Standard Libraries

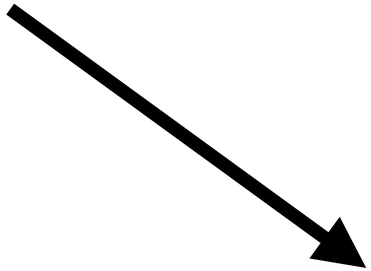
May 17th

Modules

- Smallest unit of reusable python code.
- Basically a file with functions and statements
- We write modules allll the time

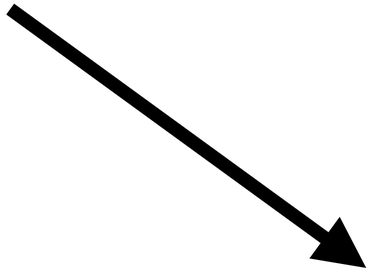
Modules

- Smallest unit of reusable python code.
- Basically a file with functions and statements
- We write modules allll the time



Modules

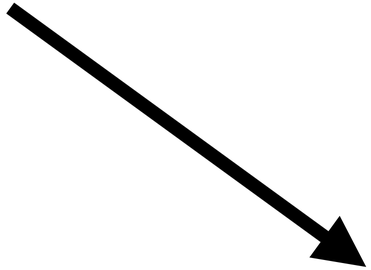
- Smallest unit of reusable python code.
- Basically a file with functions and statements
- We write modules allll the time



Package

Modules

- Smallest unit of reusable python code.
- Basically a file with functions and statements
- We write modules allll the time



Package

- A logical collection of modules
- E.g. Numpy

Standard Library

- The packages and modules that come with python
- Viewed as important/fundamental to the developers
- To access them, just need to import! (Packages not in S.L. must be downloaded with something like pip)

```
sound/
      __init__.py
      formats/
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

```
sound/
        __init__.py
        formats/
                __init__.py
                wavread.py
                wavwrite.py
                aiffread.py
                aiffwrite.py
                auread.py
                auwrite.py
                ...
        effects/
                __init__.py
                echo.py
                surround.py
                reverse.py
                ...
        filters/
                __init__.py
                equalizer.py
                vocoder.py
                karaoke.py
                ...
```

```
import sound

sound.effects.echo.echofilter(a, b)
```

```
sound/
        __init__.py
        formats/
                __init__.py
                wavread.py
                wavwrite.py
                aiffread.py
                aiffwrite.py
                auread.py
                auwrite.py
                ...
        effects/
                __init__.py
                echo.py
                surround.py
                reverse.py
                ...
        filters/
                __init__.py
                equalizer.py
                vocoder.py
                karaoke.py
                ...
```

```
import sound

sound.effects.echo.echofilter(a, b)
```

```
from sound.effects import echo

echo.echofilter(a, b)
```

```
sound/
        __init__.py
        formats/
                __init__.py
                wavread.py
                wavwrite.py
                aiffread.py
                aiffwrite.py
                auread.py
                auwrite.py
                ...
        effects/
                __init__.py
                echo.py
                surround.py
                reverse.py
                ...
        filters/
                __init__.py
                equalizer.py
                vocoder.py
                karaoke.py
                ...
```

```
import sound

sound.effects.echo.echofilter(a, b)
```

```
from sound.effects import echo

echo.echofilter(a, b)
```

```
from sound.effects.echo import echofilter

echofilter(a, b)
```

Import Conventions

- Import statements go at the top of the file
- Preferred to import the whole module instead of the “from module import function” notation

pickle

pickle

Method	Use
`pickle.dump(obj, file)`	Write pickled representation of arbitrary object obj into the file object file.
`pickle.dumps(obj)`	Return pickled representation of arbitrary object obj as a bytes string.
`pickle.load(file)`	Read the object which has been pickled into `file` into memory, return the object.
`pickle.loads()`	Execute the current statement, proceed to the next one.

Safety around pickle

PDB

pdb

Command	Use
`pdb.set_trace()`	Sets breakpoint; launches `pdb` when execution encounters the breakpoint.
`p` *`expression`* or `print` *`expression`*	Prints the value of *`expression`*.
`c` or `continue`	Continue execution from the current breakpoint.
`n` or `next`	Execute the current statement, proceed to the next one.
`s` or `step`	Execute the current statement, step into the function.
`b` *`line`* or `b` *`function`*	Sets a breakpoint at line number *`line`*, or at the beginning of *`function`*.

collections

collections

Command	Use
`collections.namedtuple`	Object that supports name-binding to elements of a tuple.
`collections.defaultdict`	Object that supports default dictionary values (to reduce need for error catching when working with dictionaries).
`collections.Counter`	Simplify counting of elements in a collection.

functools

functools

Command	Use
`functools.cache`	Caches the output of a function (like the decorator we wrote in the FP lecture).
`functools.wraps`	Updates a wrapper function to look like the wrapped function. (`__name__`, `__doc__`, etc.)
`functools.partial`	Returns a function object with some positional and keyword arguments pre-filled.

`__doc__`, `__name__`

functools

Command	Use
`functools.cache`	Caches the output of a function (like the decorator we wrote in the FP lecture).
`functools.wraps`	Updates a wrapper function to look like the wrapped function. (`__name__`, `__doc__`, etc.)
`functools.partial`	Returns a function object with some positional and keyword arguments pre-filled.

Casting strings to ints

functools

Command	Use
`functools.cache`	Caches the output of a function (like the decorator we wrote in the FP lecture).
`functools.wraps`	Updates a wrapper function to look like the wrapped function. (`__name__`, `__doc__`, etc.)
`functools.partial`	Returns a function object with some positional and keyword arguments pre-filled.

itertools

itertools

Command	Use
`itertools.cycle(iterable)`	Returns an infinite iterable, cycling through the elements of `iterable`.
`itertools.count(start [,step])`	Returns an infinite iterable, counting up from `start` in increments of `step`.

threading

What is a thread?

Multithreading: multiple concurrent flows of program execution

Multithreading: multiple concurrent flows of program execution

Keep in mind this is different then multiprocessing!!!!!!!

Why would you want to run multiple programs at once?

Re

Character	Brief Description	Simple Example
.	any character (wildcard)	w..kly
^	Begins with	^where
[ ]	character set	[0-9]
|	either or	hi|bye
+	Once or more	me+
*	zero occurances or more	me*
{}	exact number of times	I{4}
\	special character operations	\w
$	Ends with	$bar

Command	Use
`re.findall(pattern, string)`	Returns a list of all non-overlapping substrings of `string` that match `pattern`.
`re.match(pattern, string)`	Returns `MatchObject` (allows for easy match processing) if string matches pattern, `None` otherwise.

"\(\d{3}\)[-]\d{3}-\d{4}"

(123) 456-7890