Python Code Abstraction

Learning Objectives

After this lesson, you will be able to:

- Use `itertools` to implement efficient looping.
- Use list comprehensions to concisely create lists.

What Is Code Abstraction?

A key part of programming is “Don’t Repeat Yourself:”

- Write once, use many times.
- Don’t repeat yourself!
- Have we mentioned this? It bears repeating! 😁

Programmers aren’t lazy — they’re efficient!

Python is filled with functionality that has already been written for you.

- You didn’t need to write `lists.append()` — you just use it!

Code abstraction takes this to the next level.

- Python has many built-in functions that perform common but complicated tasks.

We’re going to look at just a few of these.

Like What?

Let's look at `itertools`.

- A collection of functions.
- Designed to make looping or iterating easier (iterating tools –> iter-tools)

Using `itertools`, this is what we'll learn to do in the following slides:

```
# We can group list items:
animals = ['dog', 'dog', 'dog', 'horse', 'horse']
# => dog ['dog', 'dog', 'dog'] - The three dogs are grouped together.
# => horse ['horse', 'horse'] - The two horses are grouped together.


# We can chain lists:
food = ['pizza', 'tacos', 'sushi']
colors = ['red', 'green']
# => lists_chained =['pizza', 'tacos', 'sushi', 'red', 'green']


# We can add elements:
```

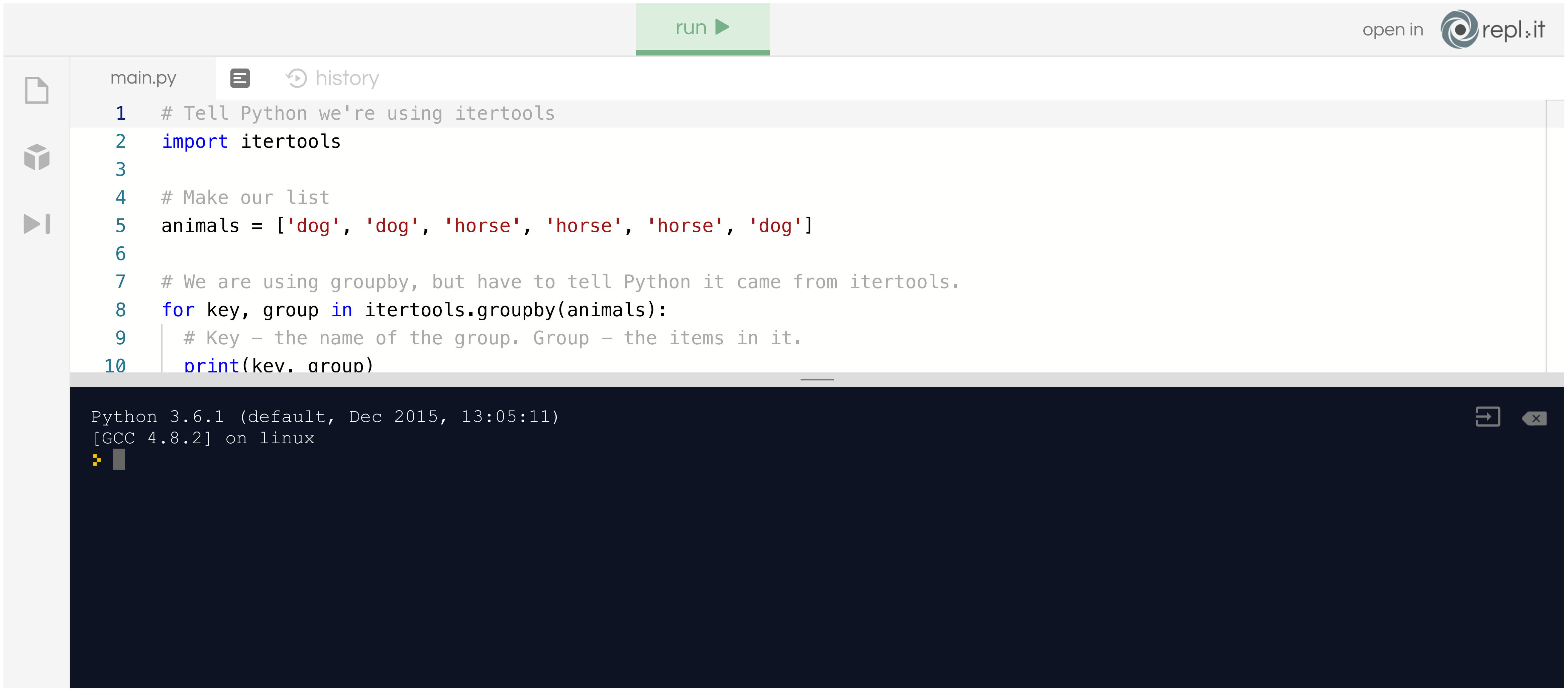
Our First Itertool: `groupby()`

Sometimes, our lists contain repeated items that work better for us if they are all grouped together. Using `groupby()`, which Python has written for us in `itertools`, we can take our list and group the items.

- `key`: The name of the group (in this case `dog` and `horse`).
- `group`: A list containing all occurrences of that key from the original list.

Our First Itertool: `groupby()`

All the gibberish-looking stuff is memory addresses. Python tells us, “I made a new object and I put it here.” We’ll talk about this on the next slide.



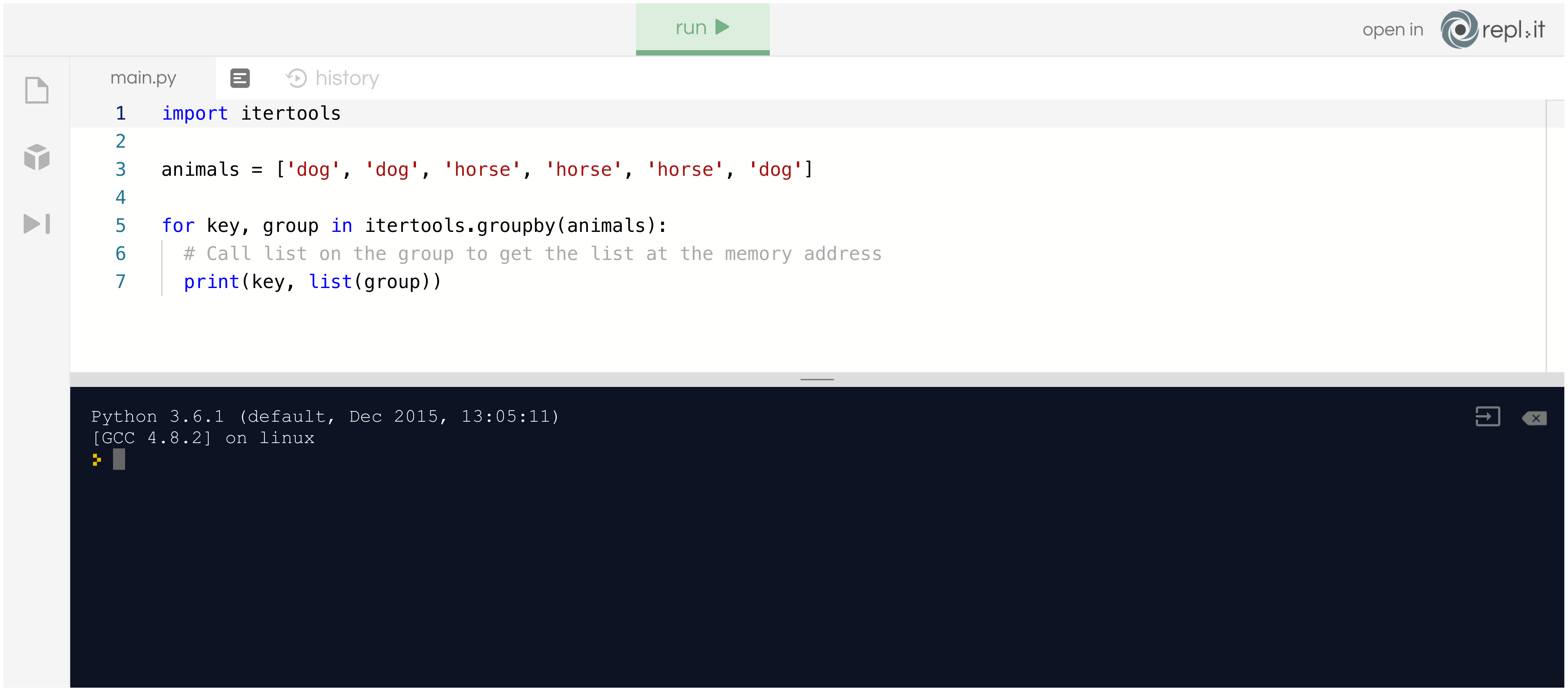
```python
# Tell Python we're using itertools
import itertools

# Make our list
animals = ['dog', 'dog', 'horse', 'horse', 'horse', 'dog']

# We are using groupby, but have to tell Python it came from itertools.
for key, group in itertools.groupby(animals):
  # Key - the name of the group. Group - the items in it.
  print(key, group)
```

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

Memory Addresses

Everything on your computer has to be stored somewhere! Computers track where things are by assigning them *memory addresses*. This way, when you want to open a picture or file, your computer knows exactly where to look.

But that memory address isn't useful. We can use `list()` to change the address back into a list. *(`list()` is explicit typecasting; do you remember it?)*



```python
import itertools

animals = ['dog', 'dog', 'horse', 'horse', 'horse', 'dog']

for key, group in itertools.groupby(animals):
  # Call list on the group to get the list at the memory address
  print(key, list(group))
```

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

Discussion: Why Is `dog` There Twice?

This is our original list:

```
animals = ['dog', 'dog', 'horse', 'horse', 'horse', 'dog']
```

`groupby()` gives us this:

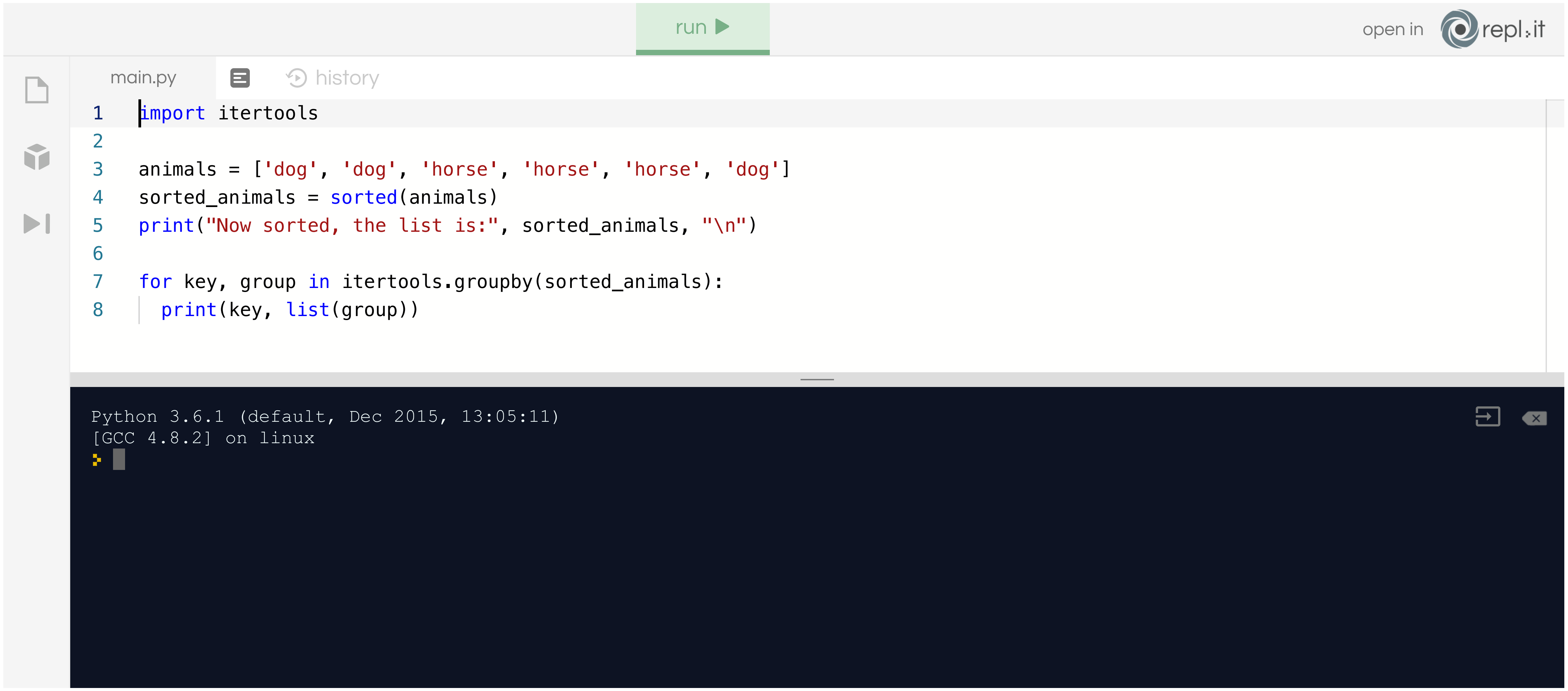
```
dog ['dog', 'dog']

horse ['horse', 'horse', 'horse']

dog ['dog']
```

Can anyone guess why `dog` is listed twice?

Sorting

`groupby()` is great, but not perfect! It will only group consecutive items. **Always** run `groupby()` on a sorted list (if you forget, you'll remember when `groupby()` returns something strange!).

Can Python sort lists? - Yes! Everything useful is built in. - There's a `sorted()` function: `new_sorted_list = sorted(list_to_be_sorted)`.



```python
import itertools

animals = ['dog', 'dog', 'horse', 'horse', 'horse', 'dog']
sorted_animals = sorted(animals)
print("Now sorted, the list is:", sorted_animals, "\n")

for key, group in itertools.groupby(sorted_animals):
  print(key, list(group))
```

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

Where Could `groupby()` Be Useful?

What if we had a list of tuples? It’s a bit hard to read.

```
things_tuple = [("animal", "wolf"), ("animal", "sparrow"), ("plant", "cactus
```

We could use `groupby()` to get this:

```
animal:
wolf is a animal
sparrow is a animal

plant:
cactus is a plant

vehicle:
yacht is a vehicle
school bus is a vehicle
car is a vehicle
```

Quick Review

We've looked at our first itertool, `groupby()`. It groups things in lists, tuples, etc. — any collection — by keys.

- `key`: The name of the group (in this case `dog` and `horse`).
- `group`: A list containing all occurrences of that key from the original list.

`groupby()` needs to be run on something sorted. We can sort with another built-in function: `sorted(list_to_be_sorted)`.

Quick Review

We only worked on lists, but tuples are a better use case for `groupby()`. `groupby()` can be run on any collection.

```python
import itertools


animals = ['dog', 'dog', 'horse', 'horse', 'horse', 'dog']
sorted_animals = sorted(animals)
print("Now sorted, the list is:", sorted_animals, "\n")


for key, group in itertools.groupby(sorted_animals):
  print(key, list(group))
```

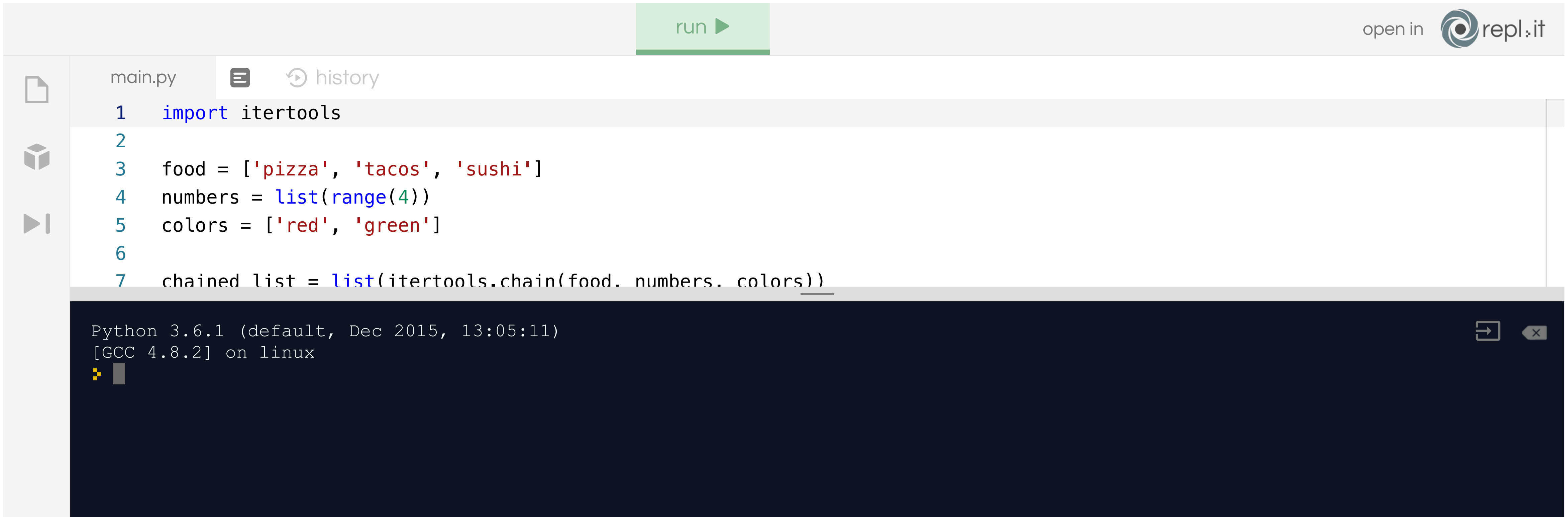
Up next: `chain()`!

A New Itertool: `chain()`

With `itertools`, we can **chain** lists:

```
food = ['pizza', 'tacos', 'sushi']

colors = ['red', 'green']

# => lists_chained =['pizza', 'tacos', 'sushi', 'red', 'green']
```

The `chain()` function takes any number of lists or sequences as parameters to turn into one. - `chained_list = list(itertools.chain(list1, list2, list3))`



```
import itertools

food = ['pizza', 'tacos', 'sushi']
numbers = list(range(4))
colors = ['red', 'green']

chained_list = list(itertools.chain(food, numbers, colors))
```

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

What Happened to the Plus Operator?

Question: Why not just use `+`?

```
chained_list = food + numbers + colors

print(chained_list)
```

Answer 1: `itertools.chain` is more efficient — it's faster, even if it's still too fast for you to notice the difference.

What Happened to the Plus Operator?

Answer 2: `itertools.chain` can contain different types of iterables.

```
import itertools


food_list = ["apples", "bananas", "oranges"]
numbers_range = range(4)
colors_dictionary = {
  "green": "peaceful",
  "blue": "calm",
  "red": "passionate"
}


# ✅ THIS WORKS. YAY!
```

You Do: `chain()`

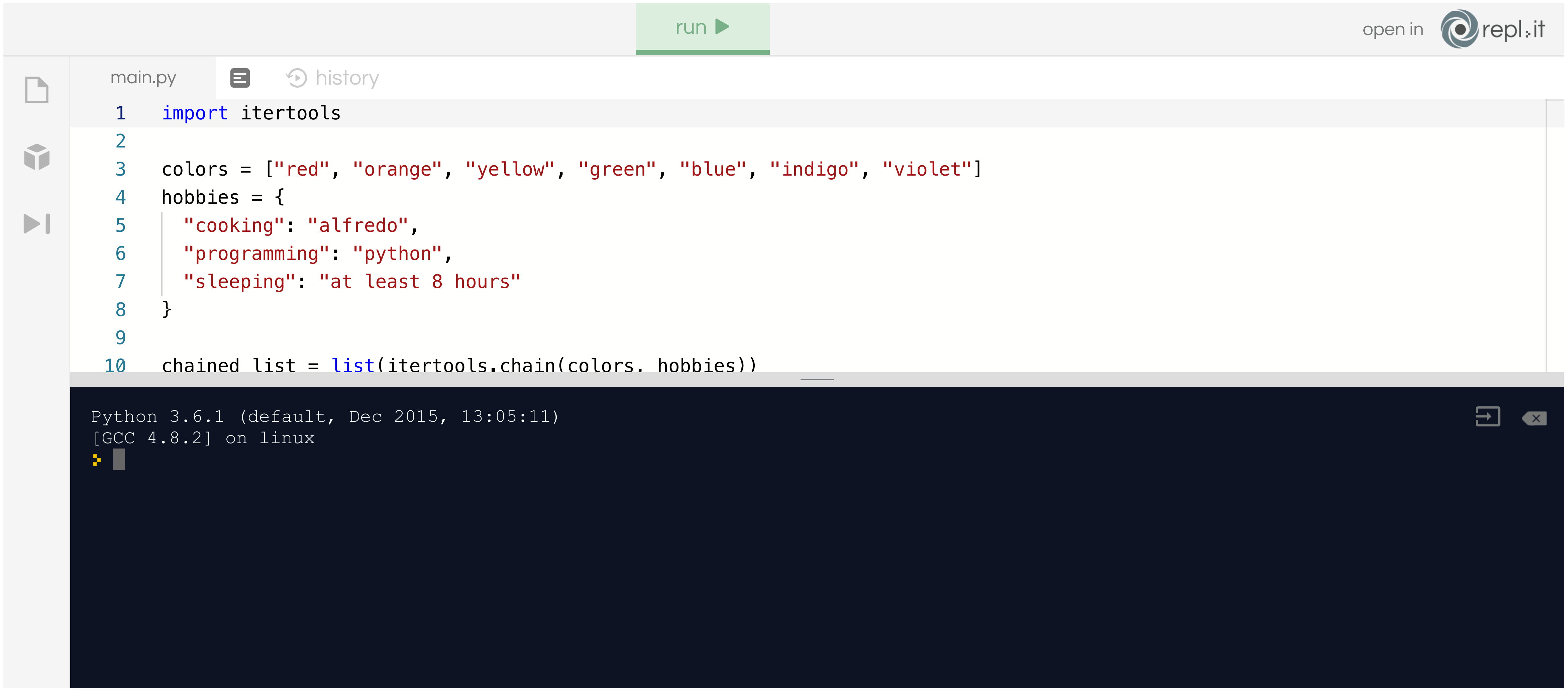
Create a local file, `my_itertools.py`. Put this at the top:

```python
import itertools
```

Below that:

- Create a list of colors.
- Create a dictionary of hobbies.
- Chain them together.
- Print out the chain!

`chain()` Answer



```python
import itertools

colors = ["red", "orange", "yellow", "green", "blue", "indigo", "violet"]
hobbies = {
  "cooking": "alfredo",
  "programming": "python",
  "sleeping": "at least 8 hours"
}

chained_list = list(itertools.chain(colors, hobbies))
```

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

Quick Review

Our second `itertool` is `chain()`, which puts lists and other collections together.

The `chain()` function takes any number of lists or sequences as parameters to turn into one.

```
import itertools

food_list = ["apples", "bananas", "oranges"]
numbers_range = range(4)
colors_dictionary = {
  "green": "peaceful",
  "blue": "calm",
  "red": "passionate"
}

chained_list = list(itertools.chain(food_list, numbers_range, colors_diction
```

Up next: `accumulate()`!

A New Itertool: `accumulate()`

What else can we do with `itertools`? - We have `groupby()` and `chain()`.

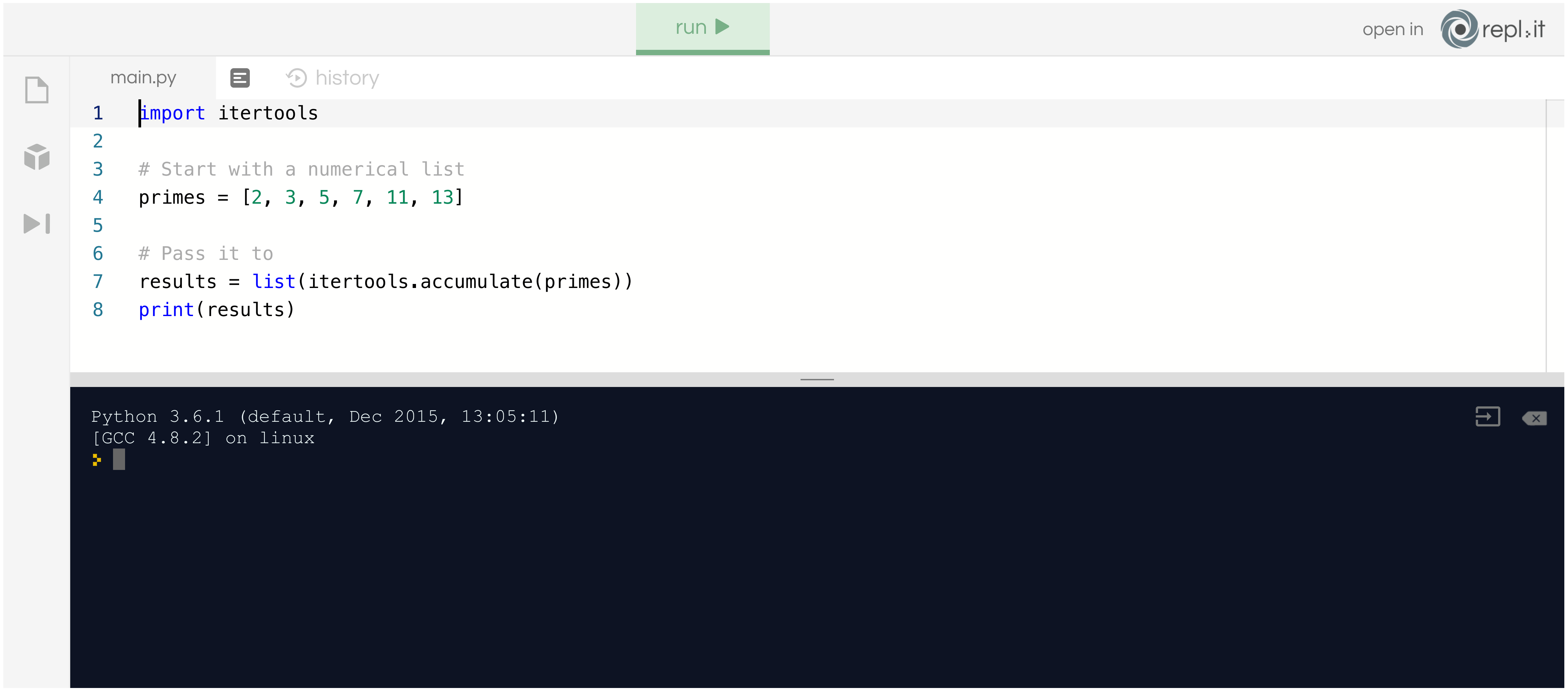
We can **accumulate** elements — add each index as it goes, making a new list with all the sums.

```
primes = [2, 3, 5, 7, 11, 13]
# => primes_added = [2, 5, 10, 17, 28, 41]


# How? It adds what's before it.
# [(2), (2+3=5), (5+5=10), (10+7=17), (17+11=28), (28+13=41)]
```

Pro tip: It's like the Fibonacci sequence!

Working Through `accumulate()`

Run this. Try changing the numbers! Set some to negative or floats.



```python
import itertools

# Start with a numerical list
primes = [2, 3, 5, 7, 11, 13]

# Pass it to
results = list(itertools.accumulate(primes))
print(results)
```

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

Quick Review

Those are all the `itertools` we're going to cover!

- `groupby()`: Grouping items in our list or collection.
- `chain()`: Concat lists or collections into one longer list.
- `accumulate()`: Add each element throughout a list, making a new list.

```
### Chain ###
food = ['pizza', 'tacos', 'sushi']
colors = ['red', 'green']
# => lists_chained =['pizza', 'tacos', 'sushi', 'red', 'green']


### Groupby ###
# Make our list.
animals = ['dog', 'dog', 'horse', 'horse', 'horse', 'dog']
for key, group in itertools.groupby(animals):
  # Key: the name of the group. Group: the items in it.
  print(key, group)
```

Up next: List comprehensions.

Changing Gears: Modifying a List

`itertools` provides abstraction for iterating over lists. We’re done with them!

Let’s move on. What about building a new list that’s slightly modified from another list? This is *extremely* common, so Python provides us with **list comprehensions**.

For anything where you can make:

```
for item in old_list:
    if < condition >
      new_list.append(< modification >)
```

You can use list comprehension syntax instead:

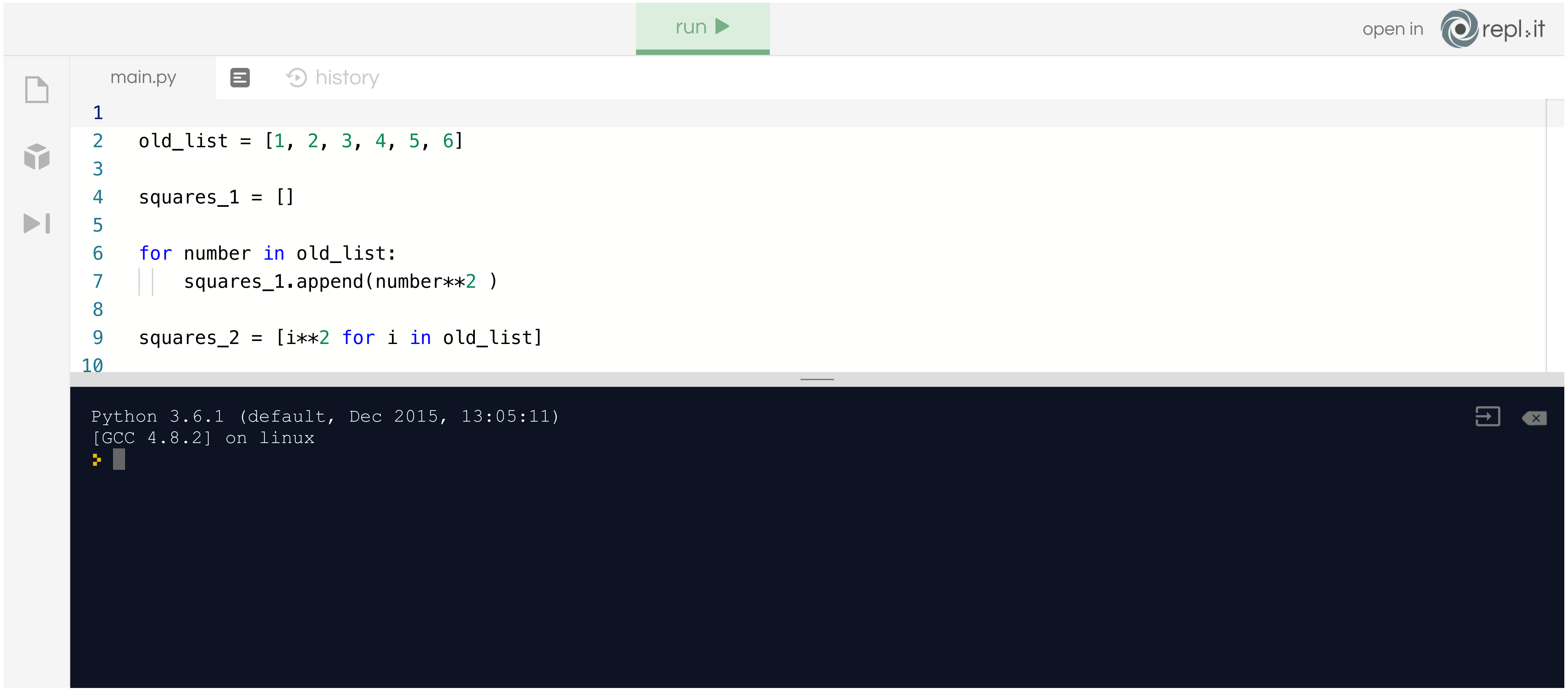
```
new_list = [modification  old_list  [condition]]
```

It turns three lines of code into one!

Example: List Comprehension

So, instead of our `for` loop, we can have `# new_list = [modification old_list [condition]]`.

Let's run this. Try changing the list or modification.



```python

old_list = [1, 2, 3, 4, 5, 6]

squares_1 = []

for number in old_list:
    squares_1.append(number**2 )

squares_2 = [i**2 for i in old_list]

```

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

List Comprehensions With a Conditional

How could we only square the even numbers?

We're familiar with a loop:

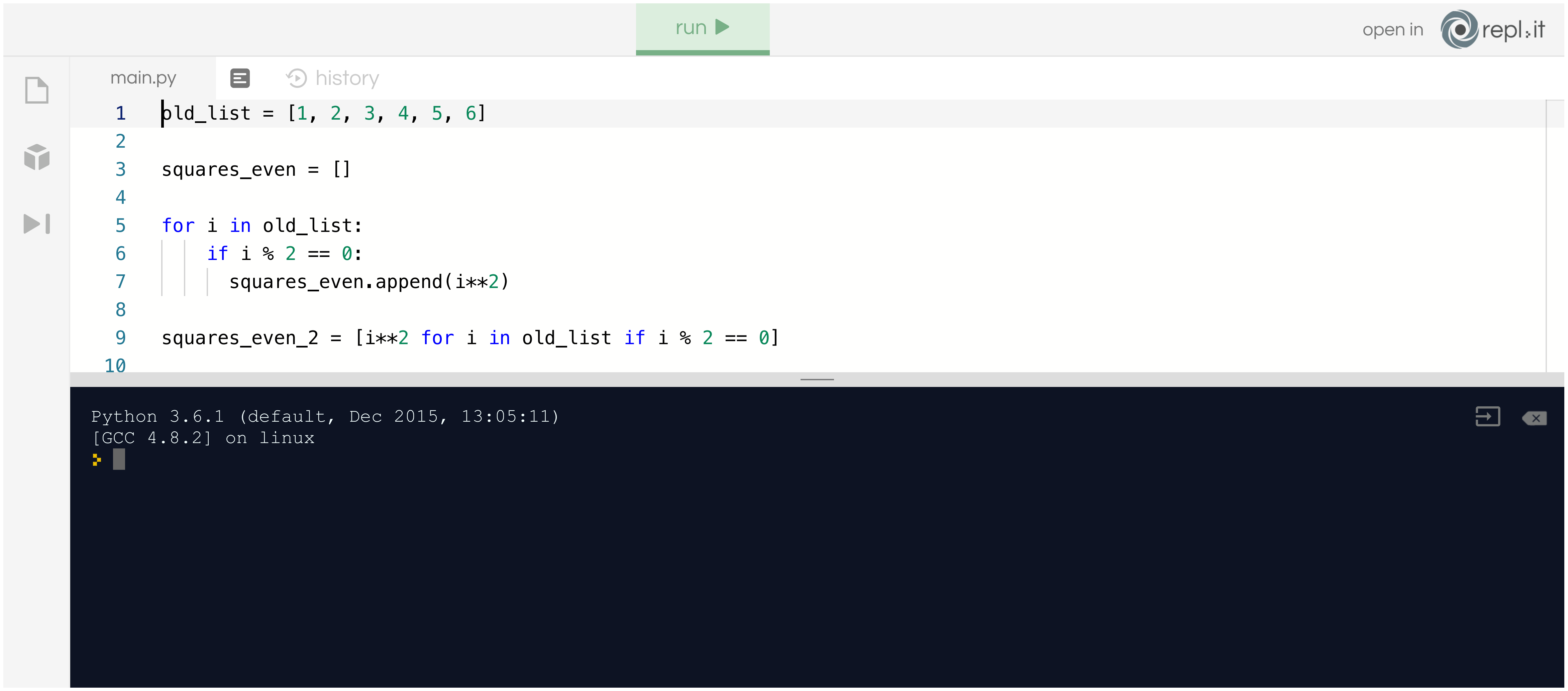
```
# All squares
for i in old_list:  # old list
    squares.append(i**2)  # modification


# Even squares
for i in old_list:  # Old list
    if i % 2 == 0:   # Conditional
      squares_even.append(i**2) # Modification
```

Now, in a list comprehension:

```
# new_list = [modification  old_list  [condition]]


squares = [i**2 for i in old_list]
```

Example: List Comprehension and Conditionals

Let's run this. Try changing the list, modification, or conditional. It's `# new_list = [modification old_list [condition]]`.



```python
old_list = [1, 2, 3, 4, 5, 6]

squares_even = []

for i in old_list:
    if i % 2 == 0:
        squares_even.append(i**2)

squares_even_2 = [i**2 for i in old_list if i % 2 == 0]
```

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

Discussion: More Conditionals Practice

We're not limited to math or numerical lists! Any list will work and any `if` conditional will work.

If you can make:

```
for item in old_list:
    if < condition >
      new_list.append(< modification >)
```

Then you can make:

```
new_list = [modification  old_list_iteration  [condition]]
```

Discussion: More Conditionals Practice

Let’s say we have a string containing both numbers and letters:

```
my_string = '99 fantastic 13 hello 2 world'
```

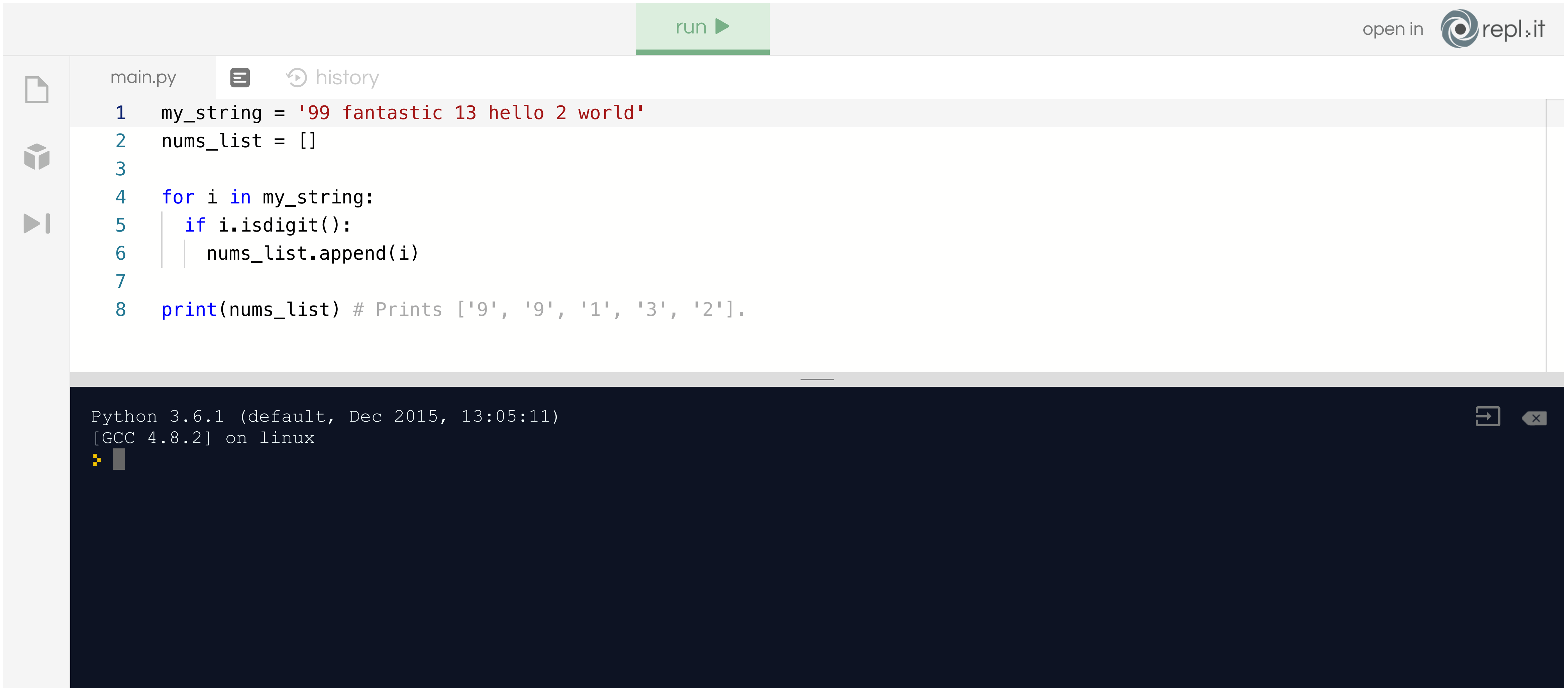
We want to write a list comprehension that will make a new list containing only the numbers that appear.

- What is our `modification`?
- What is our `old_list_iteration`?
- What is our `condition`?

Partner Exercise: Creating the List Comprehension

Get with a partner! Pick a driver.

Below, turn the `for` loop into a list comprehension. Discuss with them: Why doesn't it print `[99, 13, 2]`?



```python
my_string = '99 fantastic 13 hello 2 world'
nums_list = []

for i in my_string:
  if i.isdigit():
    nums_list.append(i)

print(nums_list) # Prints ['9', '9', '1', '3', '2'].
```

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

Summary and Q&A:

Code abstraction: Shortcut functions provided by Python for common tasks.

`itertools`:

- Abstraction for loops and iterating.
- `groupby()`: Creates groups of elements in a list matching a key. Sort elements first!
  - `animals = ['dog', 'dog', 'dog', 'horse', 'horse', 'horse']` and `for key, group in itertools.groupby(animals)` creates `dog: ['dog', 'dog', 'dog'], horse: ['horse', 'horse']`
- `chain()`: Creates one long list from many lists.
  - `chained_list = list(itertools.chain(list1, list2, list3))`

Summary and Q&A:

- `accumulate()`: Performs some operation on a list and returns the accumulated results.
    - `results = list(itertools.accumulate(primes))`

List comprehensions: - Abstraction for creating a slightly modified list. - `new_list = [modification old_list_iteration [condition]]`

Additional Reading

- What Is `itertools` and Why Should I Use It?
- `groupby()` Docs
- `chain()` and Other `itertools`
- Comprehending List Comprehensions