Importing
Python Basics 3

Dr. Ryan Krauss

Grand Valley State University

pdm

The Power of Importing

- Python by itself is cool, but somewhat limited

pdm

The Power of Importing

- Python by itself is cool, but somewhat limited
- a good deal of the power of Python comes from the huge number of modules available

pdm

The Power of Importing

- Python by itself is cool, but somewhat limited
- a good deal of the power of Python comes from the huge number of modules available
  - numpy, scipy, matplotlib, os, system, time, cherrypy, ...

pdm

The Power of Importing

- Python by itself is cool, but somewhat limited
- a good deal of the power of Python comes from the huge number of modules available
  - numpy, scipy, matplotlib, os, system, time, cherrypy, ...
- you have to import the modules to use them

pdm

Four ways to import

1. `import numpy`

pdm

Four ways to import

1. `import numpy`
   - all functions or variable can be accessed using `numpy.`

pdm

Four ways to import

1. `import numpy`
   - all functions or variable can be accessed using `numpy.`
     - i.e. `numpy.pi`

pdm

Four ways to import

1. `import numpy`
   - all functions or variable can be accessed using `numpy.`
     - i.e. `numpy.pi`
   - protects against namespace collisions

pdm

Four ways to import

1. `import numpy`
   - all functions or variable can be accessed using `numpy.`
     - i.e. `numpy.pi`
   - protects against namespace collisions
     - what if two different modules have a variable `pi` and they refer to different things?

pdm

Four ways to import

1. `import numpy`
   - all functions or variable can be accessed using `numpy.`
     - i.e. `numpy.pi`
   - protects against namespace collisions
     - what if two different modules have a variable `pi` and they refer to different things?
   - typing `numpy.` gets old

pdm

Four ways to import

1. `import numpy`
   - all functions or variable can be accessed using `numpy.`
     - i.e. `numpy.pi`
   - protects against namespace collisions
     - what if two different modules have a variable `pi` and they refer to different things?
   - typing `numpy.` gets old
2. `import numpy as np`

pdm

Four ways to import

1. `import numpy`
   - all functions or variable can be accessed using `numpy.`
     - i.e. `numpy.pi`
   - protects against namespace collisions
     - what if two different modules have a variable `pi` and they refer to different things?
   - typing `numpy.` gets old
2. `import numpy as np`
   - `np.pi`

pdm

Four ways to import

1. `import numpy`
   - all functions or variable can be accessed using `numpy.`
     - i.e. `numpy.pi`
   - protects against namespace collisions
     - what if two different modules have a variable `pi` and they refer to different things?
   - typing `numpy.` gets old
2. `import numpy as np`
   - `np.pi`
   - still projects against namespace collisions

pdm

Four ways to import

1. `import numpy`
   - all functions or variable can be accessed using `numpy.`
     - i.e. `numpy.pi`
   - protects against namespace collisions
     - what if two different modules have a variable `pi` and they refer to different things?
   - typing `numpy.` gets old
2. `import numpy as np`
   - `np.pi`
   - still projects against namespace collisions
   - if you actually used `np = 7` or something, you would break the import

pdm

Four ways to import

1. `import numpy`
   - all functions or variable can be accessed using `numpy.`
     - i.e. `numpy.pi`
   - protects against namespace collisions
     - what if two different modules have a variable `pi` and they refer to different things?
   - typing `numpy.` gets old
2. `import numpy as np`
   - `np.pi`
   - still projects against namespace collisions
   - if you actually used `np = 7` or something, you would break the import
   - some people use `import numpy as N`, but I think this is risky

pdm

Four ways to import

1. `import numpy`
   - all functions or variable can be accessed using `numpy.`
     - i.e. `numpy.pi`
   - protects against namespace collisions
     - what if two different modules have a variable `pi` and they refer to different things?
   - typing `numpy.` gets old
2. `import numpy as np`
   - `np.pi`
   - still projects against namespace collisions
   - if you actually used `np = 7` or something, you would break the import
   - some people use `import numpy as N`, but I think this is risky
     - you can never use `N` anywhere in your code without messing things up

pdm

Four ways to import (#3)

3. `from numpy import pi, arange`

pdm

Four ways to import (#3)

3. `from numpy import pi, arange`
   - `pi` and `arange` now work by themselves without `numpy.` or `np.`

pdm

Four ways to import (#3)

3. `from numpy import pi, arange`
   - `pi` and `arange` now work by themselves without `numpy.` or `np.`
     - `pi` and `arange` are now in the global namespace

pdm

Four ways to import (#3)

3. `from numpy import pi, arange`
   - `pi` and `arange` now work by themselves without `numpy.` or `np.`
     - `pi` and `arange` are now in the global namespace
   - you have to give a list of all the functions or variables you need

pdm

Four ways to import (#3)

3. `from numpy import pi, arange`
   - `pi` and `arange` now work by themselves without `numpy.` or `np.`
     - `pi` and `arange` are now in the global namespace
   - you have to give a list of all the functions or variables you need
     - this is kind of cumbersome

pdm

Four ways to import (#4)

4. `from numpy import *`

pdm

Four ways to import (#4)

4. `from numpy import *`
   - load everything in the `numpy` module into the global namespace

pdm

Four ways to import (#4)

4. `from numpy import *`
   - load everything in the `numpy` module into the global namespace

pdm

Four ways to import (#4)

4. `from numpy import *`
   - load everything in the `numpy` module into the global namespace

My old habit is to use these two lines at the beginning of every script:

```
from matplotlib.pyplot import *
from numpy import *
```

- in some ways, this is the easiest way to not have to think about modules and make IPython easy

pdm

Four ways to import (#4)

```
from matplotlib.pyplot import *
from numpy import *
```

- two risks:

pdm

Four ways to import (#4)

```
from matplotlib.pyplot import *
from numpy import *
```

- two risks:
  - namespace collisions

pdm

Four ways to import (#4)

```
from matplotlib.pyplot import *
from numpy import *
```

- two risks:
  - namespace collisions
  - masks where things come from, making it harder to learn from or maintain code

pdm

Four ways to import (#4)

```
from matplotlib.pyplot import *
from numpy import *
```

- two risks:
  - namespace collisions
  - masks where things come from, making it harder to learn from or maintain code
- some advanced Python users consider this to be poor practice

pdm

Four ways to import (#4)

```
from matplotlib.pyplot import *
from numpy import *
```

- two risks:
  - namespace collisions
  - masks where things come from, making it harder to learn from or maintain code
- some advanced Python users consider this to be poor practice
- IPython will eventually remove the `%pylab` option

pdm

Four ways to import (#4)

```
from matplotlib.pyplot import *
from numpy import *
```

- two risks:
  - namespace collisions
  - masks where things come from, making it harder to learn from or maintain code
- some advanced Python users consider this to be poor practice
- IPython will eventually remove the `%pylab` option
- `spyder` does this style of import if you click the option in settings to load `numpy` and `pylab`

pdm

My Recommendation

This is the currently accepted best practice:

```
import matplotlib.pyplot as plt
import numpy as np
```

- slightly more typing:

pdm

My Recommendation

This is the currently accepted best practice:

```
import matplotlib.pyplot as plt
import numpy as np
```

- slightly more typing:
  - `t = np.arange(0,1,0.01)`

pdm

My Recommendation

This is the currently accepted best practice:

```
import matplotlib.pyplot as plt
import numpy as np
```

- slightly more typing:
  - `t = np.arange(0,1,0.01)`
  - `y = np.sin(2*np.pi*t)`

pdm

My Recommendation

This is the currently accepted best practice:

```
import matplotlib.pyplot as plt
import numpy as np
```

- slightly more typing:
  - `t = np.arange(0,1,0.01)`
  - `y = np.sin(2*np.pi*t)`
- forces you to learn what comes from where

pdm