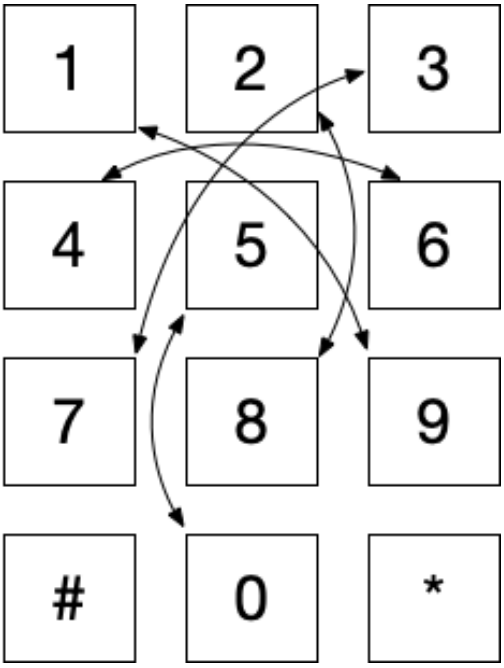
Chapter 5: Jump the Five: Working with dictionaries

> *"When I get up, nothing gets me down." - D. L. Roth*

In an episode of the television show *The Wire*, drug dealers encode telephone numbers that they text in order to obscure them from the police who they assume are intercepting their messages. They use an algorithm we'll call "Jump The Five" where a number is changed to the one that is opposite on a US telephone pad if you jump over the 5. You feel me?



If we start with "1" and jump across the 5, we get to "9," then "6" jumps the 5 to become "4," and so forth. The numbers "5" and "0" will swap with each other. In this exercise, we're going to write a Python program called `jump.py` that will take in some text as a positional argument. Each number in the text will be encoded using this algorithm. All non-number will pass through unchanged, for example:

```
$ ./jump.py 867-5309
243-0751
$ ./jump.py 'Call 1-800-329-8044 today!'
Call 9-255-781-2566 today!
```

We will need some way to inspect each character in the input text and identify the numbers. We will learn how to use a `for` loop for this and how that relates to a "list comprehension." Then we will need some way to associate a number like "1" with the number "9," and so on for all the numbers. We'll learn about a data structure in Python called a "dictionary" type that allows us to do exactly that.

In this chapter, you will learn to:

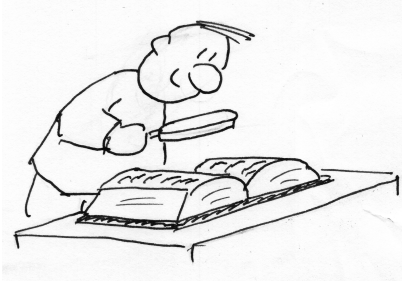
- Create a dictionary.
- Use a `for` loop to process text character-by-character.
- Check if items exist in a dictionary.
- Retrieve values from a dictionary.
- Print a new string with the numbers substituted for their encoded values.

Before we get started with the coding, we need to learn about Python's dictionaries.

Dictionaries

A Python `dict` allows us to relate some *thing* (a "key") to some other *thing* (a "value"). An actual dictionary does this. If we look up a word like "quirky" in a dictionary (https://www.merriam-webster.com/dictionary/quirky), we can find a definition. We can think of the word itself as the "key" and the definition as the "value."

quirky ➪ unusual, esp. in an interesting or appealing way



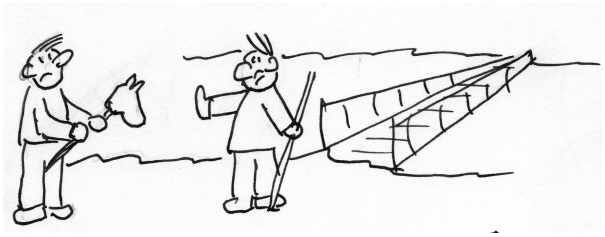
Dictionaries actually provide quite a bit more information such as pronunciation, part of speech, derived words, history, synonyms, alternate spellings, etymology, first known use, etc. (I really love dictionaries.) Each of those attributes has a value, so we could also think of the lookup itself as a dictionary:

quirky

definition	➪ unusual, esp. in an interesting or appealing way
pronunciation	➪ 'kwər-kē
part of speech	➪ adjective

Let's see how we can use Python's dictionaries to go beyond word definitions.

Creating a dictionary



In the film *Monty Python and the Holy Grail*, King Arthur and his knights must cross The Bridge of Death. Anyone who wishes to cross must correctly answer three questions from the Keeper. Those who fail are cast into the Gorge of Eternal Peril.

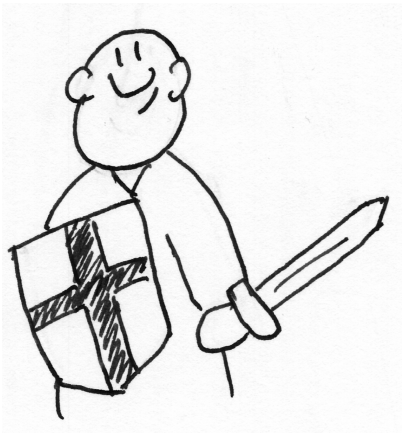
Let us ride to CAMEL…no, sorry, let us create and use a `dict` to keep track of the questions and answers as key/value pairs. Once again, I want you to fire up your `python3` / `ipython` REPL or Jupyter Notebook and type these out for yourself!

Lancelot goes first. We can use the `dict()` function to create an empty dictionary for his answers.

```
>>> answers = dict()
```

Or we can use empty curly brackets:

```
>>> answers = {}
```

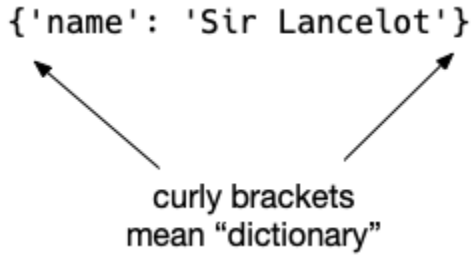


The Keeper's first question: "What is your name?" Lancelot answers "My name is Sir Lancelot of Camelot." We can add the key "name" to our `answers` by using square brackets (`[]` — not curlies!) and the literal string `'name'` :

```
>>> answers['name'] = 'Sir Lancelot'
```

If you type `answers<Enter>` in the REPL, Python will show you a structure in curlies to indicate this is a `dict` :

```
>>> answers
{'name': 'Sir Lancelot'}
```



You can verify with the `type` function:

```
>>> type(answers)
<class 'dict'>
```

Next the Keeper asks, "What is your quest?" to which Lancelot answers "To seek the Holy Grail." Let's add "quest" to the `answers`:

```
>>> answers['quest'] = 'To seek the Holy Grail'
```

There's no return value to let us know something happened, so type `answers` to inspect the variable again to ensure our new key/value was added:

```
>>> answers
{'name': 'Sir Lancelot', 'quest': 'To seek the Holy Grail'}
```

Finally the Keeper asks "What is your favorite color?," and Lancelot answers "blue."

```
>>> answers['favorite_color'] = 'blue'
>>> answers
{'name': 'Sir Lancelot', 'quest': 'To seek the Holy Grail', 'favorite_color': 'blue'}
```

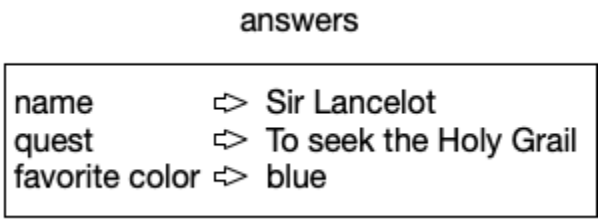
If you knew all the answers beforehand, you could create `answers` using the `dict()` function with this syntax where you do *not* have to quote the keys and the keys are separate from the values with equal signs:

```
>>> answers = dict(name='Sir Lancelot', quest='To seek the Holy Grail', favorite_color='blue')
```

Or this syntax using curlies `{}` where the keys must be quoted and are followed by a colon (`:`):

```
>>> answers = {'name': 'Sir Lancelot', 'quest': 'To seek the Holy Grail', 'favorite_color': 'blue'}
```

It might be helpful to think of the dictionary `answers` as a box that inside holds the key/value pairs that describe Lancelot's answers just the way the "quirky" dictionary holds all the information about that word.



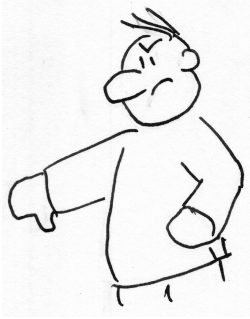
Accessing dictionary values

To retrieve the values, you use the key name inside square brackets (`[]`). For instance, I can get the `name` like so:

```
>>> answers['name']
'Sir Lancelot'
```

Let's request his "age":

```
>>> answers['age']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'age'
```



You will cause an exception if you ask for a dictionary key that doesn't exist!

Just as with lists, you can use the `x in y` to first see if a key exists in the `dict`:

```
>>> 'quest' in answers
True
>>> 'age' in answers
False
```

The `dict.get` method is a *safe* way to ask for a value:

```
>>> answers.get('quest')
'To seek the Holy Grail'
```

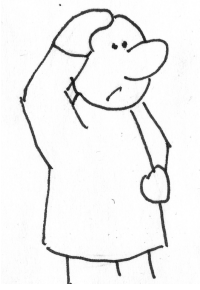
When the requested key does not exist in the `dict`, it will return the special value `None`:

```
>>> answers.get('age')
```

That doesn't print anything because the REPL won't print a `None`, but we can check the `type`:

```
>>> type(answers.get('age'))
<class 'NoneType'>
```

There is an optional second argument you can pass to `dict.get` which is the value to return *if the key does not exist*:



```
>>> answers.get('age', 'NA')
'NA'
```

Other dictionary methods

If you want to know how "big" a dictionary is, the `len` (length) function on a `dict` will tell you how many key/value pairs are present:

```
>>> len(answers)
3
```

The `dict.keys` method will give you just the keys:

```
>>> answers.keys()
dict_keys(['name', 'quest', 'favorite_color'])
```

And `dict.values` will give you the values:

```
>>> answers.values()
dict_values(['Sir Lancelot', 'To seek the Holy Grail', 'blue'])
```

Often we want both together, so you might see code like this:

```
>>> for key in answers.keys():
...     print(key, answers[key])
...
name Sir Lancelot
quest To seek the Holy Grail
favorite_color blue
```

An easier way to write this would be to use the `dict.items` method which will return a `list` of the key/value pairs:

```
>>> answers.items()
dict_items([('name', 'Sir Lancelot'), ('quest', 'To seek the Holy Grail'),
('favorite_color', 'blue')])
```

The above `for` loop could also be written using the `dict.items` method:

```
>>> for key, value in answers.items():  1
...     print(f'{key:15} {value}')      2
...
name            Sir Lancelot
quest           To seek the Holy Grail
favorite_color  blue
```

1. For each key/value pair, unpack them into the variables `key` and `value`. Note that you don't have to cal them `key` and `value`. You could use `k` and `v` or `question` and `answer`.
2. Print the `key` in a left-justified field 15 characters wide. The `value` is printed normally.

In the REPL you can execute `help(dict)` to see all the methods available to you like `pop` to remove a key/value or `update` to merge with another `dict`.

Each key in the `dict` is unique. That means if you set a value for a given key twice:

```
>>> answers = {}
>>> answers['favorite_color'] = 'blue'
>>> answers
{'favorite_color': 'blue'}
```

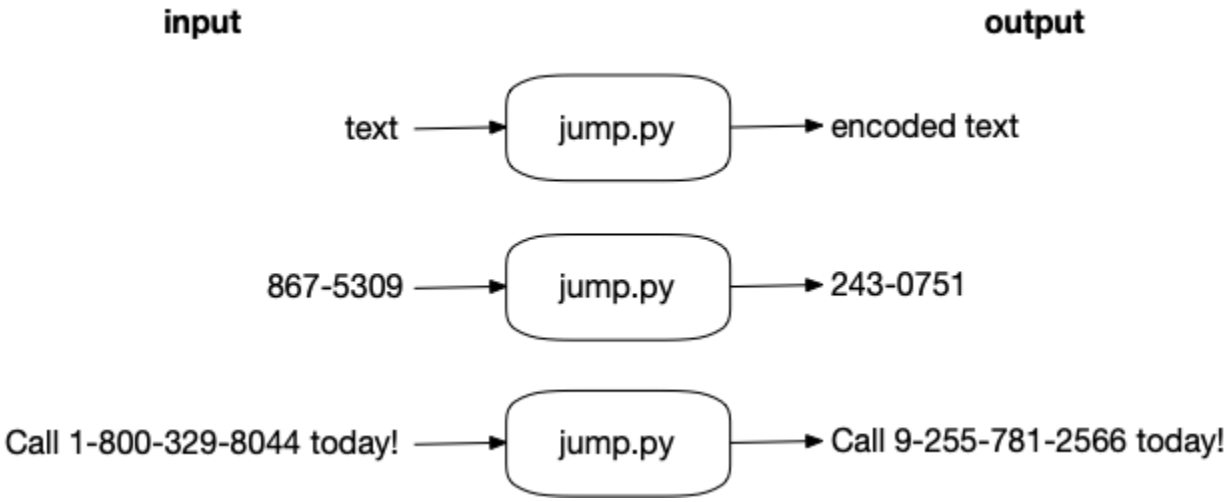
You will not have two entries but one entry with the *second* value:

```
>>> answers['favorite_color'] = 'red'
>>> answers
{'favorite_color': 'red'}
```

Keys don't have to be strings — you can also use numbers like `int` and `float`. Whatever value you use must be immutable. For instance, a `list` could not be a key because it is mutable.

Writing `jump.py`

Now let's get started with writing our program. Here is a diagram of the inputs and outputs. Note that your program will only affect the numbers in the text. Anything that is *not* a number is unchanged:



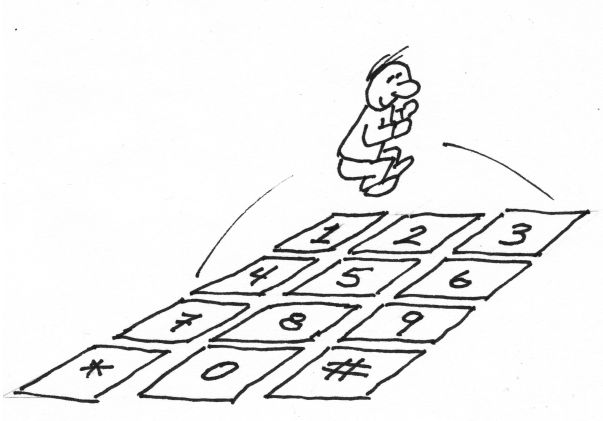
When run with no arguments, `-h`, or `--help`, your program should print a usage:

```
$ ./jump.py -h
usage: jump.py [-h] str

Jump the Five

positional arguments:
  str         Input text

optional arguments:
  -h, --help  show this help message and exit
```

Here is the substitution table for the numbers:

```
1 => 9
2 => 8
3 => 7
4 => 6
5 => 0
6 => 4
7 => 3
8 => 2
9 => 1
0 => 5
```

How would you represent this using a `dict`? Try creating a `dict` called `jump` in the REPL and then using a test. Remember that `assert` will return nothing if the statement is `True`:




```
>>> assert jumper['1'] == '9'
>>> assert jumper['5'] == '0'
```

Next, you will need a way to visit each character in the given text. I suggest you use a `for` loop like so:

```
>>> for char in 'ABC123':
...     print(char)
...
A
B
C
1
2
3
```

Now, rather printing the `char`, print the value of `char` in the `jumper` table or print the `char` itself. Look at the `dict.get` method! Also, if you read `help(print)`, you'll see there is an `end` option to change the newline that gets stuck onto the end to something else.

Hints:

- The numbers can occur anywhere in the text, so I recommend you process the input character-by-character with a `for` loop.
- Given any one character, how can you look it up in your table?
- If the character is in your table, how can you get the value (the translation)?
- If how can you `print` the translation or the value without printing a newline? Look at `help(print)` in the REPL to read about the options to `print`.
- If you read `help(str)` on Python's `str` class, you'll see that there is a `replace` method. Could you use that?

Now spend the time to write the program on your own before you look at the solutions! Use the tests to guide you.

Solution

```
#!/usr/bin/env python3
"""Jump the Five"""

import argparse


# --------------------------------------------------
def get_args():  1
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Jump the Five',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='str', help='Input text')  2

    return parser.parse_args()


# --------------------------------------------------
def main():  3
    """Make a jazz noise here"""

    args = get_args()  4
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',  5
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}

    for char in args.text:                    6
        print(jumper.get(char, char), end='')  7
    print()  8


# --------------------------------------------------
if __name__ == '__main__':
    main()  9
```

1 Define the `get_args` function first so it's easy to see when we read the program.

2 We define one positional argument called `text`.

3 Define a `main` function where the program starts.

4 Get the command-line `args`.

5 Create a `dict` for the lookup table.

6 Process each character in the text.

7 Print either the value of the `char` in the `jumper` table or the `char` if it's not present, making sure *not* to print a newline by adding `end=''`.

8 Print a newline.

9 Call the `main()` function if the program is in the "main" namespace.

Discussion

Defining the arguments

If you look at the solution, you'll see that the `get_args` function is defined first. Our program needs to define one positional argument. Since it's some "text" I expect, I call the argument `'text'` and then assign that to a variable called `text`.

```
parser.add_argument('text', metavar='str', help='Input text')
```

While all that seems rather obvious, I think it's very important to name things *what they are*. That is, please don't leave the name of the argument as `'positional'` — that does not describe what it *is*. It may seem like overkill to use `argparse` for such a simple program, but it handles the validation of the correct *number* and *type* of arguments as well as the generation of help documentation, so it's well worth the effort.

Using a `dict` for encoding

I suggested you could represent the substitution table as a `dict` where each number `key` has its substitute as the `value` in the `dict`. For instance, I know that if I jump from `1` over the `5` I should land on `9`:

```
>>> jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
...           '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
>>> jumper['1']
'9'
```

Since there are only 10 numbers to encode, this is probably the easiest way to write this. Note that the numbers are written with quotes around them, so they are actually of the type `str` and not `int` (integers). I do this because I will be reading characters from a `str`. If we stored them as actual numbers, I would have to coerce the `str` types using the `int` function:

```
>>> type('4')
<class 'str'>
>>> type(4)
<class 'int'>
>>> type(int('4'))
<class 'int'>
```

Method 1: Using a `for` loop to `print` each character

As suggested in the introduction, I can process each character of the `text` using a `for` loop. To start, I might first just see if each character of the text is in the `jumper` table using the `x in y` construct.

```
>>> text = 'ABC123'
>>> for char in text:
...     print(char, char in jumper)
...
A False
B False
C False
1 True
2 True
3 True
```

When `print` is given more than one argument, it will put a space in between each of bit of text. You can change that with the `sep` argument. Read `help(print)` to learn more.

Now let's try to translate the numbers. I could use an `if` expression where I print the value from the `jumper` table if `char` is present, otherwise print the `char`:

```
>>> for char in text:
...     print(char, jumper[char] if char in jumper else char)
...
A A
B B
C C
1 9
2 8
3 7
```

It's a bit laborious to check for every character. The `dict.get` method allows us to safely ask for a value if it is present. For instance, the letter "A" is not in `jumper`. If we try to retrieve that value, we'll get an exception:

```
>>> jumper['A']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'A'
```

But if we use `jumper.get`, there is no exception:

```
>>> jumper.get('A')
```

When a key doesn't exist in the dictionary, the special `None` value is returned:

```
>>> for char in text:
...     print(char, jumper.get(char))
...
A None
B None
C None
1 9
2 8
3 7
```

We can provide a second, optional argument to `get` that is the default value to return when the key does not exist. In our case, if a character does not exist in the `jumper`, we want to print the character itself. If we had "A," then we'd want to print "A":

```
>>> jumper.get('A', 'A')
'A'
```

But if we have, "5" then we want to print "0":

```
>>> jumper.get('5', '5')
'0'
```

So we can use that to process all the characters:

```
>>> for char in text:
...     print(jumper.get(char, char))
...
A
B
C
9
8
7
```

I don't want that newline printing after every character, so I can use `end=''` to tell Python to put the empty string at the `end` instead of a newline. When I run this in the REPL, the output is going to look funny because I have to hit <Enter> after the `for` loop for it to run, then I'll be left with `ABC987` with no newline and then the `>>>` prompt:

```
>>> for char in text:
...     print(jumper.get(char, char), end='')
...
ABC987>>>
```

And so in your code you have to add another `print()`. Mostly I wanted to point out a couple things you maybe didn't know about `print`. One is that you can have it put whatever you like for the `end`, and that you can `print()` with no arguments to print a newline. There are several other really cool things `print` can do, so I'd encourage you to read `help(print)` and try them out!

Method 2: Using a `for` loop to build a new string

There are several other ways we could solve this. I don't like all the `print` statements in the first solution, so here's another take:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    new_text = ''                           1
    for char in args.text:                  2
        new_text += jumper.get(char, char)  3
    print(new_text)                         4
```

1. In this alternate solution, you create an empty `new_text` variable.
2. Same `for` loop...
3. Append either the encoded number or the original `char` to the `new_text`
4. Print the `new_text`.

While it's fun to explore all the things we can do with `print`, that code is a bit ugly. I think it's cleaner to create a `new_text` variable and call `print` just once with that. To do this, we start off by setting a `new_text` equal to the empty string:

```
>>> new_text = ''
```

And we use our same `for` loop to process each character in the `text`. Each time through the loop, we use `+=` to append the right-hand side of the equation to the left-hand side. The `+=` adds the value on the right to the variable on the left:

```
>>> new_text += 'a'
>>> assert new_text == 'a'
>>> new_text += 'b'
>>> assert new_text == 'ab'
```

On the right, we're using the `jumper.get` method. Each character will be appended to the `new_text`:

```
>>> new_text = ''
>>> for char in text:
...     new_text += jumper.get(char, char)
...
```

Now we can call `print` one time with our new value:

```
>>> print(new_text)
ABC987
```

Method 3: Using a `for` loop to build a new `list`

This method is the same as above, but, rather than `new_text` being a `str`, it's a `list`:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    new_text = []                                  1
    for char in args.text:                         2
        new_text.append(jumper.get(char, char))    3
    print(''.join(new_text))                       4
```

1. Initialize `new_text` as an empty `list`.
2. Iterate through each character of the `text`.
3. Append the results of the `jumper.get` call to the `new_text` variable.
4. Join the `new_text` on the empty string to create a new `str` to print.

As we go through the book, I'll keep reminding you how Python treats strings and lists similarly. It's easy to go back and forth between those two types. Here I'm using `new_text` exactly the same as above, starting off with it empty and then making it longer for each character. We could actually use the exact same `+=` syntax instead of the `list.append` method:

```
for char in args.text:
    new_text += jumper.get(char, char)
```

After the `for` loop is done, we have all the new characters that need to be put back together into a new string to print.

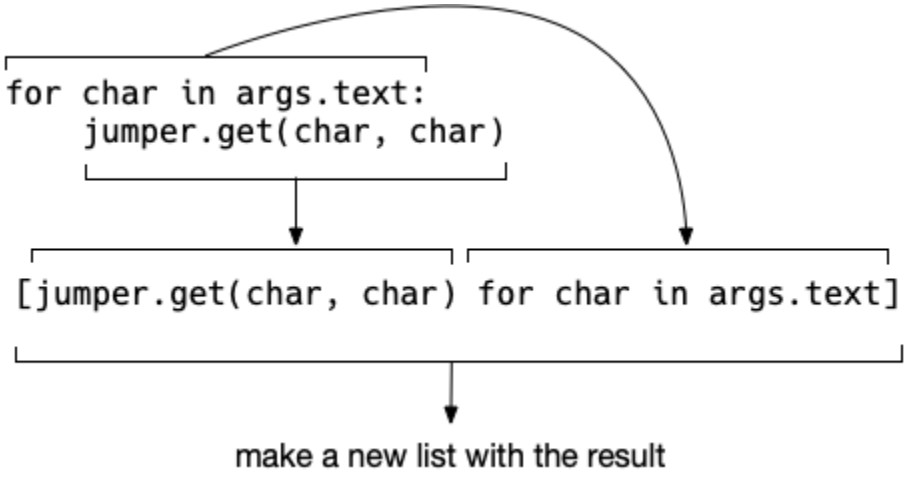
Method 4: List comprehension

A shorter solution uses a "list comprehension" which is basically a one-line `for` loop inside square brackets (`[]`) which results in a new `list` :

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    print(''.join([jumper.get(char, char) for char in args.text]))
```

A list comprension is read backwards from a `for` loop, but it's all there. It's just one line of code instead four!

```
>>> text = '867-5309'
>>> [jumper.get(char, char) for
char in text]
['2', '4', '3', '-', '0', '7', '5',
'1']
```



You can `join` that new `list` on the empty string to create a new string you can `print` :

```
>>> print(''.join([jumper.get(char, char) for char in text]))
243-0751
```

Method 5: str.translate

This last method uses a really powerful method from the `str` class to change all the characters in one step:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    print(args.text.translate(str.maketrans(jumper)))
```

The argument to `str.translate` is a translation table that defines how each character should be translated. That's exactly what our `jumper` does!

```
>>> text = 'Jenny = 867-5309'
>>> text.translate(str.maketrans(jumper))
'Jenny = 243-0751'
```

We'll revisit this function in the "Apples and Bananas" exercise where I'll explain it in greater detail.

(Not) using `str.replace`

Note that you could *not* use `str.replace` to change each number 0-9. Watch how we start off with this string:

```
>>> text = '1234567890'
```

When you change "1" to "9," now you have two 9s:

```
>>> text = text.replace('1', '9')
>>> text
'9234567890'
```

Which means when you try to change all the 9s to 1s, you end up with two 1s:

```
>>> text = text.replace('9', '1')
>>> text
'1234567810'
```

You might try to write it like so:

```
>>> text = '1234567890'
>>> for n in jumper.keys():
...     text = text.replace(n, jumper[n])
...
>>> text
'1234543215'
```

But the correctly encoded string is "9876043215", which is exactly why `str.translate` exists!

Summary

- You create a new dictionary using the `dict()` function or with empty curly brackets (`{}`).
- Dictionary values are retrieved using their keys inside square brackets or by using the `dict.get` method.
- For a `dict` called `x` , you can use `'key' in x` to determine if a key exists.
- You can use a `for` loop to iterate the characters of a `str` just like you can iterate through the elements of a `list` . You can think of strings as lists of characters.
- The `print` function takes optional keyword arguments like `end=''` which we can use to print a value to the screen without a newline.

Going Further

- Try creating a similar program that encodes the numbers with strings, e.g., "5" becomes "five", "7" becomes "seven."
- What happens if you feed the output of the program back into itself. For example, if you run `./jump.py 12345` , you should get `98760` . If you run `./jump.py 98760` , do you recover the original numbers? This is called "round-tripping," and it's a common operation with algorithms that encode and decode text.

Last updated 2020-01-14 20:37:34 -0700