More Types

None

The **None** object is used to represent the absence of a value.
It is similar to **null** in other programming languages.
Like other "empty" values, such as 0, [] and the empty string, it is **False** when converted to a Boolean variable.
When entered at the Python console, it is displayed as the empty string.

```
>>> None == None
True
>>> None
>>> print(None)
None
>>>
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

None

The **None** object is returned by any function that doesn't explicitly return anything else.

```
def some_func():
  print("Hi!")

var = some_func()
print(var)
```

Try It Yourself

Result:

```
>>>
Hi!
None
>>>
```

Tap **Try It Yourself** to play around with the code!

Dictionaries

Dictionaries are data structures used to map arbitrary keys to values.
Lists can be thought of as dictionaries with integer keys within a certain range.
Dictionaries can be indexed in the same way as lists, using **square brackets** containing keys.
Example:

```
ages = {"Dave": 24, "Mary": 42, "John": 58}
print(ages["Dave"])
print(ages["Mary"])
```

Try It Yourself

Result:

```
>>>
24
42
>>>
```

> Each element in a dictionary is represented by a **key:value** pair.

Dictionaries

Trying to index a key that isn't part of the dictionary returns a **KeyError**.
Example:

```
primary = {
  "red": [255, 0, 0],
  "green": [0, 255, 0],
  "blue": [0, 0, 255],
}

print(primary["red"])
print(primary["yellow"])
```

Try It Yourself

Result:

```
>>>
[255, 0, 0]

KeyError: 'yellow'
>>>
```

As you can see, a dictionary can store any types of data as values.

> An empty dictionary is defined as **{}**.

Dictionaries

Only immutable objects can be used as keys to dictionaries. **Immutable** objects are those that can't be changed. So far, the only mutable objects you've come across are **lists** and **dictionaries**. Trying to use a mutable object as a dictionary key causes a **TypeError**.

```
bad_dict = {
  [1, 2, 3]: "one two three",
}
```

Try It Yourself

Result:

```
>>>
TypeError: unhashable type: 'list'
>>>
```

Tap **Try It Yourself** to play around with the code!

Dictionaries

Just like lists, dictionary keys can be assigned to different values.
However, unlike lists, a new dictionary key can also be assigned a value, not just ones that already exist.

```
squares = {1: 1, 2: 4, 3: "error", 4: 16,}
squares[8] = 64
squares[3] = 9
print(squares)
```

Try It Yourself

Result:

```
{1: 1, 2: 4, 3: 9, 4: 16, 8: 64}
```

Tap **Try It Yourself** to play around with the code!

Dictionaries

To determine whether a key is in a dictionary, you can use **in** and **not in**, just as you can for a list.
Example:

```
nums = {
  1: "one",
  2: "two",
  3: "three",
}
print(1 in nums)
print("three" in nums)
print(4 not in nums)
```

Try It Yourself

Result:

```
>>>
True
False
True
>>>
```

Tap **Try It Yourself** to play around with the code!

Dictionaries

A useful dictionary method is **get**. It does the same thing as indexing, but if the key is not found in the dictionary it returns another specified value instead ('None', by default).
Example:

```
pairs = {1: "apple",
  "orange": [2, 3, 4],
  True: False,
  None: "True",
}

print(pairs.get("orange"))
print(pairs.get(7))
print(pairs.get(12345, "not in dictionary"))
```

Try It Yourself

Result:

```
>>>
[2, 3, 4]
None
not in dictionary
>>>
```

Tap **Try It Yourself** to play around with the code!

Tuples

Tuples are very similar to lists, except that they are immutable (they cannot be changed).
Also, they are created using **parentheses**, rather than square brackets.
Example:

```
words = ("spam", "eggs", "sausages",)
```

You can access the values in the tuple with their index, just as you did with lists:

```
print(words[0])
```

Try It Yourself

Trying to reassign a value in a tuple causes a TypeError.

```
words[1] = "cheese"
```

Try It Yourself

Result:

```
>>>
TypeError: 'tuple' object does not support item assignment
>>>
```

Like lists and dictionaries, tuples can be nested within each other.

Tuples

Tuples can be created without the parentheses, by just separating the values with commas.
Example:

```
my_tuple = "one", "two", "three"
print(my_tuple[0])
```

Try It Yourself

Result:

```
>>>
one
>>>
```

An empty tuple is created using an empty parenthesis pair.

```
tpl = ()
```

Tuples are faster than lists, but they cannot be changed.

List Slices

List slices provide a more advanced way of retrieving values from a list. Basic list slicing involves indexing a list with **two colon-separated integers**. This returns a new list containing all the values in the old list between the indices.
Example:

```
squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(squares[2:6])
print(squares[3:8])
print(squares[0:1])
```

Try It Yourself

Result:

```
>>>
[4, 9, 16, 25]
[9, 16, 25, 36, 49]
[0]
>>>
```

Like the arguments to **range**, the first index provided in a slice is included in the result, but the second isn't.

List Slices

If the first number in a slice is omitted, it is taken to be the start of the list.
If the second number is omitted, it is taken to be the end.
Example:

```
squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(squares[:7])
print(squares[7:])
```

Try It Yourself

Result:

```
>>>
[0, 1, 4, 9, 16, 25, 36]
[49, 64, 81]
>>>
```

> Slicing can also be done on tuples.

List Slices

List slices can also have a third number, representing the step, to include only alternate values in the slice.

```
squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(squares[::2])
print(squares[2:8:3])
```

Try It Yourself

Result:

```
>>>
[0, 4, 16, 36, 64]
[4, 25]
>>>
```

> [2:8:3] will include elements starting from the 2nd index up to the 8th with a step of 3.

List Slices

Negative values can be used in list slicing (and normal list indexing). When negative values are used for the first and second values in a slice (or a normal index), they count from the end of the list.

```
squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(squares[1:-1])
```

Try It Yourself

Result:

```
>>>
[1, 4, 9, 16, 25, 36, 49, 64]
>>>
```

> If a negative value is used for the step, the slice is done backwards.
> Using **[::-1]** as a slice is a common and idiomatic way to reverse a list.

List Comprehensions

List comprehensions are a useful way of quickly creating lists whose contents obey a simple rule.
For example, we can do the following:

```
# a list comprehension
cubes = [i**3 for i in range(5)]

print(cubes)
```

Try It Yourself

Result:

```
>>>
[0, 1, 8, 27, 64]
>>>
```

> List comprehensions are inspired by set-builder notation in mathematics.

List Comprehensions

A list comprehension can also contain an **if** statement to enforce a condition on values in the list.
Example:

```
evens=[i**2 for i in range(10) if i**2 % 2 == 0]

print(evens)
```

Try It Yourself

Result:

```
>>>
[0, 4, 16, 36, 64]
>>>
```

> Tap **Try It Yourself** to play around with the code!

List Comprehensions

Trying to create a list in a very extensive range will result in a **MemoryError**.
This code shows an example where the list comprehension runs out of memory.

```
even = [2*i for i in range(10**100)]
```

Try It Yourself

Result:

```
>>>
MemoryError
>>>
```

> This issue is solved by **generators**, which are covered in the next module.

String Formatting

So far, to combine strings and non-strings, you've converted the non-strings to strings and added them.
String formatting provides a more powerful way to embed non-strings within strings. String formatting uses a string's **format** method to substitute a number of arguments in the string.
Example:

```
# string formatting
nums = [4, 5, 6]
msg = "Numbers: {0} {1} {2}". format(nums[0], nums[1], nums[2])
print(msg)
```

Try It Yourself

Result:

```
>>>
Numbers: 4 5 6
>>>
```

> Each argument of the format function is placed in the string at the corresponding position, which is determined using the curly braces { }.

String Formatting

String formatting can also be done with named arguments.
Example:

```
a = "{x}, {y}".format(x=5, y=12)
print(a)
```

Try It Yourself

Result:

```
>>>
5, 12
>>>
```

> Tap **Try It Yourself** to play around with the code!

String Functions

Python contains many useful built-in functions and methods to accomplish common tasks.
join - joins a list of strings with another string as a separator.
replace - replaces one substring in a string with another.
startswith and **endswith** - determine if there is a substring at the start and end of a string, respectively.
To change the case of a string, you can use **lower** and **upper**.
The method **split** is the opposite of **join**, turning a string with a certain separator into a list.
Some examples:

```
print(", ".join(["spam", "eggs", "ham"]))
#prints "spam, eggs, ham"

print("Hello ME".replace("ME", "world"))
#prints "Hello world"

print("This is a sentence.".startswith("This"))
# prints "True"

print("This is a sentence.".endswith("sentence."))
# prints "True"

print("This is a sentence.".upper())
# prints "THIS IS A SENTENCE."

print("AN ALL CAPS SENTENCE".lower())
#prints "an all caps sentence"

print("spam, eggs, ham".split(", "))
#prints "['spam', 'eggs', 'ham']"
```

Try It Yourself

> Tap **Try It Yourself** to play around with the code!

Numeric Functions

To find the maximum or minimum of some numbers or a list, you can use **max** or **min**.
To find the distance of a number from zero (its absolute value), use **abs**.
To round a number to a certain number of decimal places, use **round**.
To find the total of a list, use **sum**.
Some examples:

```
print(min(1, 2, 3, 4, 0, 2, 1))
print(max([1, 4, 9, 2, 5, 6, 8]))
print(abs(-99))
print(abs(42))
print(sum([1, 2, 3, 4, 5]))
```

Try It Yourself

Result:

```
>>>
0
9
99
42
15
>>>
```

Tap **Try It Yourself** to play around with the code!

List Functions

Often used in conditional statements, **all** and **any** take a list as an argument, and return **True** if all or any (respectively) of their arguments evaluate to **True** (and **False** otherwise).
The function **enumerate** can be used to iterate through the values and indices of a list simultaneously.
Example:

```
nums = [55, 44, 33, 22, 11]

if all([i > 5 for i in nums]):
  print("All larger than 5")

if any([i % 2 == 0 for i in nums]):
  print("At least one is even")

for v in enumerate(nums):
  print(v)
```

Try It Yourself

Result:

```
>>>
All larger than 5
At least one is even
(0, 55)
(1, 44)
(2, 33)
(3, 22)
(4, 11)
>>>
```

Tap **Try It Yourself** to play around with the code!

Text Analyzer

This is an example project, showing a program that analyzes a sample file to find what percentage of the text each character occupies.
This section shows how a file could be open and read.

```
filename = input("Enter a filename: ")

with open(filename) as f:
  text = f.read()

print(text)
```

Result:

```
>>>
Enter a filename: test.txt
Ornhgvshy vf orggre guna htyl.
Rkcyvpvg vf orggre guna vzcyvpvg.
Fvzcyr vf orggre guna pbzcyvpngrq.
Syng vf orggre guna arfgrq.
Fcenfr fv orggre guna qrafr.
Ernqnovyvgl pbhagf.
Fcrpvny pnfrf nera'g fcrpvny rabthu gb oernx gur ehyrf.
Nygubhtu cenpgvpnyvgl orgnf chevgl.
Reebef fubhyq arire cnff fvyragyl.
Hayrff rkcyvpvgyl fvyraprq.
Va gur snpr bs nzovthvgl, ershfr gur grzcgngvba bg thrff.
Gurer fubhyq or bar-- naq cersrenoylbayl bar --boivbhf jnl gb qb vg.
Nygubhtu gung jnl znl abg or boivbhf ng svefg hayrff lbh'er Qhgpu.
Abj vf orggre guna arrire.
Nygubhtu arire vf bsgra orggre guna *evtug* abj.
Vs gur vzcyrzragngvba vf uneq gb rkcynva, vg'f n onq vqrn.
Vs gur vzcyrzragngvba vf rnfl gb rkcynva, vg znl or n tbbq vqrn.
Anzrfcnprf ner bar ubaxvat terng vqrn -- yrg'f qb zber bs gubfr!
```

This is sample content for demonstration purposes only.

Text Analyzer

This part of the program shows a function that counts how many times a character occurs in a string.

```
def count_char(text, char):
  count = 0
  for c in text:
    if c == char:
      count += 1
  return count
```

This function takes as its arguments the text of the file and one character, returning the number of times that character appears in the text.
Now we can call it for our file.

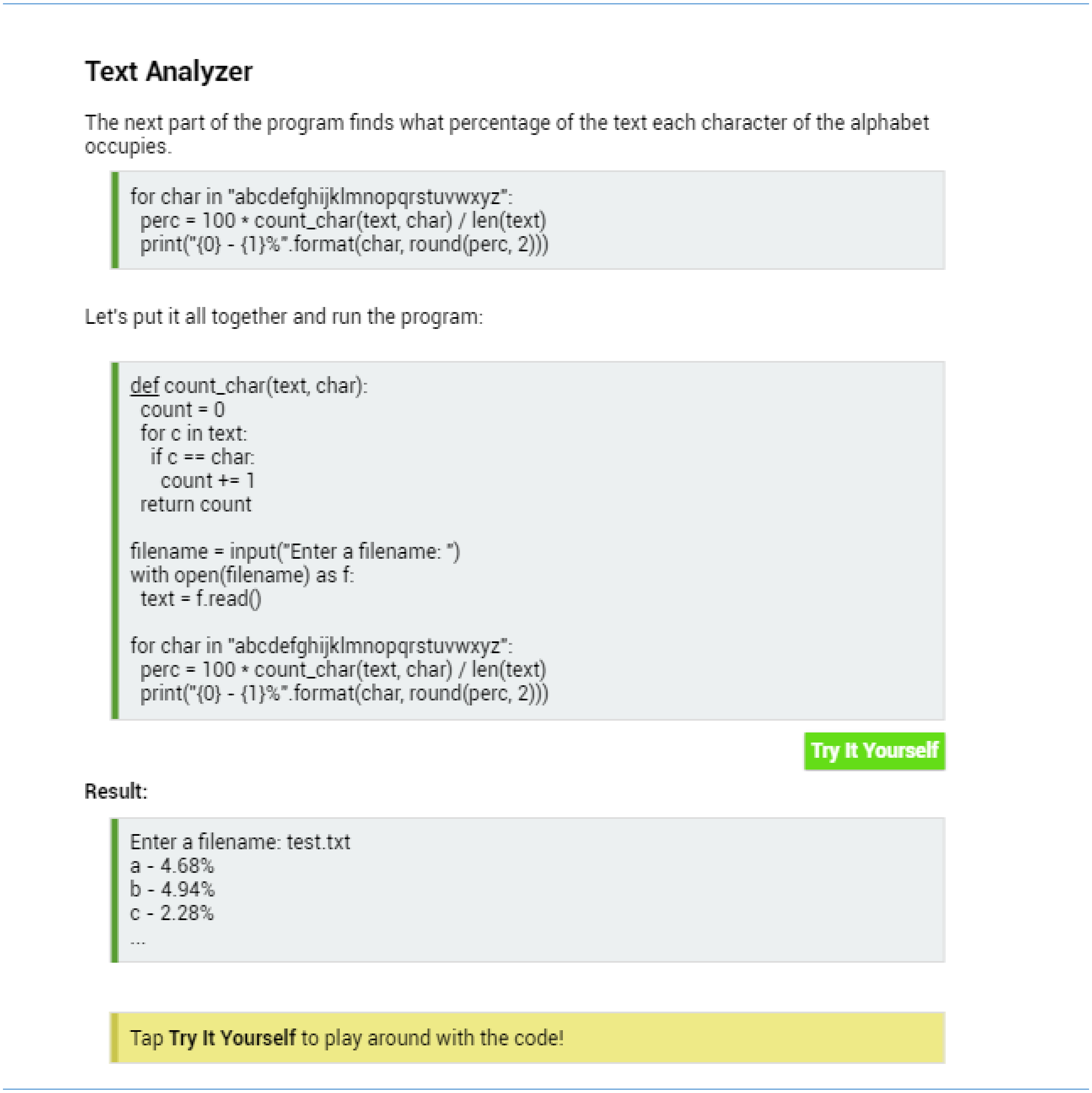
```
filename = input("Enter a filename: ")
with open(filename) as f:
  text = f.read()

print(count_char(text, "r"))
```

Result:

```
>>>
Enter a filename: test.txt
83
>>>
```

The character "r" appears 83 times in the file.



Text Analyzer

The next part of the program finds what percentage of the text each character of the alphabet occupies.

```
for char in "abcdefghijklmnopqrstuvwxyz":
 perc = 100 * count_char(text, char) / len(text)
 print("{0} - {1}%".format(char, round(perc, 2)))
```

Let's put it all together and run the program:

```
def count_char(text, char):
 count = 0
 for c in text:
  if c == char:
   count += 1
 return count

filename = input("Enter a filename: ")
with open(filename) as f:
 text = f.read()

for char in "abcdefghijklmnopqrstuvwxyz":
 perc = 100 * count_char(text, char) / len(text)
 print("{0} - {1}%".format(char, round(perc, 2)))
```

Result:

```
Enter a filename: test.txt
a - 4.68%
b - 4.94%
c - 2.28%
...
```

Tap **Try It Yourself** to play around with the code!

End.