# learn enough PYTHON TO BE DANGEROUS

SOFTWARE DEVELOPMENT, FLASK WEB APPS, AND BEGINNING DATA SCIENCE WITH PYTHON

MICHAEL HARTL

# Praise for Learn Enough Tutorials

"Just started the #100DaysOfCode journey. Today marks day 1. I have completed @mhartl's great Ruby tutorial at @LearnEnough and am looking forward to starting on Ruby on Rails from tomorrow. Onwards and upwards."

—Optimize Prime (@_optimize), Twitter post

"Ruby and Sinatra and Heroku, oh my! Almost done with this live web application. It may be a simple palindrome app, but it's also simply exciting! #100DaysOfCode #ruby @LearnEnough #ABC #AlwaysBeCoding #sinatra #heroku"

—Tonia Del Priore (@toninjaa), Twitter post; Software Engineer for a FinTech Startup for 3+ years

"I have nothing but fantastic things to say about @LearnEnough courses. I am just about finished with the #javascript course. I must say, the videos are mandatory because @mhartl will play the novice and share in the joy of having something you wrote actually work!"

—Claudia Vizena

"I must say, this Learn Enough series is a masterpiece of education. Thank you for this incredible work!"

—Michael King

"I want to thank you for the amazing job you have done with the tutorials. They are likely the best tutorials I have ever read."

—Pedro Iatzky

This leads to the elegant iteration shown in Listing 4.7.

**Listing 4.7:** Iterating through a dictionary's `items()`.

```
>>> for name, year in moonwalks.items():   # Pythonic
...     print(f"{name} first performed a moonwalk in {year}")
...
Neil Armstrong first performed a moonwalk in 1969
Buzz Aldrin first performed a moonwalk in 1969
Alan Shepard first performed a moonwalk in 1971
Eugene Cernan first performed a moonwalk in 1972
Michael Jackson first performed a moonwalk in 1983
```

Note that we've also changed to meaningful names in Listing 4.7, using `name, year` in preference to the less specific `key, value`.

### 4.4.2 Merging Dictionaries

One common operation is *merging* dictionaries, where the elements of two dictionaries are combined into one. For example, consider two dictionaries consisting of academic subjects with corresponding test scores:

```
>>> tests1 = {"Math": 75, "Physics": 99}
>>> tests2 = {"History": 77, "English": 93}
```

It would be nice to be able to create a `tests` dictionary combining all four subject–score combinations.

Older versions of Python didn't natively support merging dictionaries at all, but Python 3.5 added this `**` syntax:

```
>>> {**tests1, **tests2}    # Kind of Pythonic
{'Math': 75, 'Physics': 99, 'History': 77, 'English': 93}
```

That's pretty strange-looking syntax if you ask me, and it's included here mainly because you might encounter it in other people's code. Luckily, as of Python 3.9 there's a great way to merge dictionaries using the pipe operator `|`:

```
>>> tests1 | tests2         # Very Pythonic
{'Math': 75, 'Physics': 99, 'History': 77, 'English': 93}
```

When the dictionaries have no overlapping keys, merging them simply involves combining all key–value pairs. But if the second dictionary does have one or more keys in common, then its values take precedence. In this case, we can think of *updating* the first dictionary with the contents of the second.[7] For example, suppose we combine the tests into a single variable using a merge:

```
>>> test_scores = tests1 | tests2
{'Math': 75, 'Physics': 99, 'History': 77, 'English': 93}
```

Now suppose the student is allowed to retake tests for the two lowest scores:

```
>>> retests = {"Math": 97, "History": 94}
```

At this point, we can update the original test scores with the updated values from the retests (Listing 4.8).

**Listing 4.8:** Updating a dictionary using a merge.

```
>>> test_scores | retests
{'Math': 97, 'Physics': 99, 'History': 94, 'English': 93}
```

We see that the **`"Math"`** and **`"History"`** scores have been updated with the values from the second dictionary.

### 4.4.3 Exercises

1. Define a dictionary for a **`user`** with three attributes (keys): **`"username"`**, **`"password"`**, and **`"password_confirmation"`**. How would you test if the password matches the confirmation?
2. We've seen in Listing 2.29 and Listing 3.10 that Python strings and lists support an **`enumerate()`** function in cases where we need the iteration index. Confirm that we can do the same thing with dictionaries using code like Listing 4.9.

7. For this reason, dictionary (or, rather, hash) merges in Ruby use the **`update`** method.

3. By reversing the elements in Listing 4.8, show that dictionary merges aren't symmetric, so `d1 | d2` is not in general the same as `d2 | d1`. When are they the same?

**Listing 4.9:** Using `enumerate()` with a dictionary.

```
>>> for i, (name, year) in enumerate(moonwalks.items()):   # Pythonic
...     print(f"{i+1}. {name} first performed a moonwalk in {year}")
...
1. Neil Armstrong first performed a moonwalk in 1969
2. Buzz Aldrin first performed a moonwalk in 1969
3. Alan Shepard first performed a moonwalk in 1971
4. Eugene Cernan first performed a moonwalk in 1972
5. Michael Jackson first performed a moonwalk in 1983
```

## 4.5 Application: Unique Words

Let's apply the dictionaries from Section 4.4 to a challenging exercise, consisting of our longest program so far. Our task is to extract all the unique words in a fairly long piece of text, and count how many times each word appears.

Because the sequence of commands is rather extensive, our main tool will be a Python file (Section 1.3), executed using the `python3` command. (We're not going to make it a self-contained shell script as in Section 1.4 because we don't intend for this to be a general-purpose utility program.) At each stage, I suggest using Python to execute the code interactively if you have any question about the effects of a given command.

Let's start by creating our file:

```
(venv) $ touch count.py
```

Now fill it with a string containing the text, which we'll choose to be Shakespeare's Sonnet 116[8] (Figure 4.9[9]), as borrowed from Listing 4.6 and shown again in Listing 4.10.

8. Note that in the original pronunciation used in Shakespeare's time, words like "love" and "remove" rhymed, as did "come" and "doom".

9. Image courtesy of Psychoshadowmaker/123RF.

**Figure 4.9:** Sonnet 116 compares love's constancy to the guide star for a wandering bark (ship).

**Listing 4.10:** Adding some text.

*count.py*

```
import re


sonnet = """Let me not to the marriage of true minds
Admit impediments. Love is not love
Which alters when it alteration finds,
Or bends with the remover to remove.
O no, it is an ever-fixed mark
That looks on tempests and is never shaken
It is the star to every wand'ring bark,
Whose worth's unknown, although his height be taken.
Love's not time's fool, though rosy lips and cheeks
Within his bending sickle's compass come:
Love alters not with his brief hours and weeks,
But bears it out even to the edge of doom.
   If this be error and upon me proved,
   I never writ, nor no man ever loved."""
```

Our plan will be to use a dictionary called **uniques** with keys equal to the unique words and values equal to the number of occurrences in the text:

```
uniques = {}
```

For the purposes of this exercise, we'll define a "word" as a run of one or more *word characters* (i.e., letters or numbers, though there are none of the latter in the present text). This match can be accomplished with a regular expression (Section 4.3), which includes a pattern (**\w**) for exactly this case (Figure 4.5):

```
words = re.findall(r"\w+", sonnet)
```

This uses the **findall()** method from Section 4.3 to return a list of all the strings that match "one or more word characters in a row". (Extending this pattern to include apostrophes (so that it matches, e.g., "wand'ring" as well) is left as an exercise (Section 4.5.1).)

At this point, the file should look like Listing 4.11.

**Listing 4.11:** Adding an object and the matching words.
*count.py*

```
import re


sonnet = """Let me not to the marriage of true minds
Admit impediments. Love is not love
Which alters when it alteration finds,
Or bends with the remover to remove.
O no, it is an ever-fixed mark
That looks on tempests and is never shaken
It is the star to every wand'ring bark,
Whose worth's unknown, although his height be taken.
Love's not time's fool, though rosy lips and cheeks
Within his bending sickle's compass come:
Love alters not with his brief hours and weeks,
But bears it out even to the edge of doom.
   If this be error and upon me proved,
   I never writ, nor no man ever loved."""

uniques = {}
words = re.findall(r"\w+", sonnet)
```

Now for the heart of our program. We're going to iterate through the **words** list and do the following:

1. If the word already has an entry in the **uniques** object, increment its count by **1**.
2. If the word doesn't have an entry yet in **uniques**, initialize it to **1**.

The result, using the **+=** operator we met briefly in Section 4.3, looks like this:

```
for word in words:
    if word in uniques:
        uniques[word] += 1
    else:
        uniques[word] = 1
```

Finally, we'll print out the result to the terminal:

```
print(uniques)
```

The full program (with added comments) appears as in Listing 4.12.

**Listing 4.12:** A program to count words in text.
*count.py*

```
sonnet = """Let me not to the marriage of true minds
Admit impediments. Love is not love
Which alters when it alteration finds,
Or bends with the remover to remove.
O no, it is an ever-fixed mark
That looks on tempests and is never shaken
It is the star to every wand'ring bark,
Whose worth's unknown, although his height be taken.
Love's not time's fool, though rosy lips and cheeks
Within his bending sickle's compass come:
Love alters not with his brief hours and weeks,
But bears it out even to the edge of doom.
   If this be error and upon me proved,
   I never writ, nor no man ever loved."""

# Unique words
uniques = {}
# All words in the text
words = re.findall(r"\w+", sonnet)

# Iterate through `words` and build up a dictionary of unique words.
```