A Crash Course in Python

Lecture 2

Centre for Data Science, ITER
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.

Contents

Modules

- Certain features of Python are not loaded by default.
- A standard convention for visualizing data with matplotlib is:

```
import matplotlib.pyplot as plt
plt.plot(...)
```

- For few specific values from a module, import explicitly and use them without qualification. For example:

```
from collections import defaultdict, Counter
lookup = defaultdict(int)
my_counter = Counter()
```

Functions

- A function is a rule for taking zero or more inputs and returning a corresponding output.
- Python functions are first-class, which means that we can assign them to variables and pass them into functions just like any other arguments.
- It is also easy to create short anonymous functions, or lambdas.
- Function parameters can also be given default arguments.

Strings

- Strings can be delimited by single or double quotation marks.
- Python uses backslashes to encode special characters.
- Multiline strings can be created using three double quotes.
- The **f-string**: It provides a simple way to substitute values into strings.
- Adding first name and last name of a person using f-string.

```
first_name = "Joel"
last_name = "Grus"
full_name = f"{first_name} {last_name”}
```

Exceptions

- When something goes wrong, Python raises an exception.
- Unhandled, exceptions will cause your program to crash.
- These can be handled them using try and except:
  - Example:

```
try:
   print(0 / 0)
except ZeroDivisionError:
   print("cannot divide by zero")
```

Lists

- The most fundamental data structure in Python is the **list**, which is simply an **ordered collection**.
- Square brackets are used to slice lists. The slice **i:j** means all elements from i (inclusive) to j (not inclusive).
- Python has an **in** operator to check for list membership.
- **extend** is used to add items from another collection in list.
- A common idiom is to use an **underscore** for a value that going to throw away:

```
""" now y == 2, didn't care about the first
element """
_, y = [1, 2]
```

Tuples

- Tuples are lists' immutable cousins.
- Tuples are a convenient way to return multiple values from functions.
- Tuples (and lists) can also be used for multiple assignment.

Dictionaries

- Another fundamental data structure is a dictionary, which associates values with keys and allows you to quickly retrieve the value corresponding to a given key.
- The value for a key can be retrieve using square brackets.
- A **KeyError** is raised if a key is used which is not in the dictionary.
- The existence of a key can be check using **in**.
- Dictionaries have a **get method** that returns a default value (instead of raising an exception) when a key is used which is not in the dictionary.
- Dictionary keys must be "**hashable**"; in particular, lists can not be used as keys.

Defaultdict

- A **defaultdict** is like a regular dictionary, except that when you try to look up a key it doesn't contain, it first adds a value for it using a zeroargument function you provided when you created it.
- In order to use defaultdicts, need to import them from collections.
  - The syntax is:

```
from collections import defaultdict
word_counts = defaultdict(int)
for word in document:
  word_counts[word] += 1
```

- These will be useful when we're using dictionaries to "**collect**" results by some key and don't want to have to check every time to see if the key exists yet.

Set

- Another useful data structure is set, which represents a collection of distinct elements.
- A set can be defined by listing its elements between curly braces.
- Empty sets defined by **set()** itself not $\{\}$, as $\{\}$ already means "empty dict."
- **in** is a very fast operation on sets.
- Set used to find the distinct items in a collection.

Counters

- A Counter turns a sequence of values into a defaultdict(int)-like object mapping keys to counts.
- This gives us a very simple way to solve our word_counts problem.
- A Counter instance has a **most_common** method that is frequently useful.
  - Syntax:

```
"""Print the 10 most common words and their
counts"""
for word, count in word_counts.most_common(10):
  print(word, count)
```

Truthiness

- Booleans in Python work as in most other languages, except that they're **capitalized**.
- Python uses the value **None** to indicate a **nonexistent** value. It is similar to other languages' **null**.
- Python has an **all** function, which takes an iterable and returns True precisely when every element is truthy, and an any function, which returns True when at least one element is truthy.

```
all([True, 1, 3]) """True, all are truthy """
all([True, 1, {}]) """False, {} is falsy"""
any([True, 1, {}]) """True, True is truthy"""
all([]) """True, no falsy elements in the list"""
any([]) """False, no truthy elements in the
list"""
```

List Comprehensions

- To transform a list into another list by choosing only certain elements, by transforming elements, or both can be done using list comprehensions.

```
even_numbers = [x for x in range(5) if x % 2 == 0]
""" [0, 2, 4] """
squares = [x * x for x in range(5)]
""" [0, 1, 4, 9, 16] """
even_squares = [x * x for x in even_numbers]
""" [0, 4, 16] """
```

- If we don't need the value from the list, it's common to use an underscore as the variable.

```
zeros = [0 for _ in even_numbers]
"""has the same length as even_numbers """
```

Automated Testing and assert

- There are elaborate frameworks for writing and running tests which will raise an AssertionError if the specified condition is not truthy.
- Assert a functions which is defined to check it is doing for what it is written. For example:

```
def smallest_item(xs):
   return min(xs)
assert smallest_item([10, 20, 5, 40]) == 5
```

- Another less common use is to assert things about inputs to functions. For example:

```
def smallest_item(xs):
  assert xs, "empty list has no smallest item"
  return min(xs)
```

Generators

- **Generators** are created with functions and the **yield** operator. For example:

```
def generate_range(n):
  i = 0
  while i < n:
    yield i
    i += 1
```

- The following loop will consume the **yielded** values one at a time until none are left:

```
for i in generate_range(10):
  print(f"i:{i}")
```

- **for comprehensions** wrapped in parentheses is also used to generate **generators**.

```
evens_below_20 = (i for i in generate_range(20) if
i % 2 == 0)
```

- An **enumerate function** turns values into pairs, i.e. (index,value).

Randomness

- To generate random numbers need to use **random** module.
- The random module actually produces pseudorandom (that is, deterministic) numbers based on an internal state that can be set with **random.seed** if you want to get reproducible results.
- **random.randrange** takes either one or two arguments and returns an element chosen randomly from the corresponding range.

Randomness Cont...

- **random.shuffle** randomly reorders the elements of a list.
- For randomly choosing a sample of elements without replacement (i.e., with no duplicates), **random.sample** can be used.
- To choose a sample of elements with replacement (i.e., allowing duplicates), the multiple calls to **random.choice** is useful for this.

Regular Expressions

- Regular expressions provide a way of searching text.

```
1  import re
2  re_examples = [ """ All of these are True, because """
3      not re.match("a", "cat"), """ 'cat' doesn't start with
       'a' """
4      re.search("a", "cat"), """ 'cat' has an 'a' in it """
5      not re.search("c", "dog"), """ 'dog' doesn't have a 'c'
        in it."""
6      3 == len(re.split("[ab]", "carbs")), """ Split on a or
       b to"""
7      ['c','r','s'].
8      "R-D-" == re.sub("[0-9]", "-", "R2D2") """ Replace
       digits with dashes."""
9      ]
10 assert all(re_examples), "all the regex examples should be
       True"
```

Zip and Argument Unpacking

- To zip two or more iterables together the **zip** function transforms multiple iterables into a single iterable of tuples of corresponding function.
- If the lists are different lengths, zip stops as soon as the first list ends.
- "**unzip**" a list is can be achieve using following syntax:

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
```

- The asterisk (*) performs argument unpacking, which uses the elements of pairs as individual arguments to zip.

args and kwargs

- In python one function can take another function as its argument but this does not work when function which is passed have two arguments(in this case TypeError will occur).
- This can be achieve by argument unpacking. For example:

```
def magic(*args, **kwargs):
  print("unnamed args:", args)
  print("keyword args:", kwargs)
  magic(1, 2, key="word", key2="word2")
"""Output:
unnamed args:  (1, 2)
keyword args:  {'key':  'word', 'key2':
'word2'}"""
```

where **`args`** is a tuple of its **unnamed arguments** and **`kwargs`** is a dict of its **named arguments**.

args and kwargs cont...

```
1  def f2(x, y):
2      return x + y
3
4  def doubler_correct(f):
5      """works no matter what kind of inputs f expects"""
6      def g(*args, **kwargs):
7          """whatever arguments g is supplied, pass them through
       to f"""
8          return 2 * f(*args, **kwargs)
9      return g
10
11 g = doubler_correct(f2)
12 assert g(1, 2) == 6, "doubler should work now"
```

Type Annotations

- Python is a dynamically typed language.
- In a statically typed language our functions and objects would have specific types and known as **type annotations**. That is,

```
def add(a:  int, b:  int) -> int:
  return a + b
print(add(10, 5))"""you'd like this to be OK"""
print(add("hi", "there")) """you'd like this to
be not OK"""
```

Type Annotations Cont...

- Reasons to use type annotations in your Python code are:
  - Types are an important form of documentation.
  - There are external tools (the most popular is mypy) that will read the code, inspect the type annotations, and let you know about type errors before you ever run your code.
    For the above example it will show the error:

    ```
    error:  Argument 1 to "add" has incompatible type
    "str"; expected "int"
    ```

  - Having to think about the types in your code forces you to design cleaner functions and interfaces.
  - Using types allows your editor to help you with things like **autocomplete**.

How to Write Type Annotations

- The **typing** module provides a number of parameterized types that we can be used for type annotations. For Example:

```
from typing import List """ note capital L """
def total(xs:  List[float]) -> float:
   return sum(total)
```

- For variables type annotations is:

```
x:  int = 5
```

How to Write Type Annotations Cont...

```
1  from typing import Callable
2
3  '''The type hint says that repeater is a function that takes
       two arguments, a string and an int, and returns a string.''
       '
4
5  def twice(repeater:Callable[[str, int], str], s:str)->str:
6      return repeater(s, 2)
7
8  def comma_repeater(s: str, n: int) -> str:
9      n_copies = [s for _ in range(n)]
10     return ', '.join(n_copies)
11
12 assert twice(comma_repeater, "type hints") == "type hints, type
       hints"
```

References

[1] Data Science from Scratch Joel Grus, Shroff/O'reilly, Second Edition

Thank You
Any Questions?