Python Tools & Techniques

Packing / Unpacking

Unpacking

Can “unpack” iterables:

```
point = [10, 20]

x, y = point
```

```
a = 2
b = 3

b, a = (a, b)
```

Can gather rest using * before variable:

```
letters = ["a", "b", "c"]

first, *rest = letters
```

Spread

Can “spread” iterables:

```
fruits = {"apple", "berry", "cherry"}

foods = ["kale", "celery", *fruits]
```

Error Handling

Errors

In general, Python raises errors in places JS returns undefined:

- provide too few/too many arguments to a function
- index a list beyond length of list
- retrieve item from dictionary that doesn’t exist
- use missing attribute on an instance

- conversion failures (eg, converting “hello” to an int)
- division by zero
- and more!

In general, in Python: **explicit is better than implicit**

Catching Errors

```
# try to convert this to a number

try:
    age = int(data_we_received)
    print("You are", age)

except:
    print("Hey, you, that's not an age!")

# next line is run either way
```

It’s risky, though, to just say ***except*** — that catches *all* errors!

```
data_we_received = "42"

try:
    age = int(data_we_received)
    print("You are", Age)

except:
    print("Hey, you, that's not an age!")
```

Better to catch the specific error you’re looking for:

```
age_we_received = "42"

try:
    age = int(data_we_received)
    print("You are", Age)

except ValueError:
    print("Hey, you, that's not an age!")
```

Common Exception Types

AttributeError	Couldn’t find attr: `o.missing`
KeyError	Couldn’t find key: `d["missing"]`

IndexError	Couldn't find index: `lst[99]`
NameError	Couldn't find variable: `not_spelled_right`
OSError	Operating system error: can't read/write file, etc
ValueError	Incorrect value (tried to convert "hello" to int, etc)

Raising Errors

In Python, it's common for you to "raise" errors to signal problems:

```
if color not in {"red", "green", "blue"}:
    raise ValueError("Not a valid color!")
```

Error Handling Pattern

Raise exception when you know it should be an error Handle it at the point you can give good feedback

```
def bounded_avg(nums):
    "Return avg of nums (makes sure nums are 1-100)"

    for n in nums:
        if n < 1 or n > 100:
            raise ValueError("Outside bounds of 1-100")

    return sum(nums) / len(nums)

def handle_data():
    "Process data from database"

    ages = get_ages(from_my_db)

    try:
        avg = bounded_avg(ages)
        print("Average was", avg)

    except ValueError as exc:
        # exc is exception object -- you can examine it!
        print("Invalid age in list of ages")
```

Docstrings & Doctests

Docstrings

Docstrings are the strings at top of function or file that document it:

```
def bounded_avg(nums):
    "Return avg of nums (makes sure nums are 1-100)"

    for n in nums:
        if n < 1 or n > 100:
            raise ValueError("Outside bounds of 1-100")

    return sum(nums) / len(nums)
```

It's incredibly good style for every function to have one!

Doctests

Doctests are snippets of interactive demonstration in a docstring:

```
def bounded_avg(nums):
    """Return avg of nums (makes sure nums are 1-100)

       >>> bounded_avg([1, 2, 3])
       2

       >>> bounded_avg([1, 2, 101])
       Traceback (most recent call last):
           ...
       ValueError: Outside bounds of 1-100
    """

    for n in nums:
        if n < 1 or n > 100:
            raise ValueError("Outside bounds of 1-100")

    return sum(nums) / len(nums)
```

Can run this test:

```
$ python3 -m doctest -v your-file.py
```

(use the ***doctest*** module, verbosely showing tests found & run)

Doctests are **awesome**

Testable documentation and *readable tests*.

Importing

Python includes a "standard library" of dozens of useful modules.

These are not in the namespace of your script automatically.

You have to *import* them

choice(seq) is a useful function: given a sequence, it returns a random item

```
from random import choice

print("Let's play", choice(games))
```

"From ***random***, pull in ***choice*** function as ***choice***"

```
# can pull in several things from a place

from random import choice, randint


# can change the local name of it

from random import choice as pick_a_thing

pick_a_thing(games)
```

Sometimes, it helpful to pull in the *library itself*:

```
import random

# now, we have the obj `random`, with all the funcs/classes
# within available to us

random.choice(games)
```

Exporting/Importing Your Code

score.py

```
def get_high_score():
    ...

def save_high_score():
    ...
```

game.py

```
from score import get_high_score

high = get_high_score()
```

(unlike JS, nothing needed to "export")

Installing Libraries

Python includes dozens of useful libraries

There are over 100,000 additional available ones :)

Using Pip

To install a new package:

```
$ pip3 install forex_python
... pip output here...

$ ipython
In [1]: from forex_python.converter import convert
In [2]: convert("USD", "GBP", 10)
7.6543
```

Virtual Environments

Normally, ***pip*** makes the installed library available everywhere

This is convenient, but a little messy:

- you might not need it for every project
- you might want to more explicitly keep track of which libraries a project needs
- you might want a new version of a library for one project, but not another

Python can help us by using a “virtual environment”

Creating a Virtual Environment

```
$ cd my-project-directory
$ python3 -m venv venv
```

(“using ***venv*** module, make a folder, ***venv***, with all the needed stuff”)

That makes the virtual environment folder — but you’re not *using it* yet!

Using Your Virtual Environment

```
$ source venv/bin/activate
(venv) $   <-- notice shell prompt!
```

- You only need to **create** the virtual environment once

- You need to use ***source*** every time you open a new terminal window

What does it mean to be "using" a virtual environment?

- It makes certain ***python*** is the version of Python used to create the venv
- You have access to the standard library of Python
- You **don't** have access to globally installed pip stuff
- You get to explicitly install what you want — and it will be only for here!

Installing into Virtual Environment

- Make sure you're using your venv — do you see it in your prompt?
- Use ***pip install***, as usual
  - But now it's downloaded & installed into that ***venv*** folder
  - It won't be available/confuse global Python or other venvs — tidy!

Tracking Required Libraries

To see a list of installed libraries in a venv:

```
$ pip3 freeze
... list of installed things...
```

It's helpful to save this info in a file (typically named "requirements.txt"):

```
$ pip3 freeze > requirements.txt
```

Using Virtual Environments

- Virtual environments are large & full of stuff you didn't write yourself
- You **don't want this to get into git / Github**
- So, add `venv/` to your project's ***.gitignore***
  - Use ***git status*** to make sure it's being ignored

Recreating a Virtual Environment

When using a new Python project:

```
$ git clone http://path-to-project
$ cd that-project
$ python3 -m venv venv
```

Then, as usual when working with a venv:

```
$ source venv/bin/activate
(venv) $ pip3 install -r requirements.txt
... pip output here ...
```

Leaving Virtual Environments

Use the ***deactivate*** shell command to leave the virtual environment:

```
$ source venv/bin/activate
(venv) $ deactivate
$ ... back to regular terminal ...
```

Files

You can open an on-disk file with ***open(filepath, mode)***

- ***filepath***: absolute or relative path to file
- ***mode***: string of how to open file (eg, “r” for reading or “w” for writing)

This returns an file-type instance.

Reading

Line-by-line:

```
file = open("/my/file.txt")

for line in file:
    print("line =", line)

file.close()
```

All at once:

```
file = open("/my/file.txt")

text = file.read()

file.close()
```

Writing

```
file = open("/my/file.txt", "w")

file.write("This is a new line.")
file.write("So is this.")

file.close()
```

Note: “with” blocks

Python has an intermediate bit of syntax called a “with block”.

For example:

```
with open("/my/file.txt", "r") as file:
    for line in file:
        print("line=", line)

    # our file is still open here

# but it will be automagically closed here
```

Python will keep that file open as long as you’re inside the with block. At the point the your code is no longer indented inside that block, it will automatically close the file you’ve used.

These with-blocks are used for other kinds of resources besides files; to learn more about them, you can search for “python context mananagers”.