



Python Programming: Loops

Learning Objectives

After this lesson, you will be able to:

- Use a `for` loop to iterate a list.
- Use `range()` to dynamically generate loops.
- Use a `while` loop to control program flow.
- Use `break` to exit a loop.

Discussion: A Small List

This situation isn't so bad:

```
visible_colors = ["red", "orange", "yellow", "green", "blue", "violet"]
print(visible_colors[0])
print(visible_colors[1])
print(visible_colors[2])
print(visible_colors[3])
print(visible_colors[4])
print(visible_colors[5])
```

But what would we do if there were 1,000 items in the list to print?

The `for` Loop

The `for` loop always follows this form:

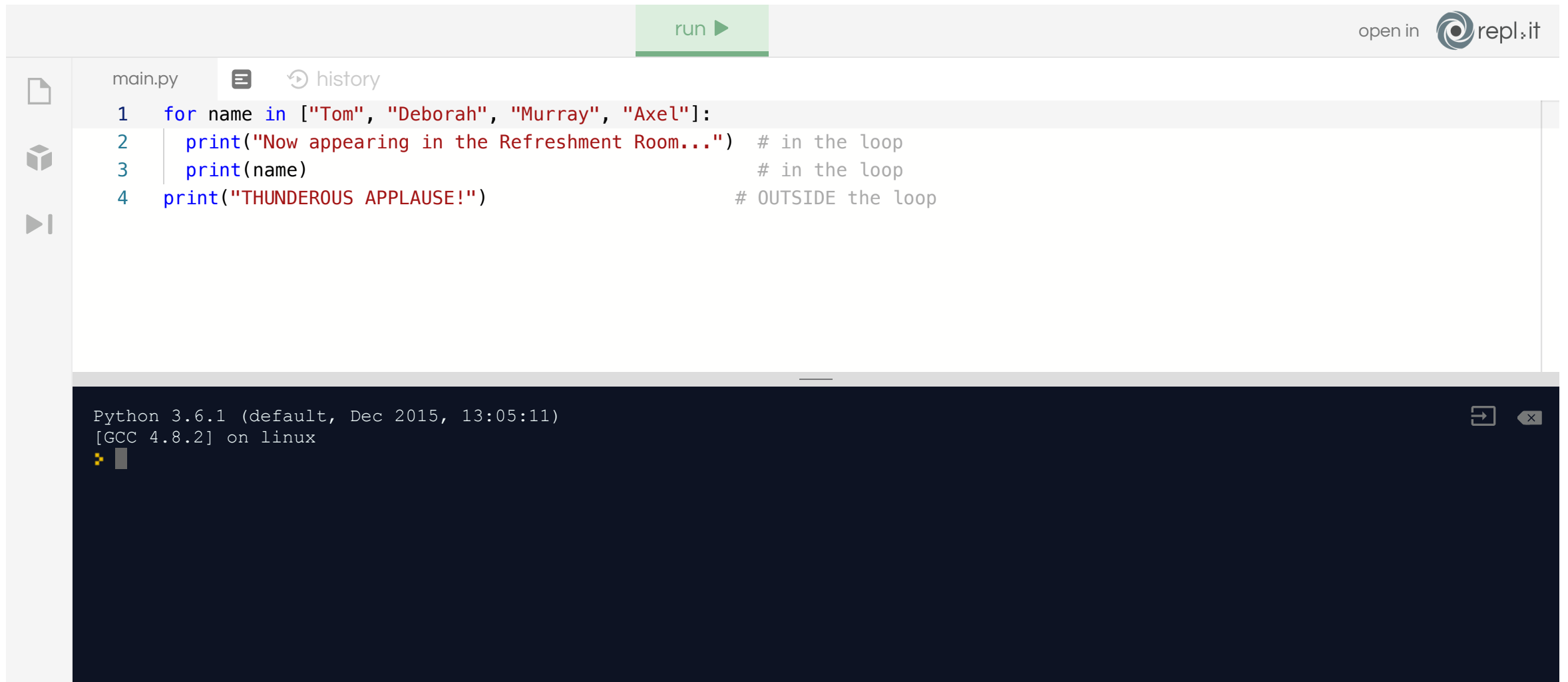
```
for item in collection:  
    # Do something with item
```

For example:

```
visible_colors = ["red", "orange", "yellow", "green", "blue", "violet"]  
  
for each_color in visible_colors:  
    print(each_color)
```

Knowledge Check: What will this code do?

Think about what the code will do before you actually run it.



The image shows a Repl.it Python environment. At the top right, there is a "run" button with a play icon and a link to "open in repl.it". The code editor displays the following Python code in a file named "main.py":

```
1 for name in ["Tom", "Deborah", "Murray", "Axel"]:  
2     print("Now appearing in the Refreshment Room...") # in the loop  
3     print(name) # in the loop  
4     print("THUNDEROUS APPLAUSE!") # OUTSIDE the loop
```

Below the code editor is a terminal window with the following text:

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

We Do: Writing a Loop

Let's write a loop to print names of guests.

First, we need a list.

- Create a local `.py` file named `my_loop.py`.
- Make your list: Declare a variable `my_list` and assign it to a list containing the names of at least five people.

We Do: Write a Loop - Making the Loop

Now, we'll add the loop.

- Skip a line and write the first line of your `for` loop.
 - For the variable that holds each item, give it a name that reflects what the item is (e.g. `name` or `person`).
- Inside your loop, add the code to print `"Hello, "` plus the name.

```
"Hello, Felicia!"  
"Hello, Srinivas!"
```

We Do: Write a loop to greet people on your guest list

Our guests are definitely VIPs! Let's give them a lavish two-line greeting.

- Inside your loop, add the code to print another sentence of greeting:

```
"Hello, Srinivas!"
```

```
"Welcome to the party!"
```


Discussion: Where Else Could We Use a Loop?

A loop prints everything in a collection of items.

- `guest_list = ["Fred", "Cho", "Brandi", "Yuna", "Nanda", "Denise"]`

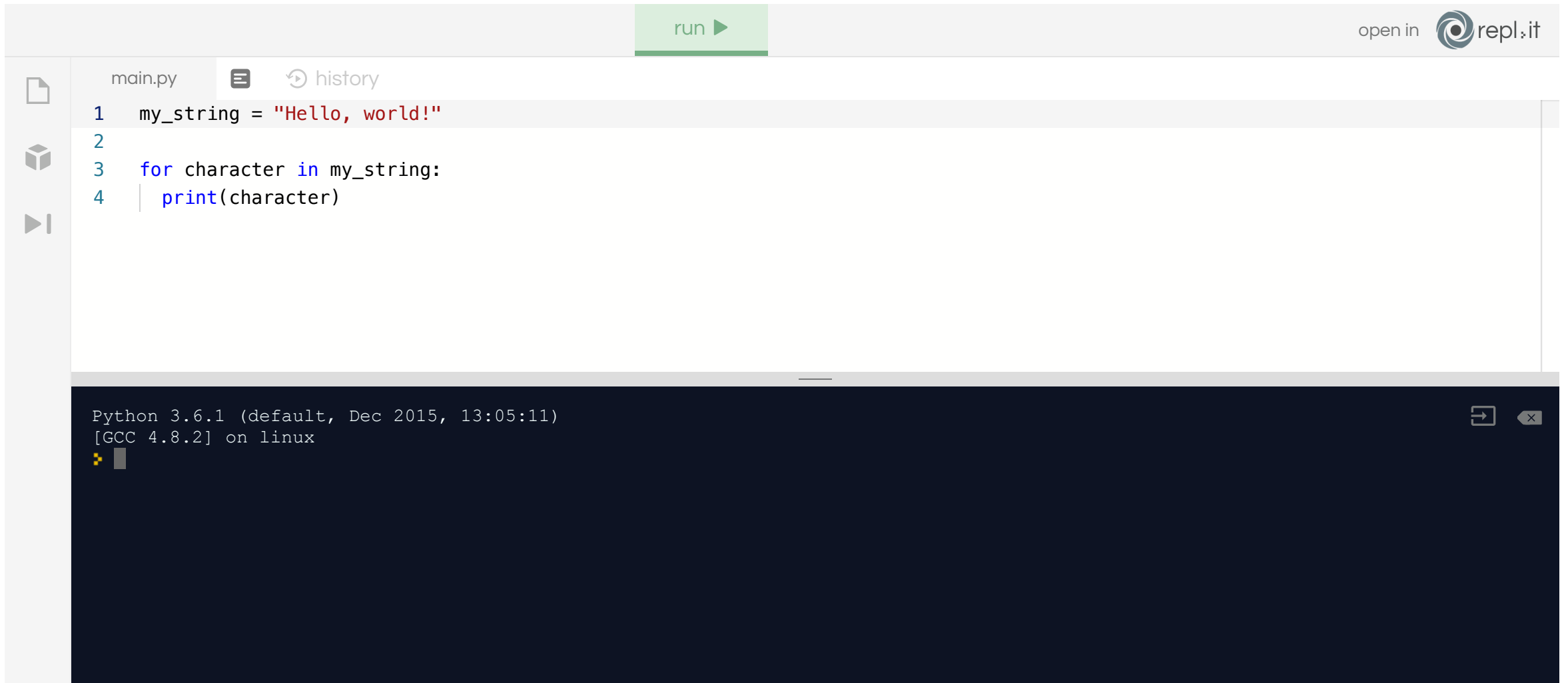
What, besides a list, could we use a loop on?

Hint: There are six on this slide!

Looping Strings

Loops are collections of strings and numbers.

Strings are collections of characters!



The screenshot shows a Repl.it Python environment. At the top, there is a "run" button with a play icon and a link to "open in repl.it". Below this, the code editor shows a file named "main.py" with the following Python code:

```
1 my_string = "Hello, world!"  
2  
3 for character in my_string:  
4     print(character)
```

Below the code editor is a terminal window with the following output:

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

What about...Looping For a Specific Number of Iterations?

We have:

```
guest_list = ["Fred", "Cho", "Brandi", "Yuna", "Nanda", "Denise"]

for guest in guest_list:
    print("Hello, " + guest + "!")
```

The loop runs for every item in the list - the length of the collection. Here, it runs 6 times.

What if we don't know how long `guest_list` will be?

Or only want to loop some of it?

Enter: Range

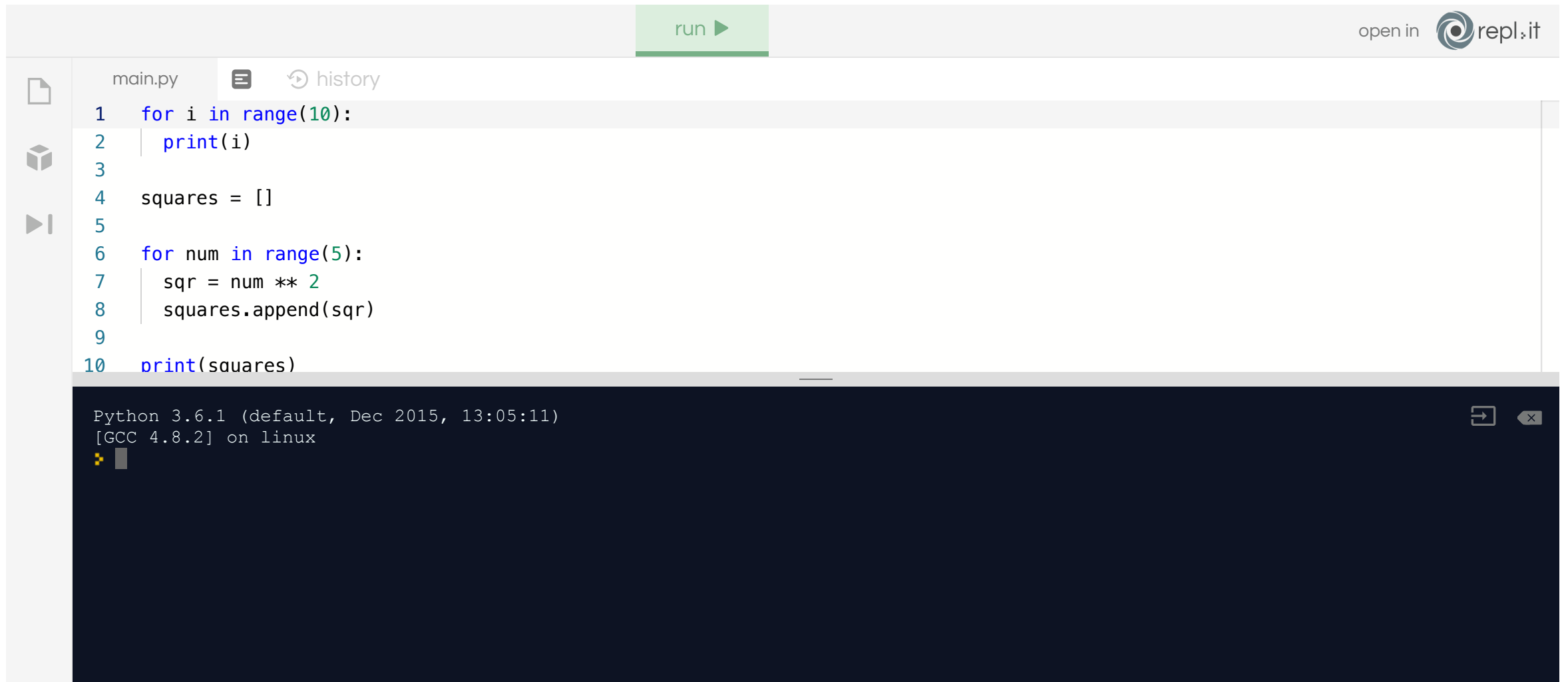
`range(x)` :

- Automatically generated.
- A list that contains only integers.
- Starts at zero.
- Stops before the number you input.

```
range(5) # => [0, 1, 2, 3, 4]
```

Looping Over a Range

Let's look at `range` in action:



The screenshot shows a Repl.it Python environment. At the top right, there is a "run" button and a link to "open in repl.it". The code editor displays the following Python code in a file named `main.py`:

```
1 for i in range(10):
2     print(i)
3
4 squares = []
5
6 for num in range(5):
7     sqr = num ** 2
8     squares.append(sqr)
9
10 print(squares)
```

Below the code editor is a terminal window with the following output:

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

Looping Over a Range

Looping over `names` here is really just going through the loop 4 times - at index `0`, `1`, `2`, and `3`.

We can instead use `range(x)` to track the index and loop `names: range(4)` is `[0, 1, 2, 3]`.

We can then use `len(names)`, which is 4, as our range.



```
run ▶ open in repl.it  
main.py history  
1 names = ["Flint", "John Cho", "Billy Bones", "Nanda Yuna"]  
2  
3 for each_name in range(len(names)):  
4     print(names[each_name])  
  
Python 3.6.1 (default, Dec 2015, 13:05:11)  
[GCC 4.8.2] on linux  
➤
```

Range to Modify Collections

Why would you use `range` on a list, when you could just loop the list?

We can't do:

```
guest_list = ["Fred", "Cho", "Brandi", "Yuna", "Nanda", "Denise"]

for guest in guest_list:
    guest = "A new name"
```

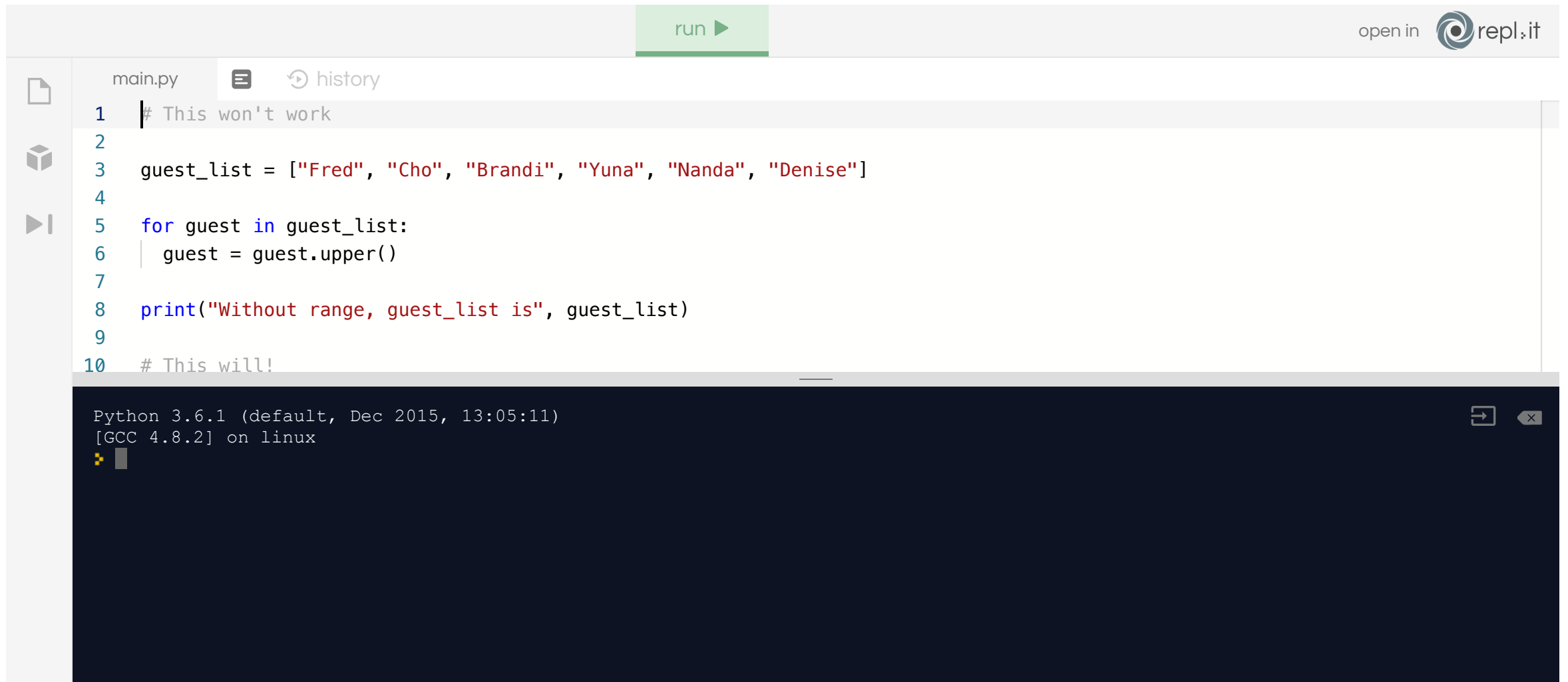
But we can do:

```
guest_list = ["Fred", "Cho", "Brandi", "Yuna", "Nanda", "Denise"]

for guest in range(len(guest_list)):
    guest_list[guest] = "A new name"
```

Looping Over a Range

Let's make the list all uppercase:



The screenshot shows a Repl.it Python environment. At the top right, there is a "run" button and a link to "open in repl.it". The code editor displays the following Python code in a file named "main.py":

```
1 # This won't work
2
3 guest_list = ["Fred", "Cho", "Brandi", "Yuna", "Nanda", "Denise"]
4
5 for guest in guest_list:
6     guest = guest.upper()
7
8 print("Without range, guest_list is", guest_list)
9
10 # This will!
```

Below the code editor is a terminal window with the following output:

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


Knowledge Check: Which of the following lines is correct?

```
my_list = ['mon', 'tue', 'wed', 'thu', 'fri']
```

```
for day in range(my_list):           # answer A
```

```
for day in range(len(my_list)):     # answer B
```

```
for day in range(my_list.length):  # answer C
```

You Do: Range

Locally, create a new file called `range_practice.py`.

In it:

- Create a list of colors.
- Using a `for` loop, print out the list.
- Using `range`, set each item in the list to be the number of characters in the list.
- Print the list.

For example:

```
["red", "green", "blue"]  
# =>  
[3, 5, 4]
```

Quick Review: For Loops and Range

`for` loops:

```
# On a list (a collection of strings)
guest_list = ["Fred", "Cho", "Brandi", "Yuna", "Nanda", "Denise"]
for guest in guest_list:
    print("Hello, " + guest + "!")

# On a string (a collection of characters)
my_string = "Hello, world!"
for character in my_string:
    print(character)

#### Range ####
```

The While Loop

What about “While the bread isn’t brown, keep cooking”?

Python provides two loop types.

`for`:

- You just learned!
- Loops over collections a finite number of times.

`while`:

- You’re about to learn!
- When your loop could run an indeterminate number of times.
- Checks if something is `True` (*the bread isn’t brown yet*) and runs until it’s set to `False` (*now the bread is brown, so stop*).

While Loop Syntax

```
# While <something> is true:  
#     Run some code  
#     If you're done, set the <something> to false  
#     Otherwise, repeat.  
  
a = 0  
while a < 10:  
    print(a)  
    a += 1
```

While Loop: Be Careful!

Don't *ever* do:

```
a = 0
while a < 10:
    print(a)
```

And don't ever do:

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

Your program will run forever!

If your program ever doesn't leave a loop, hit `control-c`.

We Do: Filling a Glass of Water

Create a new local file, `practicing_while.py`.

In it, we'll create:

- A variable for our current glass content.
- Another variable for the total capacity of the glass.

Let's start with this:

```
glass = 0
glass_capacity = 12
```

Can you start the `while` loop?

We Do: Filling a Glass of Water

Add the loop:

```
glass = 0
glass_capacity = 12

while glass < glass_capacity:
    glass += 1 # Here is where we add more water
```

That's it!

Side Note: Input()

Let's do something more fun.

With a partner, you will write a program that:

- Has a user guess a number.
- Runs until the user guesses.

But first, how do we have users input numbers?

Using `input()`.

```
user_name = input("Please enter your name:")  
# user_name now has what the user typed  
print(user_name)
```

Erase the code in your `practicing_while.py` file and put the above. Run it! What happens? Does it work?

You Do: A Guessing Game

Now, get with a partner! Let's write the the game.

Decide who will be driver and who will be navigator. Add this to your existing file.

- Set a variable, `answer` to `"5"` (yes, a string!).
- Prompt the user for a guess and save it in a new variable, `guess`.
- Create a `while` loop, ending when `guess` is equal to `answer`.
- In the `while` loop, prompt the user for a new `guess`.
- After the `while` loop, print "You did it!"

Discuss with your partner: Why do we need to make an initial variable before the loop?

You Do: A Guessing Game (Solution)

```
answer = "4"
guess = input("Guess what number I'm thinking of (1-10): ")
while guess != answer:
    guess = input("Nope, try again: ")
print("You got it!")
```

How'd you do? Questions?

Exiting a Loop

There are times when you may want to exit a loop before the final condition has been met. Perhaps the input from another part of the program has satisfied another, separate condition that makes the rest of the loop unnecessary. Enter the `break` statement.

```
while True:

    if my_condition == 1:
        # my condition is met! No need to continue on
        break

    else my_condition == 0:
        my_condition = 1

# note that this is within the scope of the while loop
print('This doesn\'t get run if the break is triggered!')
```

Continuing a Loop

There are times when you may want to to `continue` a loop *without running code beneath the `continue` statement*. The `continue` allows you to do just that! After the `continue` statement is triggered, the loop *continues the next iteration of the loop without executing any code beneath it on that iteration*.

```
number = 0

for number in range(5):
    number = number + 1

    if number == 3:
        continue

        print('My number is currently 3')

    print(f'Number is {str(number)}')

print('Out of loop')
```

Prints:

```
Number is 1
Number is 2
Number is 4
```

Passing within a Loop

The `pass` statement is like a placebo in a loop: it allows a loop to execute without any interruption. This example may seem odd, and we'll cover the more common use case in the next example.

```
number = 0

for number in range(5):
    number = number + 1

    if number == 3:
        pass

        print('My number is currently 3')

    print(f'Number is {str(number)}')

print('Out of loop')
```

Prints:

```
Number is 1
Number is 2
My number is currently 3
Number is 3
Number is 4
```

Passing within a Function or Class

The most common use case for `pass` is to act as a placeholder for a function that has yet to be written.

Developers will often do this if they're creating the architecture for a program but haven't gotten to actually building the logic yet.

```
def my_empty_function():  
    pass
```

What happens if we *don't* put the `pass` statement in the code and attempt to execute the function definition?

Throwing Exceptions within a Function or Class

Note that the previous example will allow the function to be called, but the function won't do anything. If the programmer wishes to alert the user, they may also use `raise` to interrupt the program execution. The following is common to see in larger applications that are in the process of being built by a dev team:

```
def my_empty_function():  
    raise NotImplementedError
```

What happens when we call this function? *Hint: look at the type of error that is returned!*

Summary + Q&A

Loops:

- Common, powerful control structures that let us efficiently deal with repetitive tasks.

`for` loops:

- Used to iterate a set number of times over a collection (e.g. list, string, or using `range`).
- `range` use indices, not duplicates, so it lets you modify the collection.

`while` loops:

- Run until a condition is false.
- Used when you don't know how many times you need to iterate.

That was a tough lesson! Any questions?

Additional Reading

- [Learn Python Programming: Loops Video](#)
- [Python: For Loop](#)
- [Python: Loops](#)