Topics Covered

- String basics.
- Special characters.
- Multi-line strings.
- Indexing and slicing strings.
- Common string operators and methods.
- Formatting strings.
- Built-in string functions.

---

*'Yes,' they say to one another, these so kind ladies, 'he is a stupid old fellow, he will see not what we do, he will never observe that his sock heels go not in holes any more, he will think his buttons grow out new when they fall, and believe that **strings** make theirselves.'*

– Little Women, Louisa May Alcott

According to the Python documentation[18], "Strings are immutable sequences of Unicode code points." Less technically speaking, strings are sequences of characters.[19] The term *sequence* in Python refers to an ordered set. Other common sequence types are lists, tuples, and ranges, all of which we will cover.

4.1. Quotation Marks and Special Characters

Strings can be created with single quotes or double quotes. There is no difference between the two.

4.1.1. Escaping Characters

To create a string that contains a single quote (e.g., `Where'd you get the coconuts?`), enclose the string in double quotes:

```
phrase = "Where'd you get the coconuts?"
```

Likewise, to create a string that contains a double quote (e.g., `The soldier asked, "Are you suggesting coconuts migrate?"`), enclose the string in single quotes:

```
phrase = 'The soldier asked, "Are you suggesting coconuts migrate?"'
```

Sometimes, you will want to output single quotes within a string denoted by single quotes or double quotes within a string denoted by double quotes. In such cases, you will need to *escape* the quotation marks using a backslash (`\`), like this:

```
>>> phrase = "The soldier said, \"You've got two empty halves of a coconut.\""
>>> print(phrase)
The soldier said, "You've got two empty halves of a coconut."
```

Or:

```
>>> phrase = 'The soldier said, "You\'ve got two empty halves of a coconut."'
>>> print(phrase)
The soldier said, "You've got two empty halves of a coconut."
```

Notice that the printed output does not contain the backslashes.

Special Characters

The backslash can also be used to escape characters with special meaning, such as the backslash itself:

```
>>> phrase = "Use an extra backslash to output a backslash: \\"
>>> print(phrase)
Use an extra backslash to output a backslash: \
```

Two other common special characters are the newline (`\n`) and horizontal tab (`\t`):

```
>>> print('Equation\tSolution\n 55 x 11\t 605\n 132 / 6\t 22')
Equation        Solution
 55 x 11         605
 132 / 6         22
```

Escape Sequences

Escape Sequence	Meaning
\'	Single quote
\"	Double quote
\\	Backslash
\n	Newline
\t	Horizontal tab

Raw Strings

Sometimes, a string might have a lot of backslashes in it that are just meant to be plain old backslashes. The most common example is a file path. For example:

```
'C:\news\today.txt'
```

Watch what happens when that string is assigned to a variable:

```
>>> my_path = 'C:\news\today.txt'
>>> print(my_path)
C:
ews     oday.txt
```

When we print `my_path`, the `\n` and `\t` characters get printed as a newline and a tab.

Using the "r" (for raw data) prefix on the string, we ensure that all backslashes are escaped:

```
>>> my_path = r'C:\news\today.txt'
>>> print(my_path)
C:\news\today.txt
```

If you examine the variable directly without printing it, you see that each backslash is escaped with another backslash:

```
>>> my_path
'C:\\news\\today.txt'
```

Note that backslashes will always escape single and double quotation marks, even in a raw string, so you cannot end a raw string with a single backslash:

Bad

```
my_path = r'C:\my\new\'
```

Good

```
my_path = r'C:\my\new\\'
```

Raw strings can come in particularly handy when working with regular expressions.

4.1.2. Triple Quotes

Triple quotes are used to create multi-line strings. You generally use three double quotes[20] as shown in the following example:

Demo 4.1: strings/Demos/triple_quotes.py

```
print("""----------------
LUMBERJACK SONG

I'm a lumberjack
And I'm O.K.
I sleep all night
And I work all day.
----------------""")
```

The preceding code will render the following:

```
----------------
LUMBERJACK SONG

I'm a lumberjack
And I'm O.K.
I sleep all night
And I work all day.
----------------
```

Note that quotation marks can be included within triple-quoted strings without being escaped with a backslash.

In some cases when using triple quotes, you may want to break up your code with a newline without having that newline show up in your output. You can escape the actual newline with a backslash as shown in the following demo:

Demo 4.2: strings/Demos/triple_quotes_newline_escaped.py

```
print("""We're knights of the Round Table, \
we dance whene'er we're able.
We do routines and chorus scenes \
with footwork impeccable,
We dine well here in Camelot, \
we eat ham and jam and Spam a lot.""")
```

The preceding code will render the following:

```
We're knights of the Round Table, we dance whene'er we're able.
We do routines and chorus scenes with footwork impeccable,
We dine well here in Camelot, we eat ham and jam and Spam a lot.
```

Notice that the backslashes at the end of lines 1, 3, and 5 prevent line breaks in the output.

> **Spacing in ebooks**
>
> Because ebooks do not have a set font size, it is impossible for authors to control spacing and line breaks. So, the code above (and other code samples throughout this book) may wrap in places where it would not actually wrap when programming. We're sorry about that! The best way to see how it will really look is to run these code samples on your computer.

4.2. String Indexing

Indexing is the process of finding a specific element within a sequence of elements through the element's position. Remember that strings are basically sequences of characters. We can use indexing to find a specific character within the string.

If we consider the string from left to right, the first character (the left-most) is at position 0. If we consider the string from right to left, the first character (the right-most) is at position `-1`. The following table illustrates this for the phrase "MONTY PYTHON".

	M	O	N	T	Y		P	Y	T	H	O	N

	M	O	N	T	Y		P	Y	T	H	O	N
Left to Right:	0	1	2	3	4	5	6	7	8	9	10	11
Right to Left:	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

The following demonstration shows how to find characters by position using indexing.

Demo 4.3: strings/Demos/string_indexing.py

```
phrase = "Monty Python"

first_letter = phrase[0] # [M]onty Python
print(first_letter)

last_letter = phrase[-1] # Monty Pytho[n]
print(last_letter)

fifth_letter = phrase[4] # Mont[y] Python
print(fifth_letter)

third_letter_from_end = phrase[-3] # Monty Pyt[h]on
print(third_letter_from_end)
```

The expected output for each `print` statement is shown in square brackets in the comment. Running the file should result in:

```
M
n
y
h
```

Exercise 11: Indexing Strings

10 to 20 minutes.

In this exercise, you will write a program that gets a specific character from a phrase entered by the user.

1. Open strings/Exercises/indexing.py.
2. Modify the `main()` function so that it:
   A. Prompts the user to enter a phrase.
   B. Tells the user what phrase they entered (e.g., `Your phrase is 'Hello, World!'`).
   C. Prompts the user for a number.
   D. Tells the user what character is at that position in the user's phrase (e.g., `Character number 4 is o`).
3. Here is the program completed by the user:

```
Choose a phrase: Hello, world!
Your phrase is 'Hello, world!'
Which character? [Enter number] 4
Character 4 is o
```

Challenge

As a Python programmer, you understand that the "o" in "Hello" is at position 4, because Python starts counting with 0. However, regular people will think that the character at position 4 is "l" and they will think your program is wrong. Fix your program so that it responds as the user expects. Also, to make it a little prettier, output the character in single quotes.

The program should work like this:

```
Choose a phrase: Hello, world!
Your phrase is 'Hello, world!'
Which character? [Enter number] 4
Character 4 is 'l'
```

Solution: strings/Solutions/indexing.py

```
def main():
    phrase = input("Choose a phrase: ")
    print("Your phrase is '", phrase, "'", sep="")
    pos = int(input("Which character? [Enter number] "))
    print("Character number", pos, "is", phrase[pos])

main()
```

Challenge Solution: strings/Solutions/indexing_challenge.py

```
def main():
    phrase = input("Choose a phrase: ")
    print("Your phrase is '", phrase, "'", sep="")
    pos = int(input("Which character? [Enter number] "))-1
    print("Character ", pos+1, " is '", phrase[pos], "'", sep="")

main()
```

4.3. Slicing Strings

Often, you will want to get a sequence of characters from a string (i.e., a *substring*). In Python, you do this by getting a slice of the string using the following syntax:

```
substring = orig_string[first_pos:last_pos]
```

This returns a slice that starts with the character at `first_pos` and includes all the characters up to *but not including* the character at `last_pos`.

If `first_pos` is left out, then it is assumed to be `0`. So `'hello'[:3]` would return `"hel"`.

If `last_pos` is left out, then it is assumed to be the length of the string, or in other words, one more than the last position of the string. So `'hello'[3:]` would return `"lo"`.

The following demonstration shows how to get substrings using slicing.

Demo 4.4: strings/Demos/string_slicing.py

```
phrase = "Monty Python"

first_5_letters = phrase[0:5] # [Monty] Python
print(first_5_letters)

letters_2_thru_4 = phrase[1:4] # M[ont]y Python
print(letters_2_thru_4)

letter_5_to_end = phrase[4:] # Mont[y Python]
print(letter_5_to_end)

last_3_letters = phrase[-3:] # Monty Pyt[hon]
print(last_3_letters)

first_3_letters = phrase[:3] # [Mon]ty Python
print(first_3_letters)

three_letters_before_last = phrase[-4:-1] # Monty Py[tho]n
print(three_letters_before_last)

copy_of_string = phrase[:] # [Monty Python]
print(copy_of_string)
```

The expected output for each `print` statement is shown in square brackets in the comment. Running the file should result in:

```
Monty
ont
y Python
hon
Mon
```

```
tho
Monty Python
```

Exercise 12: Slicing Strings

10 to 20 minutes.

In this exercise, you will write a program that gets a substring (or slice) from a phrase entered by the user.

1. Open strings/Exercises/slicing.py.
2. Modify the `main()` function so that it:
   A. Prompts the user to enter a phrase.
   B. Tells the user what phrase they entered (e.g., `Your phrase is 'Hello, World!'`).
   C. Prompts the user for a start number.
   D. Prompts the user for a end number.
   E. Tells the user the substring (within single quotes) that starts with the start number and ends with the end number.
3. Here is the output of the program:

```
Choose a phrase: Hello, world!
Your phrase is 'Hello, world!'
Character to start with? [Enter number] 4
Character to end with? [Enter number] 9
Your substring is 'o, wor'
```

Challenge

As with the last exercise, make your program respond as users would expect.

The new program should work like this:

```
Choose a phrase: Hello, world!
Your phrase is 'Hello, world!'
Character to start with? [Enter number] 4
Character to end with? [Enter number] 9
Your substring is 'lo, wo'
```

Solution: strings/Solutions/slicing.py

```
def main():
    phrase = input("Choose a phrase: ")
    print("Your phrase is '", phrase, "'", sep="")
    pos1 = int(input("Character to start with? [Enter number] "))
    pos2 = int(input("Character to end with? [Enter number] ")) + 1
    print("Your substring is '", phrase[pos1:pos2], "'", sep="")

main()
```

Challenge Solution: strings/Solutions/slicing_challenge.py

```
def main():
    phrase = input("Choose a phrase: ")
    print("Your phrase is '", phrase, "'", sep="")
    pos1 = int(input("Character to start with? [Enter number] ")) - 1
    pos2 = int(input("Character to end with? [Enter number] "))
    print("Your substring is '", phrase[pos1:pos2], "'", sep="")

main()
```

4.4. Concatenation and Repetition

4.4.1. Concatenation

Concatenation is a fancy word for stringing strings together. In Python, concatenation is done with the + operator. It is often used to combine variables with literals as in the following example:

Demo 4.5: strings/Demos/concatenation.py

```
user_name = input("What is your name? ")
greeting = "Hello, " + user_name + "!"
print(greeting)
```

The preceding code will render the following:

```
What is your name? Nat
Hello, Nat!
```


4.4.2. Repetition

Repetition is the process of repeating a string some number of times. In Python, repetition is done with the * operator.

Demo 4.6: strings/Demos/repetition.py

```
one_knight_says = "nee"
many_knights_say = one_knight_says * 20
print(many_knights_say)
```

The preceding code will render the following:

```
neeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneenee
```

Exercise 13: Repetition

5 to 10 minutes.

Remember our `insert_separator()` function from the “Hello, You!” programs. It looked like this:

```
def insert_separator(s="="):
    print(s, s, s, sep="")
```

Using repetition, we can improve `insert_separator()` so that the number of times the separating character shows up is passed into the function.

1. Open strings/Exercises/hello_you.py in your editor.
2. Modify the `insert_separator()` function so that the number of times the separating character shows up is passed in as a parameter. It should default to show up 30 times.
3. Modify the calls to `insert_separator()` so that they pass in an argument to the new parameter.

Solution: strings/Solutions/hello_you.py

```
def say_hello(name):
    print('Hello, ', name, '!', sep='')

def insert_separator(s="=", repeat=30):
    print(s * repeat)

def recite_poem():
    print("How about a Monty Python poem?")
    insert_separator("-", 20)
    print("Much to his Mum and Dad’s dismay")
    print("Horace ate himself one day.")
    print("He didn’t stop to say his grace,")
    print("He just sat down and ate his face.")

def say_goodbye(name):
    print('Goodbye, ', name, '!', sep='')
```

```
def main():
    your_name = input('What is your name? ')
    insert_separator("-", 20)
    say_hello(your_name)
    insert_separator()
    recite_poem()
    insert_separator()
    say_goodbye(your_name)

main()
```

4.5. Combining Concatenation and Repetition

Concatenation and repetition can be combined. Repetition takes precedence, meaning it occurs first. Consider the following:

```
>>> 'a' + 'b' * 3 + 'c'
'abbbc'
```

Notice the output is “abbbc”. In other words, “b” will be repeated three times before it is concatenated with “a” and “c”.

We can force the concatenation to take place first by using parentheses:

```
>>> ('a' + 'b') * 3 + 'c'
'abababc'
```

Notice the output is “abababc”. In other words, “a” will be concatenated with “b”, then “ab” will be repeated three times, and finally “ababab” will be concatenated with “c”.

The following demo shows an example of combining concatenation with repetition:

Demo 4.7: strings/Demos/combining_concatenation_and_repetition.py

```
flower = input("What is your favorite flower? ")
reply = "A " + flower + (" is a " + flower) * 2 + "."
print(reply)
```

The preceding code will render the following:

```
What is your favorite flower? dandelion
A dandelion is a dandelion is a dandelion.
```

4.6. Python Strings are Immutable

Python strings are immutable, meaning that they cannot be changed. However, it is easy to make a copy of a string and then assign the copy to the same variable as the original string.

To illustrate, we will use Python’s built-in `id()` function, which returns the identity of an object:

```
>>> name = 'Nat'
>>> id(name)
1670060967728
>>> name += 'haniel'
>>> name
'Nathaniel'
>>> id(name)
1670060968240
```

Notice that the id of `name` changes when we modify the string in the variable.

Because strings are immutable, methods that operate on strings (i.e., string methods) cannot modify the string in place. Instead, they return a value. Sometimes that value is a modified version of the string, but it is important to understand that the original string is unchanged. Consider, for example, the `upper()` method, which returns a string in all uppercase letters:

```
>>> name = 'Nat'
>>> name.upper() # Returns uppercase copy of 'Nat'
'NAT'
```

```
>>> name # Original variable is unchanged
'Nat'
```

If you want to change the original variable, you must assign the returned value back to the variable:

```
>>> name = 'Nat'
>>> name = name.upper()
>>> name
'NAT'
```

4.7. Common String Methods

4.7.1. String Methods that Return a Copy of the String

Methods that Change Case

- `str.capitalize()` – Returns a string with only the first letter capitalized.
- `str.lower()` – Returns an all lowercase string.
- `str.upper()` – Returns an all uppercase string.
- `str.title()` – Returns a string with each word beginning with a capital letter followed by all lowercase letters.
- `str.swapcase()` – Returns a string with the case of each letter swapped.
- `str.replace(old, new[, count])` – A string with `old` replaced by `new` `count` times.

```
>>> 'hELLo'.capitalize()
'Hello'
>>> 'hELLo'.lower()
'hello'
>>> 'hELLo'.upper()
'HELLO'
>>> 'hello world'.title()
'Hello World'
>>> 'hELLo'.swapcase()
'HellO'
>>> 'mommy'.replace('m', 'b')
'bobby'
>>> 'mommy'.replace('m', 'b', 2)
'bobmy'
```

> **Square Brackets in Code Notation**
>
> As we mentioned in the **Math lesson**, square brackets in code notation indicate that the contained portion is optional. To illustrate, consider the `str.replace()` method:
>
> ```
> str.replace(old, new[, count])
> ```
>
> This means that the `count` parameter is optional. The syntax allows for nesting optional parameters within optional parameters. For example:
>
> ```
> str.find(sub[, start[, end]])
> ```
>
> This indicates that `end` cannot be specified unless `start` is also specified. The outside brackets in `[, start[, end]]` indicate that the whole section is optional. The inside brackets indicate that `end` is optional even if `start` is specified. If it were written as `[start, end]`, it would indicate that `start` and `end` are optional, but if one is included, the other must also be included.

Methods that Strip Characters

- `str.strip([chars])` – Returns a string with leading and trailing `chars` removed.
- `str.lstrip([chars])` – Returns a string with leading `chars` removed.
- `str.rstrip([chars])` – Returns a string with trailing `chars` removed.

`chars` defaults to whitespace.

```
>>> '  hello  '.strip()
'hello'
>>> 'hello'.lstrip('h')
'ello'
>>> 'hello'.rstrip('o')
'hell'
```

4.7.2. String Methods that Return a Boolean

- `str.isalnum()` – Returns `True` if all characters are letters or numbers.
- `str.isalpha()` – Returns `True` if all characters are letters.
- `str.islower()` – Returns `True` if string is all lowercase.
- `str.isupper()` – Returns `True` if string is all uppercase.
- `str.istitle()` – Returns `True` if string is title case.

```
>>> 'Hello World!'.isalnum()
False
>>> 'Hello World!'.isalpha()
False
>>> 'hello'.islower()
True
>>> 'HELLO'.isupper()
True
>>> 'Hello World!'.istitle()
True
```

`str.isspace()`

`True` if string is made up of only whitespace.

```
>>> '   '.isspace()
True
>>> '  hi  '.isspace()
False
```

`str.isdigit()`, `str.isdecimal()`, and `str.isnumeric()`

The `str.isdigit()`, `str.isdecimal()`, and `str.isnumeric()` all check to see if a string has only numeric characters.

1. All three will return `True` if the string contains only Arabic digits (i.e., `0` through `9`):

   ```
   '42'.isdigit() # True
   '42'.isdecimal() # True
   '42'.isnumeric() # True
   ```

2. All three will return `False` if any character is non-numeric:

   ```
   '4.2'.isdigit() # False
   '4.2'.isdecimal() # False
   '4.2'.isnumeric() # False
   ```

Beyond that, the difference is mostly academic for most Python developers:

1. `'2³'.isnumeric()` and `'2³'.isdigit()` return `True`, but `'2³'.isdecimal()` returns `False`.
2. `'¼'.isnumeric()` returns `True`, but `'¼'.isdigit()` and `'¼'.isdecimal()` return `False`.

So, which should you use? It doesn't make much difference really, but `isdigit()` is the most popular, perhaps because the name most closely matches the intention of the function.

If you're really curious about it, run strings/Demos/numbers.py, which will create numbers.txt and numbers.html files in the same folder showing tabular results, like this (but with much more

data):

Char	isdigit	isdecimal	isnumeric
0	TRUE	TRUE	TRUE
1	TRUE	TRUE	TRUE
2	TRUE	TRUE	TRUE
3	TRUE	TRUE	TRUE
²	TRUE	FALSE	TRUE
¼	FALSE	FALSE	TRUE

`str.startswith()` and `str.endswith()`

- `str.startswith(prefix[, start[, end]])` – Returns `True` if `string` starts with `prefix`.
- `str.endswith(suffix[, start[, end]])` – Returns `True` if `string` ends with `suffix`.

Both of these methods start looking at `start` index and end looking at `end` index if `start` and `end` are specified.

```
>>> 'hello'.startswith('h')
True
>>> 'hello'.endswith('o')
True
```

> **is… Methods**
>
> All the preceding methods that begin with `is…` return `False` for *empty* strings (strings with zero length).

4.7.3. String Methods that Return a Number

String Methods that Return a Position (Index) of a Substring

- `str.find(sub[, start[, end]])` – Returns the lowest index where `sub` is found. Returns `-1` if sub isn't found.
- `str.rfind(sub[, start[, end]])` – Returns the highest index where `sub` is found. Returns `-1` if sub isn't found.
- `str.index(sub[, start[, end]])` – Same as `find()`, but errors when `sub` is not found.
- `str.rindex(sub[, start[, end]])` – Same as `rfind()`, but errors when `sub` is not found.

All of these methods start looking at `start` index and end looking at `end` index if `start` and `end` are specified.

```
>>> 'Hello World!'.find('l')
2
>>> 'Hello World!'.rfind('l')
9
>>> 'Hello World!'.index('l')
2
>>> 'Hello World!'.rindex('l')
9
```

str.count(sub[, start[, end]])

Returns the number of times `sub` is found. Start looking at `start` index and end looking at `end` index.

```
>>>"Hello World!".count('l')
3
```

4.8. String Formatting

Python includes powerful options for formatting strings.

4.8.1. The `format()` Method

One common way to format strings is to use the `format()` method combined with the Format Specification Mini-Language[21].

Let's start with a simple example and then we'll explain the mini-language in detail:

```
>>> '{0} is an {1} movie!'.format('Monty Python','awesome')
'Monty Python is an awesome movie!'
```

The curly braces are used to indicate a replacement field, which takes position arguments specified by index (as in the preceding example) or by name (as in the following example):

```
>>> '{movie} is an {adjective} movie!'.format(movie='Monty Python',
...                                           adjective='awesome')
'Monty Python is an awesome movie!'
```

The field names (position arguments) can be omitted:

```
'{} is an {} movie!'.format('Monty Python', 'awesome')
```

When the field names are omitted, the first replacement field is at index `0`, the second at index `1`, and so on.

These examples really just show another form of concatenation and could be rewritten like this:

```
'Monty Python' + ' is an ' + 'awesome' + ' movie!'
```

Or:

```
movie = 'Monty Python'
adjective = 'awesome'
movie + ' is an ' + adjective + ' movie!'
```

When doing a lot of concatenation, using the `format()` method can be cleaner. However, as the name implies, the `format()` method does more than just concatenation; it also can be used to *format* the replacement strings. It is mostly used for formatting numbers.

4.8.2. Format Specification

The format specification is separated from the field name or position by a a colon (:), like this:

```
{field_name:format_spec}
```

Because the field name is often left out, it is commonly written like this:

```
{:format_spec}
```

The format specification[22] is:

```
[[fill]align][sign][width][,][.precision][type]
```

That looks a little scary, so let's break it down from right to left.

Type

```
[[fill]align][sign][width][,][.precision][type]
```

Type is specified by a one-letter specifier, like this:

```
'{:s} is an {:s} movie!'.format('Monty Python','awesome')
```

The `s` indicates that the replacement field should be formatted as a string. There are many different types, but, unless you are a mathematician or scientist[23], the most common types you'll be working with are strings, integers, and floats.

The default formatting for strings and integers are string format (`s`) and decimal integer (`d`), which are generally what you will want, so you can leave the one-letter type specifier off. Consider the following:

Demo 4.8: strings/Demos/formatting_types.py

```
# Full formatting strings.
sentence = 'On a scale of {0:d} to {1:d}, I give {2:s} a {3:d}.'
sentence = sentence.format(1, 5, 'Monty Python', 6)
print(sentence)

# Simplify by removing field names (indexes).
sentence = 'On a scale of {:d} to {:d}, I give {:s} a {:d}.'
sentence = sentence.format(1, 5, 'Monty Python', 6)
print(sentence)

# Further simplify by removing default type specifiers.
sentence = 'On a scale of {:} to {:}, I give {:} a {:}.'
sentence = sentence.format(1, 5, 'Monty Python', 6)
print(sentence)

# And with the field name and type specifier gone, we can
# remove the colon separator.
sentence = 'On a scale of {} to {}, I give {} a {}.'
sentence = sentence.format(1, 5, 'Monty Python', 6)
print(sentence)
```

The final line of code has the advantage of being brief, but the disadvantage of being obscure. As a rule, we prefer clarity over brevity. We can make it clearer using field names:

```
>>> 'On a scale of {low} to {high}, {movie} is a {rating}.'.format(low=1, high=5, movie='Monty Python',
'On a scale of 1 to 5, Monty Python is a 6.'
```

Floats

You will typically format floats as fixed point numbers using `f` as the one-letter specifier, which has a default precision of 6. If neither type nor precision is specified, floats will be as precise as they need to be to accurately represent the value.

Fixed point type specified:

```
>>> import math
>>> 'pi equals {:f}'.format(math.pi)
'pi equals 3.141593'
```

No type specified:

```
>>> import math
>>> 'pi equals {}'.format(math.pi)
'pi equals 3.141592653589793'
```

Another formatting option for floats is percentage (`%`). We will cover that shortly.

Precision

```
[[fill]align][sign][width][,][.precision][type]
```

The precision is specified before the type and is preceded by a decimal point, like this:

```
>>> import math
>>> 'pi equals {:.2f}'.format(math.pi)
'pi equals 3.14'
```

Separating the Thousands

```
[[fill]align][sign][width][,][.precision][type]
```

Insert a comma before the precision value to separate the thousands with commas, like this:

```
>>> '{:,.2f}'.format(1000000)
'1,000,000.00'
```

Width

```
[[fill]align][sign][width][,][.precision][type]
```

The width is an integer indicating the minimum number of characters of the resulting string. If the passed-in string has fewer characters than the specified width, *padding* will be added. By default, padding is added *after* strings and *before* numbers, so that strings are aligned to the left and numbers are aligned to the right.

Here are some examples:

```
>>> '{:5}'.format('abc')
'abc  '
>>> '{:5}'.format(123)
'  123'
>>> '{:5.2f}'.format(123)
'123.00'
```

In all three cases, the width of the formatted string is set to `5`. Notice the padding on the first two examples: after the string and before the number.

In the final example, we format the number `123`, but the format type has been specified as fixed point (`f`) with a precision of `2`. So, the resulting string (`'123.00'`) is six characters long – longer than the specified width, so it just returns the full string without padding.

Sign

```
[[fill]align][sign][width][,][.precision][type]
```

By default, negative numbers are preceded by a negative sign, but positive numbers are not preceded by a positive sign. To force the sign to show up, add a plus sign (`+`) before the precision, like this:

```
>>> 'pi equals {:+.2f}'.format(math.pi)
'pi equals +3.14'
```

Alignment

```
[[fill]align][sign][width][,][.precision][type]
```

You can change the default alignment by preceding the width (and sign if there is one) with one of the following options:

Alignment

Options	Meaning
<	Left aligned (default for strings).
>	Right aligned (default for numbers).
=	Padding added between sign and digits. Only valid for numbers.
^	Centered.

Some examples:

```
>>> '{:>5}'.format('abc')
'  abc'

>>> '{:<5}'.format(123)
'123  '

>>> '{:^5}'.format(123)
' 123 '

>>> '{:=+5}'.format(123)
'+ 123'
```

Fill

```
[[fill]align][sign][width][,][.precision][type]
```

By default, spaces are used for padding, but this can be changed by inserting a fill character before the alignment option, like this (note the period after the colon):

```
>>> '{:.^10.2f}'.format(math.pi)
'...3.14...'
```

And now with a dash:

```
>>> '{:-^10.2f}'.format(math.pi)
'---3.14---'
```

Percentage Type

As mentioned earlier, another option for type is percentage (%):

```
>>> questions = 25
>>> correct = 18
>>> grade = correct / questions
>>> grade
0.72
>>> '{:.2f}'.format(grade)
'0.72'
>>> '{:.2%}'.format(grade)
'72.00%'
>>> '{:.0%}'.format(grade)
'72%'
```

4.8.3. Long Lines of Code

The Python Style Guide[24] suggests that lines of code should be limited to 79 characters. This can be difficult as each line of code is considered a new statement; however, it can usually be accomplished through some combination of:

1. Breaking method arguments across multiple lines.
2. Concatenation.
3. Triple-quoted multi-line strings.

All three methods are shown in the following file:

Demo 4.9: strings/Demos/long_code_lines.py

```
# EXAMPLE 1: Breaking method arguments across multiple lines
phrase = ("On a scale of {} to {}, I give {} a {}.".format(1, 5,
                                                           "Monty Python", 6))
print(phrase)

location = "ponds"
items = "swords"
beings = "masses"
adjective = "farcical"
```

```
# EXAMPLE 2: Concatenation
quote = ("Listen, strange women lyin' in {} " +
         "distributin' {} is no basis for a system of " +
         "government. Supreme executive power derives from " +
         "a mandate from the {}, not from some {} " +
         "aquatic ceremony.").format(location, items,
                                     beings, adjective)

print(quote)

# EXAMPLE 3: Triple quotes
quote = """Listen, strange women lyin' in {} \
distributin' {} is no basis for a system of \
government. Supreme executive power derives from \
a mandate from the {}, not from some {} \
aquatic ceremony.""".format(location, items,
                            beings, adjective)

print(quote)
```

Remember that the backward slashes at the end of each line escape the newline character.

Also, notice that the arguments passed to `format()` are broken across lines and vertically aligned to make it clear that they are related.

Run the file to see the resulting strings.

Exercise 14: Playing with Formatting

10 to 20 minutes.

In this exercise, you will practice formatting strings. Here are two options for practicing:

Option 1: Run `'{}'.format('')` at the Python shell and try formatting different values in different ways. For example, try running:

```
>>> '{:.0%}'.format(.87)
'87%'
```

Option 2: From strings/Demos run `python formatter.py`, which will prompt you for a format to try and an entry to format:

```
Format to try: {:f}
Entry to format: math.pi
Result:  3.141593
Enter for another or 'q' to quit.
```

Try different format specifications with different values:

Format	Value	Expected Result
{:f}	math.pi	3.141593
{}	math.pi	3.141592653589793
{:.2f}	math.pi	3.14
{:.2f}	1000000	1000000.00
{:,.2f}	1000000	1,000,000.00
{:>20}	'abc'	abc

4.9. Formatted String Literals (f-strings)

Formatted string literals or *f-strings*[25] use a syntax that implicitly embeds the `format()` function within the string itself. The coding tends to be less verbose.

The following demonstration compares string concatenation and string formatting with the f-string syntax.

Demo 4.10: strings/Demos/f_strings.py

```
import math
user_name = input("What is your name? ")

# Concatenation:
greeting = "Hello, " + user_name + "!"

# The format() method:
greeting = "Hello, {}!".format(user_name)

# f-string:
greeting = f"Hello, {user_name}!"
print(greeting)

# format specification is also available:
pi_statement = f"pi is {math.pi:.4f}"
print(pi_statement)
```

The curly braces within the f-string contain the variable name and optionally a format specification. The string literal is prepended with an `f`.

Everything you learned earlier about formatting can be applied to the f-string because the same formatting function is called. Practice with f-strings by running the following lines of code:

Basic f-strings

```
>>> import math
>>> movie = 'Monty Python'
>>> adjective = 'awesome'
>>> f'{movie} is an {adjective} movie!'
'Monty Python is an awesome movie!'
>>> low = 1
>>> high = 5
>>> rating = 6
>>> f'On a scale of {low} to {high}, {movie} is a {rating}.'
'On a scale of 1 to 5, Monty Python is a 6.'
```

f-strings with Formatting Specifications

```
>>> f'pi equals {math.pi:f}'
'pi equals 3.141593'
>>> f'pi equals {math.pi}'
'pi equals 3.141592653589793'
>>> f'pi equals {math.pi:.2f}'
'pi equals 3.14'
>>> f'{1000000:,.2f}'
'1,000,000.00'
>>> f"{'abc':20}"
'abc                 '
>>> f'{123:20}'
'                 123'
>>> f'{123:5.2f}'
'123.00'
>>> f'pi equals {math.pi:+.2f}'
'pi equals +3.14'
>>> f"{'abc':>20}"
'                 abc'
>>> f'{123:<20}'
'123                 '
>>> f'{123:=+20}'
'+                123'
>>> f'pi equals {math.pi:.^10.2f}'
'pi equals ...3.14...'
>>> f'pi equals {math.pi:-^10.2f}'
'pi equals ---3.14---'
>>>
>>> questions = 25
```

```
>>> correct = 18
>>> grade = correct / questions
>>> f'{grade:.2f}'
'0.72'
>>> f'{grade:.2%}'
'72.00%'
>>> f'{grade:.2f}'
'0.72'
```

4.10. Built-in String Functions

`str(object)`

Converts object to a string.

```
>>> str(42)
'42'
```

len(string)

Returns the number of characters in the string.[26]

```
>>> len('foo')
3
```

`min()` and `max()`

`min(args)` returns the smallest value and `max(args)` returns the largest value of the passed-in arguments.

```
>>> min('w', 'e', 'b')
'b'
>>> min('a', 'B', 'c')
'B'
>>> max('w', 'e', 'b')
'w'
>>> max('a', 'B', 'c')
'c'
```

Note that all uppercase letters come before lowercase letters (e.g., `min('Z', 'a')` returns 'Z').

> min() and max() with Numbers and Iterables
>
> The `min()` and `max()` functions also work with ***numbers*** and ***iterables***.

Exercise 15: Outputting Tab-delimited Text

25 to 40 minutes.

In this exercise, you will write a program that repeatedly prompts the user for a Company name, Revenue, and Expenses and then outputs all the information as tab-delimited text. Here is the program after it has run:

```
Company: Pepperpots
Revenue: 1200000
Expenses: 999002
Again? Press ENTER to add a row or Q to quit.
Company: Ni Knights
Revenue: 19
Expenses: 24
Again? Press ENTER to add a row or Q to quit.
Company: Round Knights
Revenue: 777383
Expenses: 777382
Again? Press ENTER to add a row or Q to quit. q
 Company          Revenue          Expenses          Profit
Pepperpots      $1,200,000.00   $999,002.00     $200,998.00
Ni Knights      $     19.00     $     24.00     $     -5.00
Round Knights   $777,383.00     $777,382.00     $      1.00
```

In this exercise, you can use the `format()` method or f-strings or any combination of the two.

1. Open strings/Exercises/tab_delimited_text.py.
2. Modify the `add_headers()` function so that it creates a header row and appends it to `_output`. The four headers should each take up 10 spaces, be aligned to the center, and be separated by tabs, like this:

   ```
   '  Company  \t Revenue  \t Expenses \t  Profit  \n'
   ```

   Don't just copy that string. Use the `format()` method.
3. Modify the `add_row()` function so that it adds a row to `_output` by prompting the user for values for company, revenue, and expenses, and then calculating profit.
   A. All "columns" should be 10 spaces wide.
   B. The company name should be a left-aligned string.
   C. The other three columns should be formatted in U.S. dollars (e.g., `$1,200,000.00`) and right-aligned.
4. Save and run the file. Try entering data for at least three companies.

Exercise Code 15.1: strings/Exercises/tab_delimited_text.py

```
_output = ""

def add_headers():
    # Write your code here
    pass

def add_row():
    # Write your code here

    # The rest of the function prompts the user to add another row
    # or quit. On quitting, it prints _output. Leave it as is.

    again = input("Again? Press ENTER to add a row or Q to quit. ")
    if again.lower() != "q":
        add_row()
    else:
        print(_output)

def main():
    # Call add_headers() and add_row()
    add_headers()
    add_row()

main()
```

Solution: strings/Solutions/tab_delimited_text.py

```
_output = ""

def add_headers():
    global _output
    c_header = "{:^10}".format("Company")
    r_header = "{:^10}".format("Revenue")
    e_header = "{:^10}".format("Expenses")
    p_header = "{:^10}".format("Profit")
    _output += "{}\t{}\t{}\t{}\n".format(c_header, r_header,
                                        e_header, p_header)

def add_row():
    global _output

    c = input("Company: ")
    r = float(input("Revenue: "))
    e = float(input("Expenses: "))
    p = r - e # profit

    c_str = "{:<10}".format(c)
    r_str = "${:>10,.2f}".format(r)
    e_str = "${:>10,.2f}".format(e)
    p_str = "${:>10,.2f}".format(p)

    new_row = "{}\t{}\t{}\t{}\n".format(c_str, r_str, e_str, p_str)

    _output += new_row
-------Lines Omitted-------
```

Solution: strings/Solutions/tab_delimited_text_f_string.py

```
_output = ""

def add_headers():
    global _output
    c_header = f"{'Company':^10}"
    r_header = f"{'Revenue':^10}"
    e_header = f"{'Expenses':^10}"
    p_header = f"{'Profit':^10}"
    _output += f"{c_header}\t{r_header}\t{e_header}\t{p_header}\n"

def add_row():
    global _output

    c = input("Company: ")
    r = float(input("Revenue: "))
    e = float(input("Expenses: "))
    p = r - e # profit

    c_str = f"{c:<10}"
    r_str = f"${r:>10,.2f}"
    e_str = f"${e:>10,.2f}"
    p_str = f"${p:>10,.2f}"

    new_row = f"{c_str}\t{r_str}\t{e_str}\t{p_str}\n"

    _output += new_row
-------Lines Omitted-------
```

Conclusion

In this lesson, you have learned to manipulate and format strings.