Using Dictionaries

*The **dict** object in Python is a very powerful data tool. Unlike a list, which contains an ordered collection of items accessible by indexing, a dictionary works more like its namesake: the contents are accessed via specific look up 'keys'. We'll see how useful this can be for a range of tasks, including writing a quiz, keeping track of data as you iterate through some process, etc.*

The basics

You can think of a list as being a convenient way of storing data by a predefined index:
Consider the list below:

```
["Alpha", "Bravo", "Charlie", "Delta"]
```

It is equivalent to:

Index	0	1	2	3
Element	"Alpha"	"Bravo"	"Charlie"	"Delta"

Now, if we instead use a dictionary, we can effectively define our own 'index' (or 'key'):
Consider the dictionary below:

```
{"A": "Alpha", "B": "Bravo", "C": "Charlie", "D": "Delta}
```

This is equivalent to:

Key	"A"	"B"	"C"	"D"
Value	"Alpha"	"Bravo"	"Charlie"	"Delta"

You may already be able to see how useful this could be. If there is a natural 'key' to use, it is more intuitive and therefore less prone to user/programmer error.

The values can be 'extracted' in much the same way as a list:

```
my_list = ["Alpha", "Bravo", "Charlie", "Delta"]
my_list[2] returns "Charlie"
```

```
my_dict = {"A": "Alpha", "B": "Bravo", "C": "Charlie", "D": "Delta"}
my_dict["C"] returns "Charlie"
```

Note: Dictionaries use curly braces, so an empty one is `d = {}`, rather than `l = []`.

How to loop through a dictionary

One important difference between dictionaries and lists is that dictionaries are fundamentally unordered. That is, the values are stored according to their key, but the keys are in no particular order. You can loop through the keys (or the keys and values) in a dictionary, but in older versions of python, you might even find that the order they loop in may change from one loop to the next. Not usually a problem. See examples below:

```
scrabble = {"A": 9, "E": 12, "I": 9, "O": 8, "U": 4}

for vowel in scrabble:
  print(f"There are {scrabble[vowel]} of the letter {vowel}")
```

The loop iterates through keys of the dictionary, using **scrabble[vowel]** to get the value.

```
for vowel, frequency in scrabble.items():
  print(f"There are {frequency} of the letter {vowel}")
```

This loop uses the dictionary method **items()** so you can work with each key/value pair.

Dictionary comprehensions

Recall list comprehensions:

We can make the process of populating a *list* quicker using 'list comprehensions' such as:

Using a loop to append to the list:	Using list comprehensions to do the same:

`numbers = []`
`for i in range(10):`
`  numbers.append(i**2)`
	`numbers = [i**2 for i in range(10)]`

Both of these code snippets do the same job: they generate a list that contains all the square numbers from 0^2 to 9^2. There's no real difference 'under the hood', but any code which improves readability and conciseness is generally preferable.

Dictionary comprehensions work in much the same way, and avoid lots of messing around:

Using a loop to add to the dictionary:	Using dictionary comprehensions to do the same:

`grades = {}`
`for grade in ["Dist", "Pass", "Merit", "Fail"]:`
`  grades[grade] = 0`
	`grades = {grade: 0 for grade in ["Dist", "Pass", "Merit", "Fail"]}`

Mini Task 1.

Find the number of factors of each integer below 1000, then print out all the numbers with more than 20 factors, along with the number of factors they have.

Hints (and a possible solution on the next page):

- *Set up an empty dictionary at the start, using curly braces. You could use dictionary comprehensions to pre-populate it with 0 factors for each of your numbers.*
- *Loop through your dictionary, and for each value, test all possible factors.*
- *Increment the value stored in the dictionary for your number if you find a factor.*
- *At the end, loop through the **items** and print if there are enough factors.*

Mini Task 2.

Find the total number of each letter in the text of the following famous sonnet:

Shall I compare thee to a summer's day?
Thou art more lovely and more temperate:
Rough winds do shake the darling buds of May,
And summer's lease hath all too short a date:
Sometime too hot the eye of heaven shines,
And often is his gold complexion dimm'd;
And every fair from fair sometime declines,
By chance or nature's changing course untrimm'd;
But thy eternal summer shall not fade
Nor lose possession of that fair thou owest;
Nor shall Death brag thou wander'st in his shade,
When in eternal lines to time thou growest:
So long as men can breathe or eyes can see,
So long lives this and this gives life to thee.

Hints (and a possible solution on the next page):

- *Set up a dictionary with each letter as a key, initially with 0 as its frequency.*
- *Copy the text into python directly (taking care to enclose the whole thing in speech-marks), or by loading from a text document if you're feeling fancy.*
- *Loop through each character, check if it's in the alphabet, and if so, add to dict.*
- *Use **.items()** to tidily print out all the information.*

Mini Task 1 Possible Solution

```
results = {i: 0 for i in range(1, 1000)}
for n in results:
    for i in range(1, n + 1):
        if n % i == 0:
            results[n] += 1

for n, num_factors in results.items():
    if num_factors > 20:
        print(f"{n} has {num_factors} factors.")
```

Dictionary comprehensions help at the start, then for each key in the dictionary, we loop through all possible factors, adding 1 to the associated value whenever one is found.
Finally, looping through **results.items()** lets us print the results which are relevant.

Bonus:

Rather than simply keeping a tally of the *number* of factors, we could instead have the 'value' within the dictionary actually be a list, and append the actual factors as we go:

```
results = {i: [] for i in range(1, 1000)}
for n in results:
    for i in range(1, n + 1):
        if n % i == 0:
            results[n].append(i)

for n, factors in results.items():
    if len(factors) > 20:
        print(f"{n} has factors: {factors}.")
```

Mini Task 2 Possible Solution

```
sonnet = ["Shall I compare thee to a summer's day?",
"Thou art more lovely and more temperate:",
"Rough winds do shake the darling buds of May,",
"And summer's lease hath all too short a date:",
"Sometime too hot the eye of heaven shines,",
"And often is his gold complexion dimm'd;",
"And every fair from fair sometime declines,",
"By chance or nature's changing course untrimm'd;",
"But thy eternal summer shall not fade",
"Nor lose possession of that fair thou owest;",
"Nor shall Death brag thou wander'st in his shade,",
"When in eternal lines to time thou growest:",
"So long as men can breathe or eyes can see,",
"So long lives this and this gives life to thee."]

abc = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
results = {char: 0 for char in abc}

for line in sonnet:
    for char in line:
        if char.upper() in abc:
            results[char.upper()] += 1

for char, freq in results.items():
    print(f"{char}: {freq}")
```

The poem itself has here just been pasted directly into the code. A more elegant solution (and more easily applied to longer pieces of text), is to upload a text document, and loop through the lines:

```
with open("sonnet.txt", "r") as in_file:
    lines = in_file.readlines()
    for line in lines:
        for char in line:
            ...
```

Making a Quiz

For the final activity, we are going to make a quiz. The quiz questions could be on any topic, and the answers we will accept as correct will be stored in a list contained within a dictionary, where the questions themselves are the keys. We'll create a fairly small collection, but you can play around with making a more user-friendly input method, perhaps by accepting a text file as input and creating the dictionary directly from that.

```
1   import random, time
2   quiz = {"What is 15 in binary? " : ["1111", "00001111"],
3   "What is cosec(x) the reciprocal of? " : ["sin(x)", "sin", "sine"],
4   "What's the most common letter tile in Scrabble? " : ["e"]}
```

Although the **quiz** object is one single object (a dictionary), provided we only press enter after the commas (which separate each key/value pair), python will allow us to overflow onto multiple lines. This can help readability immensely.

```
5
6   input("When you're ready to start the quiz, press Enter.")
7   start = time.time()
8   number, correct = 0, 0
```

I am using the **time** module here to add an extra feature: the **start** time is saved at the beginning of the quiz so I can calculate the total time the quiz takes to complete.
I'm also tallying the number of questions asked and the number answered correctly.

```
9   for question, answers in quiz.items():
10      number += 1
11      guess = input(f"{number}. {question}")
12      if guess.lower() in answers:
13          correct += 1
14          print(f"   Correct!")
15      else:
16          print(f"   Wrong!  The correct answer was {answers[0]}.")
```

Using **quiz.items()** lets me work with the question and the list of correct answers simultaneously.
Converting the user's input to lowercase makes it easier to match with our list of valid answers.

```
18  end = time.time()
19  duration = round(end - start,1)
20  print(f"You scored {correct} out of {number}. ")
21  print(f"You took {duration} seconds.")
```

At the end we check the time (measured in seconds) to determine the total duration, and provide feedback.

List and dictionary comprehensions can get pretty sophisticated. Check out this alternative quiz where comprehensions avoid the use of two nested loops:

```
quiz = {f"{a} x {b} = ": [f"{a*b}"] for a in range(7, 10) for b in range(a, 10)}
```

The above code produces the same as:

```
quiz = {}
for a in range(7, 10):
    for b in range(a, 10):
        quiz[f"{a} x {b} = "] = [f"{a*b}"]
```

Experiment with your own adaptations.

Can you make a quiz to help a sibling with their home-schooling?