuples are sometimes wrapped in a class namespace to simulate what would be done with a const structure in a C-based language.
- This is a way of holding appconstants, or constants your program will use (like the API endpoint of a server, credentials, etc.)

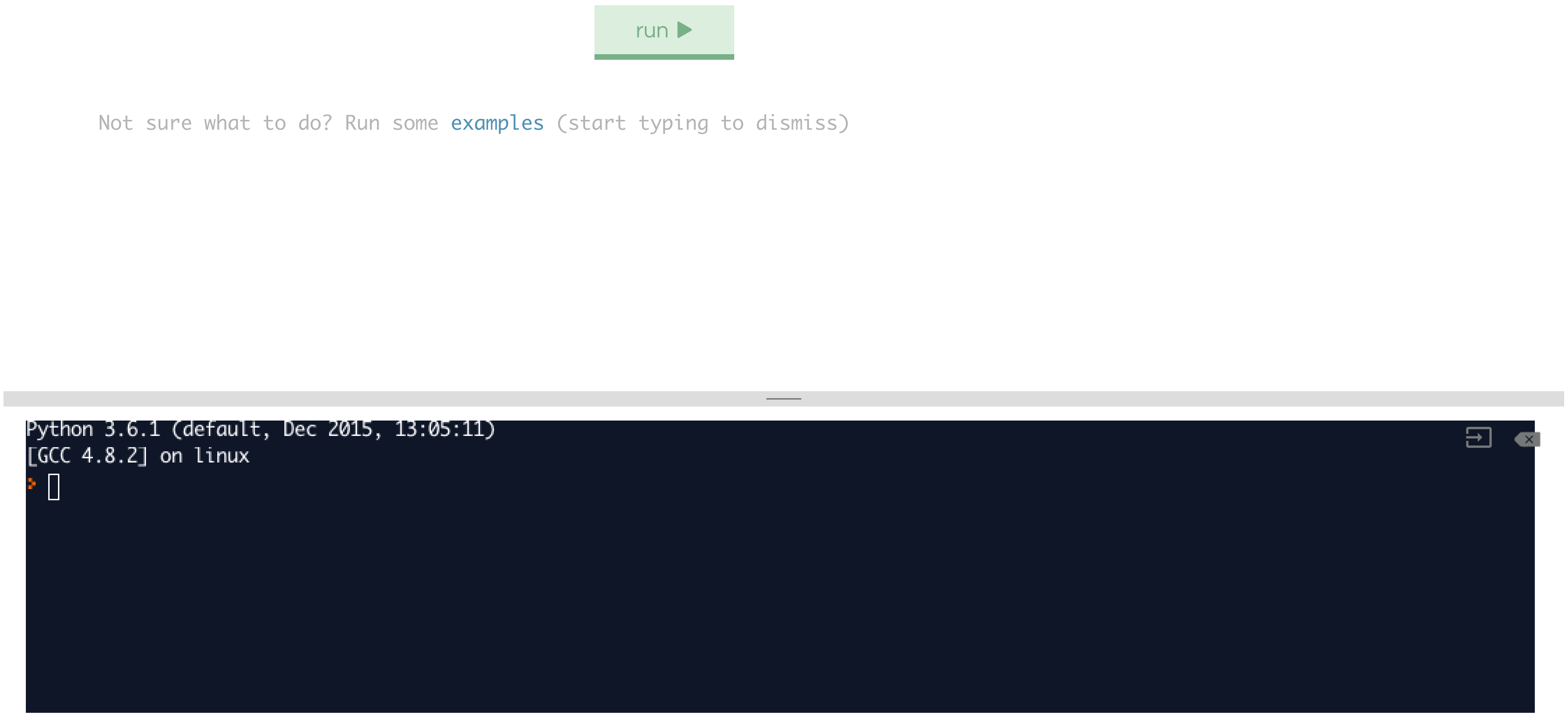
## Tuple Syntax

- Created with parentheses `()`.
- Access values via indices (like regular lists, but *not* like sets).

```
rainbow_colors_tuple = ("red", "orange", "yellow", "green", "blue", "indigo"
print(rainbow_colors[1])
# Prints "orange"
```

- Tuples can be printed with a `for` loop (just like a set or list!).

```
rainbow_colors_tuple = ("red", "orange", "yellow", "green", "blue", "indigo"


for color in rainbow_colors_tuple:
  print(color)
```



Not sure what to do? Run some examples (start typing to dismiss)

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
> 
```

## Quick Review: Sets, Tuples, Lists

**List**:

- The original, normal object: `["red", "red", "yellow", "green"]`.
- Has duplicates; mutable: `append()`, `insert(index)`, `pop()`, `pop(index)`

**Set**:

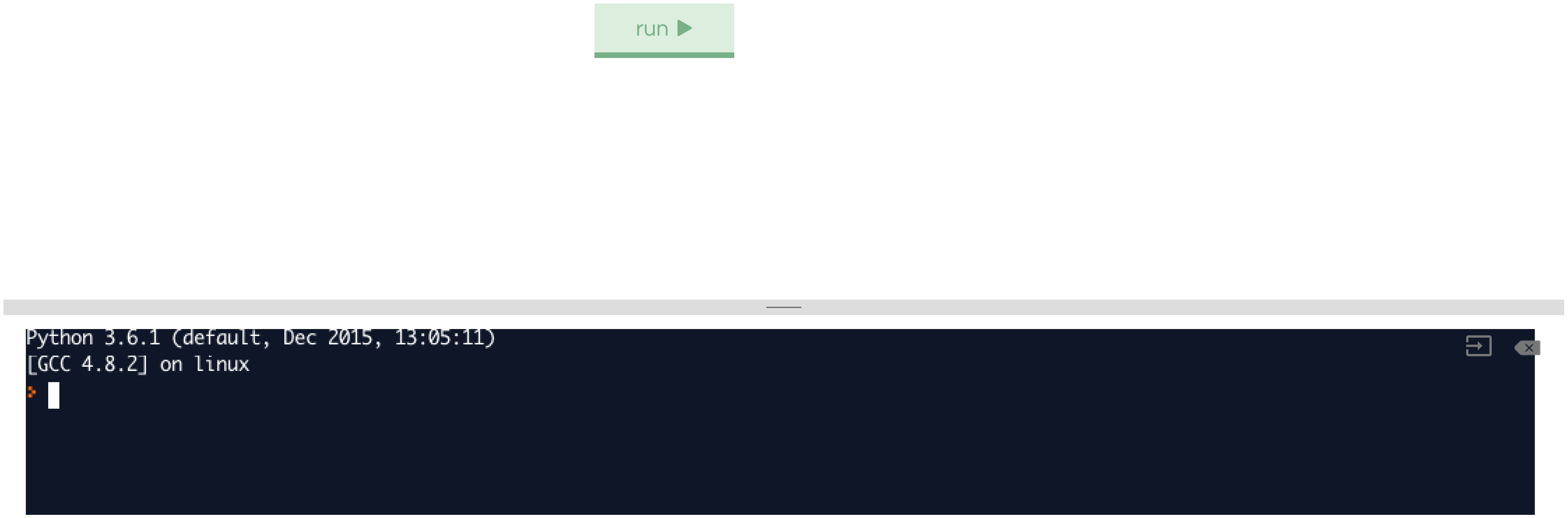
- List without duplicates: `{"red", "yellow", "green"}`.
- Mutable: `add()` and `remove(element)`

**Tuple**:

- Has duplicates, but immutable: You can't change it!
- `("red", "red", "yellow", "green")` will *always* be `("red", "red", "yellow", "green")`.

## Quick Review: Sets, Tuples, Lists

```
### Creation ###

# List

my_list = ["red",  "yellow", "green", "red"]

# Sets

my_set = {"red",  "yellow", "green"}

my_set2 = set(my_list))

my_set = set(a_list_to_convert)

# Tuples

my_tuple = ("red",  "yellow", "green")


### Appending a New Value ###
```



```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
>
```

## You Do: List Types Practice

Create a local file, `sets_tuples.py`. In it:

- Create a list (`[]`), set (`{}`), and tuple (`()`) of some of your favorite foods.
- Create a second set from the list.

Next, in every list type that you can:

- Add `"pizza"` anywhere; append `"eggs"` to the end.
- Remove `"pizza"`.
- Re-assign the element at index `1` to be `"popcorn"`.
- Remove the element at index `2` and re-insert it at index `0`.
- Print the element at index `0`.

Print your final lists using a loop, then print their types. Don't throw an error!

## Summary and Q&A

We've learned two new types of lists:

Sets:

- A mutable list without duplicates.
- Handy for storing emails, usernames, and other unique elements.

```
email_set = {'my_email@gmail.com', 'second_email@yahoo.com', "third_email@hc
```

Tuples:

- An immutable list that allows duplicates.
- Handy for storing anything that won't change.

```
rainbow_tuple = ("red", "orange", "yellow", "green", "blue", "indigo", "viol
```

## Additional Reading

- Repl.it that recaps Tuples
- Python Count Occurrences of Letters, Words and Numbers in Strings and Lists-Video
- Storing Multiple Values in Lists
- Sets and Frozen Sets
- Sets
- Python Tuple
- Tuples
- Strings, Lists, Tuples, and Dictionaries Video
- Python Data Structures: Lists, Tuples, Sets, and Dictionaries Video