CIS 1100

Announcements

Python
Fall 2024
University of Pennsylvania

Announcements

This lecture contains less "testable material" than normal. What you might need for an exam:

- Being able to **read** type annotations in signatures
- Being able to quickly glean the purpose & usage of a given function or library given friendly documentation

CIS 1100 TA Application went live on Friday

- Due Friday, Nov 22 at 11:59
- Includes both a common and a supplemental application, so make sure to fill out both
- TA Application Panel to ask about the TA experience tonight! from 7-8:30pm in Wu & Chen Auditorium (Levine 101)

CIS 1100

Basics of Typing

Python
Fall 2024
University of Pennsylvania

Mark "A" if this snippet runs without error; mark "B" if this snippet leads to an error.

(M1)

```
"hello" + 4
```

(M2)

```
"hello" + "4"
```

(M3)

```
["never", "let" , "it", "be"] + "said"
```

(M4)

```
["never", "let" , "it", "be"] + ["said"]
```

(M5)

```
x = 4
x = "4"
```

Types in Python

Python expressions always produce **values** that belong to a certain **type**. There are several design considerations about Python that make it interesting to work with.

- Python is **strongly typed**: operations that are not defined for a datatype are not permitted.

```
"hello" + 4 # no good—leads to an error
```

- Python is **dynamically typed**: variables can change their types while the program is running.

```
x = 4
x = "4" # not a problem!
```

Types in Java

Java is a **strongly** and **statically** typed language that is **compiled**.

- **Compiled:** the code is translated from Java to a different representation before it can be run
- Strong:

```
System.out.println(4 + true); // program won't compile!
```

- **Static:** variable types are determined when the program is compiled and cannot change while the program is running.

```
int x = 4;
x = "four"; // leads to a compiler error!
```

Static Type Systems

Other languages (like Java) require you to manually identify the types of variables & functions inside of the code.

- Annoying:
  - You have to commit to the types you pick
  - Lots of extra stuff laying around in your code
- Helpful:
  - The compiler checks for you before the program runs that all of the values will take the shapes they're supposed to.
  - The stricter the compiler, the more likely your program is to be bug free once you get it to run.

Static Typing in Java

First, a function in Python:

```
def count_occurrences_in_sequence(seq, target):
    count = 0
    for elem in seq:
        if target == elem:
            count += 1
    return count
```

Static Typing in Java

The same function in Java:

```
public static int countOccurrencesInSequence(String[] seq, String target) {
    int count = 0;
    for (String elem : seq) {
        if (target.equals(elem)) {
            count += 1;
        }
    }
    return count;
}
```

- Local variables declared with a type
- Function return type is noted
- Each input argument is given a type

The Joys and "Joys" of Python

Joys:

- Python is fast to write
- Python is FUN to write
- Python can be **quick** and dirty—and that's valid!

"Joys":

- Python can be arcane and hard to read
- Python functions need to defend against inputs of unpredictable types
- Python can be quick and **dirty**—and that's stressful!

Goals

From PEP 484:

> It should also be emphasized that Python will remain a dynamically typed language, and the authors have no desire to ever make type hints mandatory, even by convention.

But:

- Can we make Python code do a better job of explaining itself without comments?
- Can we decide to *sometimes* enforce some amount of static type checking?

CIS 1100

Variable Type Annotations

Python
Fall 2024
University of Pennsylvania

Type Annotations for Code Legibility

- Good variable & function names help a reader interpret the meaning of a snippet of code.
- Adding a note for the type of a variable/function can help that become all the more clear.

Type Annotations for Variables and Functions

For variables, `: type` annotates the variable with the type of the value it's supposed to store. For functions, `: type` annotates the input types and `-> ret_type` annotates the intended return type.

```
my_variable : type = <some_value>

def my_func(arg1 : type1, arg2 : type2) -> ret_type:
    ...
```

Annotating Primitives

Examples from `mypy` documentation.

```
x: int = 1
x: float = 1.0
x: bool = True
x: str = "test"
```

(These are somewhat obvious cases— *"why bother when the type can be inferred from the value?"*—but at least they help signal a commitment to maintain that type for the variable)

Annotating Sequences & Collections

Examples from `mypy` documentation.

For collections, we specify that it's a `collection` of `type` values by writing some form of `collection[type]`

```
x: list[int] = [1]
x: set[int] = {6, 7}
```

For a `dict`, which maps keys to values, we can identify the types of the keys and the values.

```
x: dict[str, float] = {"field": 2.0}
```

Annotating Tuples

Examples from `mypy` documentation.

Two common uses for tuples are

- *fixed-length containers of mixed types*

```
x: tuple[int, str, float] = (3, "yes", 7.5)
```

- *variable-length containers for a single type*

```
x: tuple[int, ...] = (1, 2, 3)
```

Annotating Mixed Types

Perhaps a variable stores an `int` unless it stores `None`, indicating a missing value. We can use the `|` operator to indicate this:

```
excused_absences: int | None = query_student_absences(student_name)
```

Annotating Mixed Types

Perhaps a list stores a bunch of `int` values or `None` values, indicating places where the value is missing. We can use the `|` operator to indicate this:

```
all_excused_absences: list[int | None] = [query_student_absences(name) for name in student_names]
```

CIS 1100

Function Type Annotations

Python
Fall 2024
University of Pennsylvania

Function Type Annotations

For functions, `: type` annotates the input types and `-> ret_type` annotates the intended return type.

```
def my_func(arg1 : type1, arg2 : type2) -> ret_type:
    ...
```

Works for keyword arguments, too:

```
def my_func(pos_arg1 : type1, kwarg1 : type2 = <default>) -> ret_type:
    ...
```

Function Type Annotations

For functions, `: type` annotates the input types and `-> ret_type` annotates the intended return type.

```
def my_func(arg1 : type1, arg2 : type2) -> ret_type:
    ...
```

Works for keyword arguments, too:

```
def my_func(pos_arg1 : type1, kwarg1 : type2 = <default>) -> ret_type:
    ...
```

(S7)

- What are the input type(s) for this function?
- What type would it return?

```
def greet(name: str) -> str:
    ...
```

(L11)

- What are the input type(s) for this function?
- What type should it return?
- Based on the name and input types, can you make an educated guess at what a function like this would do?

```
def calculate_total(item_prices: list[float], sales_tax: float = 0.08) -> float:
    ...
```

Example: Caesar Cipher

Based on the docstring for this function from Caesar,

can we add type annotations to the function signature?

```
def string_to_symbol_list(message):
    """
    Description: converts a string to a symbol list, where each element of the
                 list is an integer encoding of the corresponding element of
                 the string.
    Input:  the message text (stored in a string) to be converted
    Output: the encoding of the message into a list of integers
    """
    ...
```

Example: Caesar Cipher

```
def string_to_symbol_list(message: str) -> list[int]:
    """
    Description: converts a string to a symbol list, where each element of the
                 list is an integer encoding of the corresponding element of
                 the string.
    Input:  the message text to be converted
    Output: the encoding of the messagex
    """
    ...
```

(C12)

Rewrite this function signature to feature type annotations. Try to be as specific as possible when annotating lists/sets/dicts, e.g. `list[int]` or `dict[float, float]`.

```
def ta_endorsements(restaurants, min_rating):
    """Given a dictionary mapping restaurant names to
    restaurant data, return a set of the names of all
    restaurants that exceed the given min_rating."""
    ...
```

Key Takeaways

Type annotations can save a little ambiguity when you're reading a function signature and figuring out how to use or implement it.

- They're just comments, basically, but they can save a little reading
- They're entirely optional, but you should use them as much as you like

CIS 1100

Static Type Checkers

Python
Fall 2024
University of Pennsylvania

Compilers

This is commonly done in other languages with a compiler. This can tell you when you made a mistake in your typing and prevent you from proceeding.

From Java:

```
Palindrome.java:13: error: bad operand types for binary operator '<='
        if (word <= 1) {
                 ^
  first type:  String
  second type: int
```

Static Type Checkers in Python

Short answer: automatic type enforcement just not that commonly done!

- Hard to get right in a language not designed for it
- Just use a different language, really. (Julia is super cool if you want Pythonish + static typing)

Static Type Checkers in Python

Longer answer: you can do it with MyPy.

We don't do this because:

- it doesn't actually stop you from running the poorly-typed code
- most libraries in Python aren't written with annotations and so MyPy complains CONSTANTLY.

CIS 1100

Reading Documentation
Tips & Tricks

Python
Fall 2024
University of Pennsylvania

Code is Hard to Read

So people write a lot of **documentation** for their code, including:

- docstrings/function header comments
- inline comments
- big long webpages explaining the use & purpose of their code

Unfortunately, a lot of documentation is also kind of hard to read.

A Guided Walkthrough of Some Examples, Featuring Some Tips for Getting Through Documentation Hell

Imagine you're taking on a part of a project where you have to implement some kind of spell checker. Given a word that you think is misspelled, how would you suggest a replacement?

A Reasonable Start...

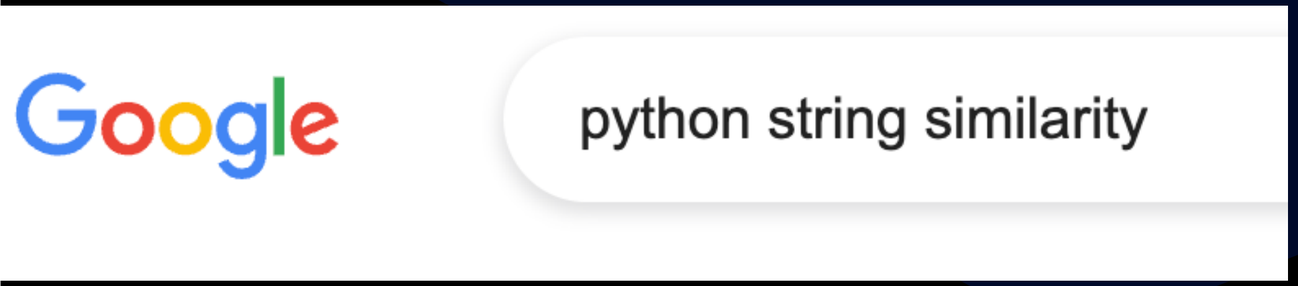
Just Google it!

(Disclaimer—**don't** do this while you're doing CIS 1100 homeworks, please!)

- We're (mostly) trying to give you practice with the fundamentals of Python that you need to be well-versed in if you want to be able to write code quickly and independently
- Google & AI searches for simple stuff lead to full answers which deprives you of the learning and also is plagiarism (and therefore Bad and Forbidden)
- As we reach the end of the course, we're setting time aside for some topics that will help you **after** you move on

Anyways...

You'd want to suggest a word that looks similar to the typo, so you're trying to find the word that's most similar from a known set of words.



Google

python string similarity

All Videos Images

Between two strings

Search Labs | AI

This is going to be 10% wrong > 50% of the time.

Python offers various libraries and techniques to calculate string similarity. Here's a breakdown of the most popular methods:

1. Levenshtein Distance:

- Measures the minimum number of edits (insertions, deletions, or substitutions) needed to transform one string into another.
- Lower distance indicates higher similarity.
- Implemented in the `Levenshtein` library:

Python

```python
from Levenshtein import distance

distance('kitten', 'sitting')  # Output: 3
```

Uhhh...

```
>>> from Levenshtein import distance
ModuleNotFoundError: No module named 'Levenshtein'
```

OK, so not built in. We could fix with `pip`, but then we need to go searching for more documentation. Leads to a case where...

- you want help with X
- you find solution Y, which needs some tweaking
- so you go to solution Z, which fixes Y and maybe makes absolutely no progress on X

Tip #1: Resist the Siren Song of AI Results!

The First Real Search Result

Stack Overflow
https://stackoverflow.com › questions › find-the-similar...

Find the similarity metric between two strings - python

How do I get the probability of a **string** being **similar** to another **string** in **Python**? I want to get a decimal value like 0.9 (meaning 90%) etc.

16 answers · Top answer: There is a built in. from difflib import SequenceMatcher def similar(a…

Compare **Similarity** of two strings - **python** - Stack Overflow	Feb 16, 2022
Similarity measure for Strings in **Python** - Stack Overflow	Nov 29, 2018
String similarity metrics in **Python** [duplicate] - Stack Overflow	Sep 24, 2009
Most efficient **string similarity** metric function - **python**	May 29, 2018

More results from stackoverflow.com

Not Exactly Right Either...?

Find the similarity metric between two strings

Asked 11 years, 4 months ago Modified 1 year, 3 months ago Viewed 492k times

506

How do I get the probability of a string being similar to another string in Python?

I want to get a decimal value like 0.9 (meaning 90%) etc. Preferably with standard Python and library.

e.g.

```
similar("Apple","Appel") #would have a high prob.

similar("Apple","Mango") #would have a lower prob.
```

python probability similarity metric

But wait...

16 Answers

There is a built in.

>800 upvotes?

Sorted by: Highest score (default)

894

There is a built in.

```
from difflib import SequenceMatcher

def similar(a, b):
    return SequenceMatcher(None, a, b).ratio()
```

Using it:

```
>>> similar("Apple","Appel")
0.8
>>> similar("Apple","Mango")
0.0
```

Share Improve this answer Follow

answered Jun 30, 2013 at 8:18

Inbar Rose

43.3k • 24 • 89 • 134

Answer + Example?

Author isn't being a jerk?

Tip #2: Seek Out Well-Vetted Answers!

Tip #3: Embrace the Python Built-Ins!

A Good Lead

This is not technically a solution to our question, but it is a promising lead to a built-in.

```
from difflib import SequenceMatcher

def similar(a, b):
    return SequenceMatcher(None, a, b).ratio()
```

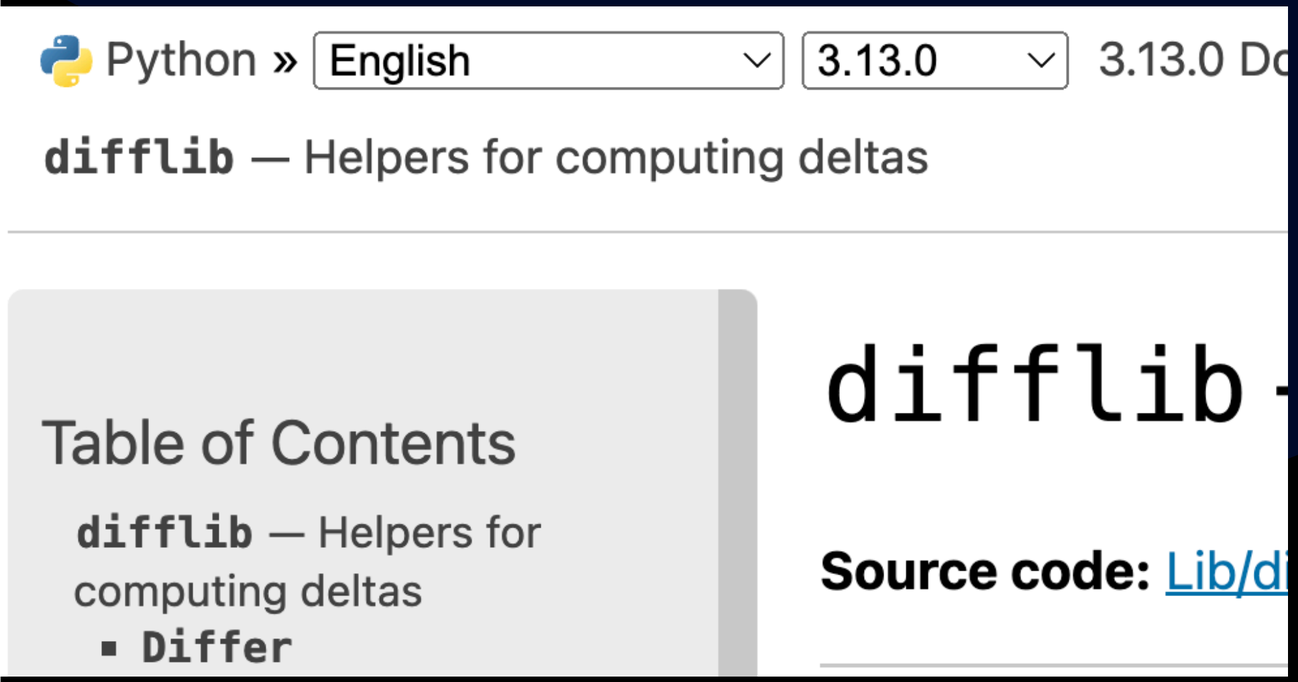
Understanding Python Module Documentation

We can find the `difflib` module in the Python documentation.

But now we have to hope to understand this nonsense!

Steps for Success:

1. Start with the title and intro paragraph
2. Then, check the table of contents for something you think you might care about
3. If something seems promising, navigate to it from the T.O.C.
4. If nothing seems promising CTRL-F for important keywords
5. Don't be afraid to **read**, but don't default to reading straight through...



From the header:

> This module provides classes and functions for comparing sequences. It can be used for example, for comparing files, and can produce information about file differences in various formats, including HTML and context and unified diffs.

Okayyy...

Table of Contents

«

Table of Contents

(S8) Which of these function/class names seem most promising for picking the best replacements for typo'd words based on similar spellings?

So Close, Yet So Far...

(or, *"Good lord, that's a lot of text..."*)

`difflib.get_close_matches(word, possibilities, n=3, cutoff=0.6)`

Return a list of the best "good enough" matches. *word* is a sequence for which close matches are desired (typically a string), and *possibilities* is a list of sequences against which to match *word* (typically a list of strings).

Optional argument *n* (default `3`) is the maximum number of close matches to return; *n* must be greater than `0`.

Optional argument *cutoff* (default `0.6`) is a float in the range [0, 1]. Possibilities that don't score at least that similar to *word* are ignored.

The best (no more than *n*) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
```

Deep Breaths

Start with the signature:

```
difflib.get_close_matches(word, possibilities, n=3, cutoff=0.6)
```

1. Ignore the keyword arguments at the start because they're **optional**
2. Decide: Do the names of the positional arguments seem helpful? Can you make an initial guess at their types?

Deep Breaths

Then, read just the first sentence:

```
difflib.get_close_matches(word, possibilities, n=3, cutoff=0.6)
```

Return a list of the best "good enough" matches. *word* is a sequence for which close matches are desired (typically a string), and *possibilities* is a list of sequences against which to match *word* (typically a list of strings).

This is going to be the text that:

- explains the purpose most succinctly
- describes the most important arguments

This seems promising!!

Deep Breaths

To verify, skip to the **examples!!!**

The text between the signature and the examples is a *perfidious trick.* It is designed to keep you *lost in a morass of petty details.* For a goal-oriented programmer like you, there's nothing in this zone but *trite nonsense.*

What's That!?

An example of a built-in that solves my EXACT problem?

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
```

That's Python, baby!!

Now, With Confidence In Our Mission...

(L13) Given a typo'd word `l` and a list of valid english words `english`, how could I write a line of code to select the "best" possibility to replace the typo?

```
difflib.get_close_matches(word, possibilities, n=3, cutoff=0.6)
```

Return a list of the best “good enough” matches. *word* is a sequence for which close matches are desired (typically a string), and *possibilities* is a list of sequences against which to match *word* (typically a list of strings).

Optional argument *n* (default `3`) is the maximum number of close matches to return; *n* must be greater than `0`.

Optional argument *cutoff* (default `0.6`) is a float in the range [0, 1]. Possibilities that don’t score at least that similar to *word* are ignored.

The best (no more than *n*) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

More Documentation Reading

```
>>> l = ["this", "is", 100, "percent", "important"]
>>> print("👏".join(l))
--------
```

str.join(*iterable*)

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` objects. The separator between elements is the string providing this method.

(S9) What would appear on the last line as a result of running this code?

Working With New Packages

If you need to solve a problem by installing a library that's not built-in, you'll have to install something.

- There are usually a number of options provided for how to do this
- There is usually **one good way** to do this

pandas

Getting started | User Guide | API reference | Development | Rele note

Installation

Package overview

Getting started tutorials

Comparison with other tools

Community tutorials

Getting started

Getting started

Installation

Getting started

Installation

Working with conda?

pandas is part of the Anaconda distribution and can be installed with Anaconda or Miniconda:

```
conda install -c conda-forge pandas
```

Prefer pip?

pandas can be installed via pip from PyPI.

```
pip install pandas
```

In-depth instructions?

Installing a specific version? Installing from source? Check the advanced installation page.

Learn more

This is a trap. Never do this unless you have good reason to do it.

Don't Fall Prey to These...

Install a nightly build

Matplotlib makes nightly development build wheels available on the scientific-python-nightly-wheels Anaconda Cloud organization. These wheels can be installed with `pip` by specifying scientific-python-nightly-wheels as the package index to query:

```
python -m pip install \
  --upgrade \
  --pre \
  --index-url https://pypi.anaconda.org/scientific-python-nightly-wheels/simple \
  --extra-index-url https://pypi.org/simple \
  matplotlib
```

You can also clone the repository using **git** and install from source:

```
git clone https://github.com/pyglet/pyglet.git
```

Tip #4: Most popular libraries worth using have simple pip installations