Tips

Help Center

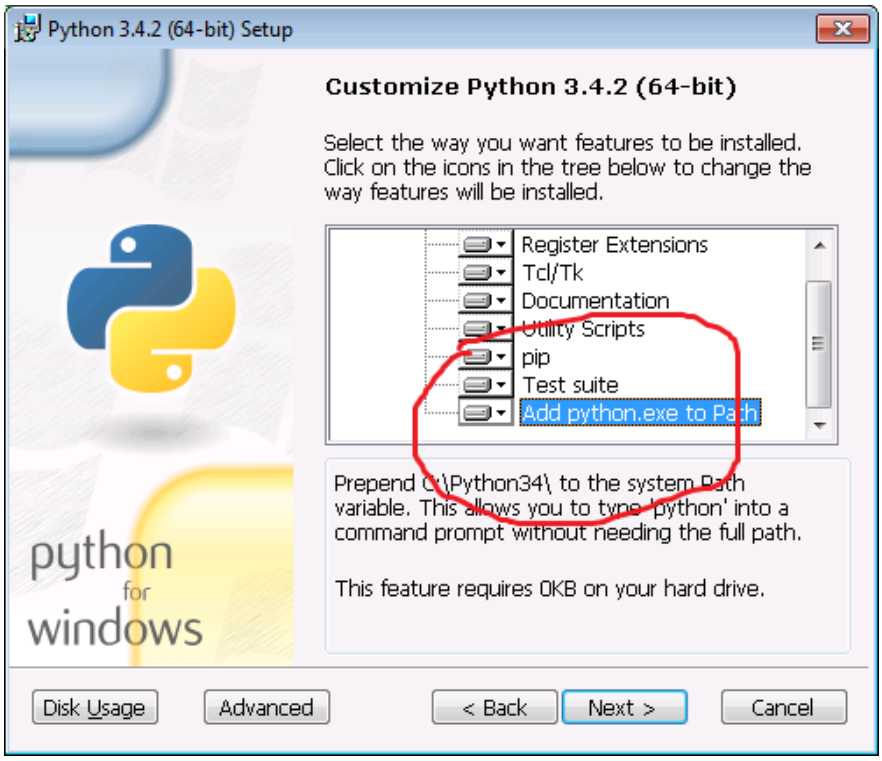
- Installing Python 3
- Getting started with the first assignment: Python Lab and Python Tutor
- Submitting assignments, usernames, and passwords
- Submission Help
- Python Commands and Help
- Docstrings and Doctests
- Testing and the REPL
- Remove `print()` if it is not asked for
- Editors and Integrated Development Environments (IDEs)
- Indentation
- Quotation marks
- Comprehensions
- Types
  - Dictionary
  - NumPy
- Plotting and viewing images
- Doctest Failures
- *What to do when my correct solution is graded incorrect?* A checklist

---

Installing Python 3

You can obtain Python 3, for free, from the official website. On the website's navigation bar, there is a Downloads link. Select the version of Python 3 for your system, and install it. Make sure you obtain Python 3, and not Python 2.

When installing, you will want to have your `Path` modified. You need to select this option when installing on a Windows system. Have the installer `Add python.exe to Path`:



To start Python, open a console (also called a shell or a terminal or, under Windows, a "Command Prompt"), and type `python3` (or perhaps just `python` if you are using Windows) to the console (or shell or terminal or Command Prompt) and hit the `Enter` key. After a few lines telling you what version you are using (e.g., Python 3.4.2), you should see >>> followed by a space. This is the prompt; it indicates that Python is waiting for you to type something. When you type an expression and hit the `Enter` key, Python evaluates the expression and prints the result, and then prints another prompt. To get out of this environment, type `quit()` and `Enter`, or `Control-D`. To interrupt Python when it is running too long, type `Control-C`.

This environment is sometimes called a **REPL**, an acronym for "read-eval-print loop." It reads what you type, evaluates it, and prints the result, if any. You should interact with Python primarily through the REPL.

There are two other ways to run Python code. First, you can import a module from within the REPL. This will be an important part of your interaction with Python. Second, you can run a Python script from the command line (outside the REPL). This is what you must do to submit your work to us.

More documentation about installing and using Python 3 for Windows or Mac OS X can also be found on the Python website:

- Windows
- Mac OS X

Getting started with the first assignment: Python Lab and Python Tutor

One way you can receive some assistance with the Python Lab assignment is with Python Tutor. On this site, you can test, debug, run and visualize your code. You can step through it forwards or backwards in time. You can even ask it to check your answers to problems in the Python Lab

assignment. **However:** Python Tutor does **not** submit your answers to Coursera. Please make sure you submit your work using the `coursera_submit.py` script.

Submitting assignments, usernames, and passwords

Assignment submissions are done by using the `coursera_submit.py` submission script. This script should be in your `matrix` directory, along with all of your other work. Submission requires authentication. You can find your submission login and submission password on the Programming Assignments page. Your submission login and submission password are not necessarily the same login and password you use for Coursera.

When submitting your assignment, you have three ways you can authenticate. The first two methods below involve input at the command line; the last method involves creating a text file.

1. **Entering the submission username and submission password when prompted.** When running the script, you will be asked to enter your submission login and submission password. For example, suppose you are a user, Homer, with username `homer@quijibo.doh` and password `forbiDdend0nut`. Your submission process will start out like this:

   ```
   $ python3 coursera_submit.py python_lab.py
   No profile.txt found
   = Coding the Matrix Homework and Lab Submission
   Importing your stencil file
   Fetching problems
   username: homer@quijibo.doh
   password: forbiDdend0nut
   This assignment has the following parts:
     1) Minutes in a Week
   ```

   If you are running on Windows, you might need to use the command `python` instead of `python3`.

2. **Supplying the submission login and submission password.** When running the script, you may specify your submission login and password:

   ```
   $ python3 coursera_submit.py --username homer@quijibo.doh --password forbiDdend0nut
    python_lab.py
   No profile.txt found
   = Coding the Matrix Homework and Lab Submission
   Importing your stencil file
   Fetching problems
   This assignment has the following parts:
     1) Minutes in a Week
   ```

3. **Using `profile.txt`.** You can create a `profile.txt` file that includes your submission login and submission password. Place this file with all of your other work in your `matrix` working directory. The contents of the file are two lines:

```
USERNAME homer@quijibo.doh
PASSWORD forbiDdend0nut
```

When running the submission script, we are now no longer prompted for a login and password:

```
$ python3 coursera_submit.py python_lab.py
= Coding the Matrix Homework and Lab Submission
Importing your stencil file
Fetching problems
This assignment has the following parts:
  1) Minutes in a Week
```

Also different is that the output no longer mentions that `profile.txt` is missing. If you see this message, and you think you have created a `profile.txt` file, make sure that the `profile.txt` file, along with your `coursera_submission.py` script and your work are all in the same directory.

You may submit different parts of the assignment during the submission process. When for example, when running `python3 coursera_submit.py python_lab.py`, we can see

```
  26) (Task 26) Procedure nextInts
  27) (Task 27) Procedure cubes
  28) (Task 28) Procedure dict2list
  29) (Task 29) Procedure list2dict

Which parts do you want to submit? (Ex: 1, 4-7):
```

You may also submit at the command-line, by specifying the problems at the end:

```
$ python3 coursera_submit.py python_lab.py 2-4
= Coding the Matrix Homework and Lab Submission
Importing your stencil file
Fetching problems
== Submitting "(Task 2) Remainder"
Your submission has been accepted and will be graded shortly.
== Submitting "(Task 3) Divisibility"
Your submission has been accepted and will be graded shortly.
== Submitting "(Task 4) Conditional Expression"
Your submission has been accepted and will be graded shortly.
```

After you submit, the system will process your submission. This may take some time, and you may see a score of 0/1 until it is done processing. To check if the grader has graded a problem, you can click on the `View` button next to the problem on the assignment page. This button is located in the `Feedback` column.

Submission Help

When the submission script is not working the way you expect it to, here are a few things you can check.

- **Make sure your code works in the REPL.** If your code is well-written, you should not see any error messages. For example, suppose in `python_lab.py` we have the line

  ```
  print(1+)
  ```

  When importing the file in the REPL, we see:

  ```
  >>> import python_lab
  Traceback (most recent call last):
    File "", line 1, in
    File "python_lab.py", line 6
      print(1+)
              ^
  SyntaxError: invalid syntax
  ```

  If you have problems with your file, the submission script will notify you, and prevent you from submitting. For example:

  ```
  $ python3 coursera_submit.py The_Vector_Space_problems.py
  CHALLENGE URL  https://class.coursera.org/matrix-002/assignment/challenge
  = Coding the Matrix Homework and Lab Submission
  Importing your stencil file
  name 'foo' is not defined
  !! It seems that you have an error in your stencil file. Please make sure Python ca
  n import your stencil before submitting, as in...

        underwood:matrix klein$ python3
        Python 3.4.1
        >>> import The_Vector_Space_problems
  ```

  When you see a message like this, it means you will need to debug your code. One way to start is by doing what the script suggests. In the REPL:

  ```
  >>> import The_Vector_Space_problems
  ```

- **Verify that you are using the correct stencil.** Your stencils should be downloaded from the links we provide. If you do not use the proper stencil file, you will see the following message when you try to submit:

```
This is not a Coursera stencil.  Make sure your stencil is obtained from http://gra
ding.codingthematrix.com/coursera/
```

- **Make sure you are using Python 3 and have the necessary modules.** The submission script may not work on older versions of Python. Using Python 3.1, for example, I receive the following message when using the submission script:

  ```
  $ ~/Software/Python-3.1.5/python coursera_submit.py python_lab.py
  Traceback (most recent call last):
    File "coursera_submit.py", line 360, in
      import argparse
  ImportError: No module named argparse
  ```

  If you are using Python 3.1 and are missing `argparse`, you may find this page to be helpful.

- **Check that your version of Python has https support.** If you see this message:

  ```
  urllib.error.URLError: <urlopen error unknown url type: https>
  ```

  it means you may be missing support for https. This issue may arise if you built Python yourself, or you obtained Python from an unofficial source. Try the following:

  1. Download the latest version of `coursera_submit.py`.
  2. Submit using the `--http` option. For example, to submit `python_lab.py`, we may enter

     ```
     python3 coursera_submit.py --http python_lab.py
     ```

Python Commands and Help

If you want information about a function, you can use the built-in `help()` function. You may exit the `help()` view by entering the `q` key. For example, if you want information about the built-in `abs()` function, you can enter `help(abs)` in the command-line:

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(...)
    abs(number) -> number

    Return the absolute value of the argument.
```

Similarly, if you want information about the `plot()` function from `plotting.py`:

```
>>> from plotting import plot
>>> help(plot)
```

```
Help on function plot in module plotting:

plot(L, scale=4, dot_size=3, browser=None)
    plot takes a list of points, optionally a scale (relative to a 200x200 frame),
    optionally a dot size (diameter) in pixels, and optionally a browser name.
    It produces an html file with SVG representing the given plot,
    and opens the file in a web browser. It returns nothing.
```

Python module documentation can also be viewed using `help()`. For example, to view documentation on the `math` module:

```
>>> import math
>>> help(math)
Help on module math:

NAME
    math

MODULE REFERENCE
    http://docs.python.org/3.4/library/math

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.
```

You can also view official documentation about native built-in functions here: https://docs.python.org/3/. Note that this site does NOT contain documentation for the course resources (that is, anything found here: http://resources.codingthematrix.com/).

Docstrings and Doctests

In Python, we may write documentation using docstring syntax. Docstrings appear at the start of definitions and are used to describe what is being written and implemented in between pairs of either

`"""` or `'''`. For example, in a file `my_program.py`, we might see:

```
def cube(x):
    """
    Return the cube of x.

    Args:
        x: Number.  The number to cube.
    Returns:
        Number.  The cube of x.

    >>> cube(3)
    27
    >>> cube(0)
    0
    >>> cube(-1)
    -1
    """
    return x*x*x
```

The function `cube()` is defined on the first line. After the initial `"""` and before `>>> cube(3)` are lines describing what the function does. Normally the first lines below the function definition describe the function. Proceeding this, there is often a description of what the function takes as input and gives as output, as in this example.

What you see in `help()` matches the docstring content. If we take a look inside `plotting.py`, and find the `plot()` function we saw earlier, we'll see:

```
def plot(L, scale=4, dot_size = 3, browser=None):
    """ plot takes a list of points, optionally a scale (relative to a 200x200 frame),
        optionally a dot size (diameter) in pixels, and optionally a browser name.
        It produces an html file with SVG representing the given plot,
        and opens the file in a web browser. It returns nothing.
    """
```

Below the function description of `cube()` are entries that have >>>, like in the prompt. These entries are called doctests. Immediately after the >>> lines are what we expect the program to produce upon executing the preceding line, just like in the prompt. One powerful feature of Python is its ability to include these tests in the code itself. We can run doctests on `my_program.py` with `python3 -m doctest my_program.py`:

```
$ python3 -m doctest my_program.py
$
```

There is no output, indicating that the doctests were successful. However, suppose we had a file,

`my_other_program.py`, containing

```
def fourth_power(x):
    '''
    >>> fourth_power(2)
    16
    '''
    return x*x
```

Here, we can see why docstrings are important: the function name implies that what will be returned is the fourth power of the input, but the function only returns the second power of the input. On inspection of the doctest, however, we might presume that the programmer who wrote this really meant *fourth power*. Running a doctest with `python3 -m doctest my_other_program.py`, we see:

```
$ python3 -m doctest my_other_program.py
**********************************************************************
File "/Users/homer/matrix/my_other_program.py", line 3, in my_other_program.fourth_pow
er
Failed example:
    fourth_power(2)
Expected:
    16
Got:
    4
**********************************************************************
1 items had failures:
   1 of   1 in my_other_program.fourth_power
***Test Failed*** 1 failures.
```

Before submitting, test your code in the REPL. This is the best way to get feedback! The course grader will not give any feedback.

Working with your code interactively can save you a lot of pain, agony and frustration. Many hard-to-find errors can be caught before submitting a script. For example, let's take a look at this script, `hello.py`:

```
def say_hello():
    '''Return the string `hello world`.'''
    x = 'hello' + 'world!'
        return x
```

What's wrong? Well, let's run it from the command line:

```
$ python3
>>> from hello import say_hello
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/homer/Desktop/hello.py", line 4
    return x
           ^
TabError: inconsistent use of tabs and spaces in indentation
```

Tab and spacing issues. Now, this might be easier to spot when you look above, but if your editor is configured so that a tab looks like four spaces, it would be very difficult to see. Let's fix the code and run it again. `hello.py` now reads:

```
def say_hello():
    '''Return the string `hello world`.'''
    x = 'hello' + 'world!'
    return x
```

Running it in the command line:

```
$ python3
>>> from hello import say_hello
>>> say_hello()
'helloworld!'
```

This isn't quite right. The words 'hello' and 'world' don't have a space in between. Let's fix `hello.py`:

```
def say_hello():
    '''Return the string `hello world`.'''
    x = 'hello ' + 'world!'
    return x
```

And run it again:

```
$ python3
>>> from hello import say_hello
>>> say_hello()
'hello world!'
```

Note that the function could have been written out and tested in the terminal:

```
$ python3
>>> def say_hello():
...     '''Return the string `hello world`.'''
...     x = 'hello ' + 'world!'
...     return x
...
```

```
>>> say_hello()
'hello world!'
```

Remove `print()` if it is not asked for

Using `print()` is a very useful way to test code. However, the grader expects only what is asked for in the assignment: no more, no less. This includes any output from `print()`.

Example: if a problem asks for a function `factorial(n)` that computes the factorial of a non-negative integer `n`, *n!*, this solution will be accepted by the grader:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

This solution will be rejected by the grader:

```
def factorial(n):
    print('n is currently', n)
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Editors and Integrated Development Environments (IDE)

Please configure your software environment with the following settings:

1. **Python:** Python 3 is being used.
2. **Tabs:** Four space characters are used for indentation.
3. **Special characters:** Check that quotation marks are not altered.

When indenting, use four spaces and not tabs

The code provided and the grader will expect indentations to use the space character, and not tab. If you use the tab key, make sure your editor is configured so that spaces are inserted. The convention used here: four spaces per indent.

Make sure that the correct quotation mark characters are used

In Python, you will need to use the single quotation mark character `'`, which is located next to the

`Enter` key on most keyboards. This character normally shares the same key as the double quotation mark character `"`. Some editors may attempt to replace your characters automatically with another character. For example, `'` might automatically be replaced with `’`. Please configure your editor so that the single quotation mark character `'` and double quotation mark character `"` are not altered. Your code may not be accepted by Python or the grader if you use special characters.

- Example: We have two files, `woohoo.py` and `doh.py`. Each contain one line:
  - `woohoo.py`: `print('woohoo!')`
  - `doh.py`: `print(’doh!’)`

  Notice that `doh.py` is using `’` and not `'`. When we try to run these files with `python`, we see:

```
$ python3 woohoo.py
woohoo!
$ python3 doh.py
  File "doh.py", line 1
    print(’doh!’)
              ^
SyntaxError: invalid character in identifier
```

- Example: We have two files, `embiggens.py` and `cromulent.py`. Each contain one line:
  - `embiggens.py`: `print("A noble spirit embiggens the smallest man.")`
  - `cromulent.py`: `print(“I don't know why; it’s a perfectly cromulent word.”)`

  Notice that `cromulent.py` is using `“` and `”`, but not `"`. When we try to run these files with `python`, we see:

```
$ python3 embiggens.py
A noble spirit embiggens the smallest man.
$ python3 cromulent.py
  File "cromulent.py", line 1
    print(“I don't know why; it’s a perfectly cromulent word.”)
          ^
SyntaxError: invalid character in identifier
```

Write comprehensions in one line

Problems that ask for a comprehension should be done in one line. The grader may reject anything you do using multiple lines.

Example: Creating a set of 10 elements, which consists of consecutive odd integers starting from 1. This is fine:

```
{2*i + 1 for i in range(10)}
```

This is not fine:

```
{2*i + 1 \
    for i in range(10)}
```

Types

The grader expects that your solutions have the type as specified in the assignment. For example, if a problem asks for a set, you should answer with a set, and not a list. Functions may also expect a certain type as input. For example, a function may require a list of numbers. In general, you will want to avoid using types generated by external modules, such as `collections.defaultdict` and `numpy.float64`, and use native datatypes, such as `str`, `dict` and `int`. You can check the type of an expression with the `type()` function:

```
>>> type(set)
<class 'type'>
>>> type({-4, -2, 1, 2, 5, 0})
<class 'set'>
>>> type([1,2,3])
<class 'list'>
>>> type(1 + 4)
<class 'int'>
>>> type('hello')
<class 'str'>
>>> type((1,0,3))
<class 'tuple'>
```

- Example: A set with zero elements. Normally you can create a set with `{` and `}`:

  ```
  >>> S = {i for i in range(5)}
  >>> S
  {0, 1, 2, 3, 4}
  >>> type(S)
  <class 'set'>
  ```

  However, when making a set with zero elements, you cannot use `{}`:

  ```
  >>> T = {}
  >>> type(T)
  <class 'dict'>
  ```

  If you would like to make an empty set, you can use `set()`:

  ```
  >>> U = set()
  >>> type(U)
  <class 'set'>
  ```

```
>>> len(U)
0
```

- Example: dictionary. When an assignment asks for a dictionary, you should use one with type `dict`:

```
>>> type({1:1})
<class 'dict'>
>>> from collections import *
>>> type(defaultdict(int))
<class 'collections.defaultdict'>
```

- Example: NumPy. A number NumPy produces may not necessarily be a float or int. You will want to avoid returning NumPy types, and return native Python datatypes instead:

```
>>> import numpy
>>> type(numpy.sum([1,2,3]))
<class 'numpy.int64'>
>>> type(sum([1,2,3]))
<class 'int'>
>>> type(numpy.mean([1,2,3]))
<class 'numpy.float64'>
>>> type(sum([1,2,3])/3)
<class 'float'>
```

Plotting and viewing images

If you have trouble using the scripts to view images (for example, using the `plot()` function in `plotting.py`) here are a few things that might help:

- Using different web browsers. Try installing Firefox and set that to be your default browser.
- Script inputs. For example, `plot()` in `plotting.py` takes an optional 4th argument, browser. You can try setting this to a browser you have installed.

Python can only use what it recognizes. To see a list of supported browsers, you can use the `webbrowser` module. On the system I am using, the browser argument in `plot()` can be any of the keys seen below. Your system will probably not have this same set of keys, so make sure you check before assigning browser a string.

```
>>> import webbrowser
>>> webbrowser._browsers.keys()
dict_keys(['konqueror', 'firefox', 'iceape', 'links', 'w3m', 'xdg-open', 'gvfs-open',
'mozilla', 'x-www-browser', 'www-browser', 'gnome-open', 'lynx', 'iceweasel'])
```

A list of web browser types can be found here: https://docs.python.org/3/library/webbrowser.html.

If these options don't seem to be working for you, then you might consider editing the script you are using.

- Example: `plot()` in `plotting.py`. This script attempts to generate an HTML file containing the output of the `plot()` function in a temporary directory. This might cause problems for some people. We may have the script place the HTML file in the directory you are using Python in. To do this, we need to change a line:

  1. Line 25. This has `hpath = create_temp('.html')`. Change this to read:

  ```
  hpath = os.path.join(os.getcwd(), 'myplot.html')
  ```

Doctest Failures

Don't worry if a doctest fails! Python is finicky, and your program may actually be correct! You can always check your program output to make sure that what you are getting is correct.

- Example: Suppose you are working on Problem 5.14.13 in the text, and you wrote:

  ```
  >>> a0 = Vec({'a', 'b', 'c', 'd'}, {'a': 1})
  >>> a1 = Vec({'a', 'b', 'c', 'd'}, {'b': 1})
  >>> a2 = Vec({'a', 'b', 'c', 'd'}, {'c': 1})
  >>> ans = Vec({'a', 'c', 'b', 'd'}, {'a': 2, 'c': 6, 'b': 4, 'd': 0})
  >>> rep2vec(Vec({0, 1, 2}, {0: 2, 1: 4, 2: 6}), [a0, a1, a2])
  Vec({'a', 'c', 'b', 'd'}, {'a': 2, 'c': 6, 'b': 4, 'd': 0})
  ```

  This might fail, because Python did not like the space in `'d'}, {'a': 2`. That is, it may expect

  ```
  Vec({'a', 'c', 'b', 'd'},{'a': 2, 'c': 6, 'b': 4, 'd': 0})
  ```

  One way to get around these issues is to modify your tests. In this case, we can check against the known answer:

  ```
  >>> a0 = Vec({'a', 'b', 'c', 'd'}, {'a': 1})
  >>> a1 = Vec({'a', 'b', 'c', 'd'}, {'b': 1})
  >>> a2 = Vec({'a', 'b', 'c', 'd'}, {'c': 1})
  >>> ans = Vec({'a', 'c', 'b', 'd'}, {'a': 2, 'c': 6, 'b': 4, 'd': 0})
  >>> rep2vec(Vec({0, 1, 2}, {0: 2, 1: 4, 2: 6}), [a0, a1, a2]) == ans
  True
  ```

- Example: Representation issues might get in the way. In Problem 5.14.14 for example, if you had

  ```
  >>> a0 = Vec({'a', 'b', 'c', 'd'}, {'a': 1})
  >>> a1 = Vec({'a', 'b', 'c', 'd'}, {'b': 1})
  >>> a2 = Vec({'a', 'b', 'c', 'd'}, {'c': 1})
  ```

```
    >>> ans = Vec({0, 1, 2}, {0: 3, 1: 0, 2: -2})
    >>> vec2rep([a0, a1, a2],Vec({'a', 'b', 'c', 'd'}, {'a': 3, 'c': -2}))
    Vec({0, 1, 2}, {0: 3, 1: 0, 2: -2})
```

Python may not like it because it wants to see floating-point numbers:

```
    >>> a0 = Vec({'a', 'b', 'c', 'd'}, {'a': 1})
    >>> a1 = Vec({'a', 'b', 'c', 'd'}, {'b': 1})
    >>> a2 = Vec({'a', 'b', 'c', 'd'}, {'c': 1})
    >>> ans = Vec({0, 1, 2}, {0: 3, 1: 0, 2: -2})
    >>> vec2rep([a0, a1, a2], Vec({'a', 'b', 'c', 'd'}, {'a': 3, 'c': -2}))
    Vec({0, 1, 2},{0: 3.0, 1: 0.0, 2: -2.0})
```

- Example: Suppose you are working on Problem 5.14.18 in the text. The problem asks for a list of elements that satisfy a few conditions, and the example given is

```
    >>> a0 = Vec({'a', 'b', 'c', 'd'}, {'a': 1})
    >>> a1 = Vec({'a', 'b', 'c', 'd'}, {'b': 1})
    >>> a2 = Vec({'a', 'b', 'c', 'd'}, {'c': 1})
    >>> a3 = Vec({'a', 'b', 'c', 'd'}, {'a': 1, 'c': 3})
    >>> superset_basis([a0, a3], [a0, a1, a2])
    [Vec({'a', 'c', 'b', 'd'},{'a': 1, 'c': 3}), Vec({'a', 'c', 'b', 'd'},{'a': 1})
, Vec({'a', 'c', 'b', 'd'},{'b': 1})]
```

Python expects the output to be given just as written, but a valid solution only needs to contain those 3 vectors. We can check the solution without worrying about the order by slightly modifying the test. We know that the output of superset_basis has 3 elements, which are a0, a1 and a3, so:

```
    >>> a0 = Vec({'a', 'b', 'c', 'd'}, {'a': 1})
    >>> a1 = Vec({'a', 'b', 'c', 'd'}, {'b': 1})
    >>> a2 = Vec({'a', 'b', 'c', 'd'}, {'c': 1})
    >>> a3 = Vec({'a', 'b', 'c', 'd'}, {'a': 1, 'c': 3})
    >>> result = superset_basis([a0, a3], [a0, a1, a2])
    >>> len(result) == 3
    True
    >>> all([x in result for x in [a0, a1, a3]])
    True
```

- Example: `print()` statements can throw off the doctest. Let's say we have a file, `foo.py`, with the following:

```
def return_bar():
    """
    return the string 'bar'
    >>> return_bar()
    'bar'
```

```
    """
    x = 'bar'
    print(x)
    return x
```

Running the doctest with `python3 -m doctest foo.py`, we get:

```
$ python3 -m doctest foo.py
**********************************************************************
File "foo.py", line 4, in foo.return_bar
Failed example:
    return_bar()
Expected:
    'bar'
Got:
    bar
    'bar'
**********************************************************************
1 items had failures:
   1 of   1 in foo.return_bar
***Test Failed*** 1 failures.
```

The culprit is the `print()` statement! Commenting it out, we make the world a better place:

```
def return_bar():
    """
    return the string 'bar'
    >>> return_bar()
    'bar'
    """
    x = 'bar'
    #print(x)
    return x
```

```
$ python3 -m doctest foo.py
$
```

No failures!

What do I do when the grader says my correct solution is incorrect? A checklist

1. **Give the grader time to process your solution:** After you submit, the system will process your submission. This may take some time, and you may see a score of 0/1 until it is done processing.

To check if the grader has graded a problem, you can click on the View button next to the problem on the assignment page. This button is located in the Feedback column.

2. **Comprehensions:** If the solution is supposed to be a comprehension, make sure it is on one line.
3. **Variables:** Often the stencil has you assign your solution to a variable. Make sure you don't change the name of that variable (or assign to the same variable later in the stencil).
4. **Mutation:** Make sure your procedure is not mutating its arguments. You can check that by assigning values to variables, calling your procedure with those values as arguments, and then verifying that the values of the variables have not changed.
5. **Procedures:** Often the stencil has you define a procedure with a given name and with given arguments. Don't change the name or the arguments.
6. **Prints:** Remove print statements from your code.
7. **Operators:** If you are doing operations on Vecs or Mats, make sure you are using the operators (such as + and * and == and []) rather than the named procedures (such as `getitem` and `dot` and `equal`).
8. **Values**: Don't just use `print( )` or == to verify that `Vecs` and `Mats` (e.g. the values returned by your procedures) are correct. Look at the representation of the value, e.g. `"Vec({'a','b','c'}, {'a':1, ...})"`.
9. **Files**: Make sure you are working with the correct file. If you have multiple versions of an assignment, make sure you are submitting the correct one. Make sure you have saved your changes before submitting.
10. **Doctests:** Run the doctests provided in the stencil to make sure your code passes those. This includes using copy-and-paste to try out the doctests in the REPL.
11. **TEST YOUR CODE:** Import your code from the Python REPL, and try it out on examples of your own devising. As an experienced programmer, you know about creating tests that probe the code, that challenge it with tricky inputs.
12. **Subroutines:** Make sure you understand what arguments are expected by procedures you are using. (For example, perhaps the procedure expects a list of Vecs but you are passing a Mat.)
13. **Specs:** Make sure your procedure matches the specification given in the PDF file. (For example, perhaps the problem description in the PDF file asks you to write a procedure that takes a Mat as argument, but your code expects a list of Vecs.) You can check the type of your variables with `type()`.
14. **Explanations:** Go back to the description of the problem in the pdf file, and see if perhaps you have misinterpreted it. Sometimes, despite our best efforts, our problem specifications are ambiguous or unclear.
15. **Forum:** Seek help on the Forum, in the thread addressing the specific problem you are struggling with. Don't post your code. Instead, provide some input-output examples (in doctest form).
16. **Stop:** Recognize when you've spent enough time on a problem, and just move on. Let that problem go. You don't have to get 100% to be successful in this course. But if you are completely convinced that your solution is correct, indicate it with a comment:
    # CORRECT
    We will regularly review the solutions thus marked. This won't affect the grade you are assigned for that part, but if we truly missed something, it will help us improve the grading for the future.

Afterwards: When you have struggled and achieved success, watch out for postings by other students who are still struggling with the same problem. Maybe you can post a helpful suggestion (without

giving away the solution).

Created Wed 21 Jan 2015 10:53 AM PST

Last Modified Wed 4 Mar 2015 7:41 PM PST