

# Data Structures + *TROUBLESHOOTING!*

---

BILD 62

## *From last week*

- There are multiple ways to write and run Python code
- Assigning variables & rules for variable names
- Types of objects we met: **integers**, **floats**, **strings**
- How to **concatenate** and **slice** strings
- Python syntax rules:
  - Spaces and white space do not matter
  - Indentation matters
- Functions we learned: `import`, `print`, `type`, `isinstance`, `float`, `int`, `str`

# Syntax Rules in Python

- Python is **case sensitive**: `letsroll`  $\neq$  `LetsRoll`
- White space does not matter (e.g., line 9 or 11 below)
- Indentation *does* matter — use **tab** to indent your code
- **Indexing** (we'll come back to this later)

- Python starts indexing at 0. So if you have a list of numbers:

```
list = [ 2 , 5 , 7 , 1 , 9 , 2 ]
```

and you ask for `list[1]`, you'll get 5.

## *From last class:* Rules for creating new variable names

**Variable names should only ever contain letters, numbers, and underscores.**

- Do not start with a number.
- No spaces in variable names.
- Variable names are **case-sensitive**.
- Names cannot be keywords (e.g., and, break, try).
- Names cannot contain symbols, including dashes ( - )

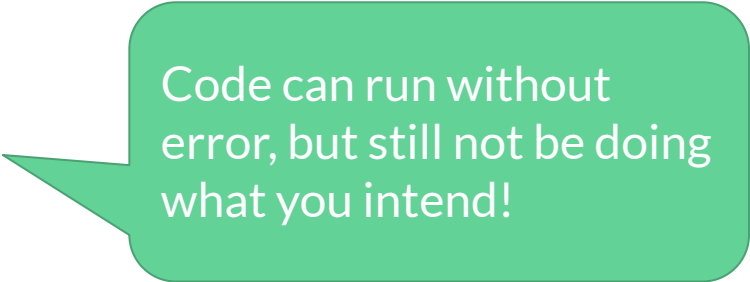


# Troubleshooting

# You'll encounter various types of errors

- **Syntax:** language rules broken
  - E.g., quotes missing, incorrect indentation
- **Runtime:** unable to execute
  - E.g., zero division error, or an unrecognized variable
- **Semantic/Logical:** unexpected output, e.g.:

```
>>> name = "Alice"  
>>> print("Hello name")  
>>> Hello name
```



Code can run without error, but still not be doing what you intend!

For a full list of possible errors:

<https://www.tutorialsteacher.com/python/error-types-in-python>

Use  
`print()`  
often!



**Mathieu Alain** @mathieualain@mastodon.social  
@miniapeur



People: What debugging tool you use ?

me: `print()`



[Original tweet](#)





Yani Bellini Saibene

@yabellini@fosstodon.org

7m\*

@grimalkina my @thecarpentries rubber duck and my Groot. They are very helpful 😊

I also love to be the rubber duck for my team mates or friends 🦹

[Translate](#)



# Rubber ducking!!!!

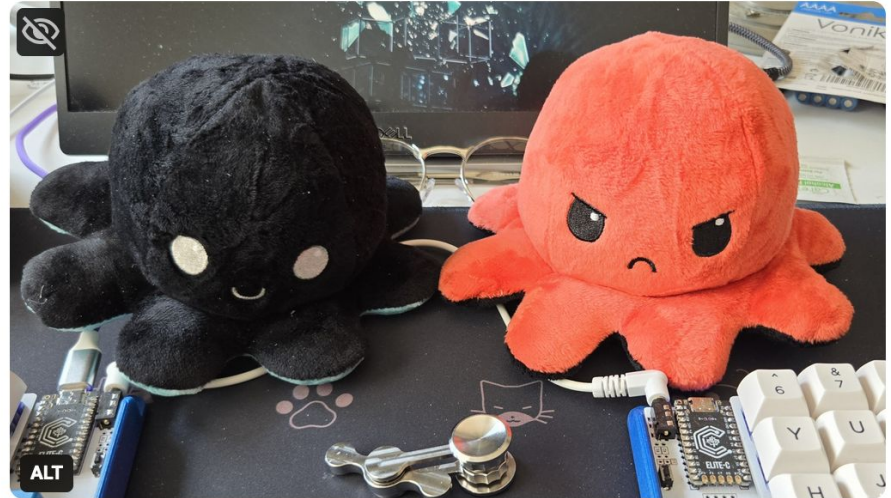


sungo

@sungo@anti.social.sungo.cloud

1h

@grimalkina Not always ducks 🦸



0





# Introducing: Stack Overflow

If you have a question about something, chances are at least a thousand other people on the internet had that same question.

(This is what genAI models are learning from!)

<https://stackoverflow.com/questions/1549801/what-are-the-differences-between-type-and-isinstance>

The screenshot shows a Stack Overflow question page. The header includes the Stack Overflow logo, navigation links for Products, Customers, and Use cases, and a search bar. The left sidebar contains links for Home, PUBLIC, Stack Overflow (selected), Tags, Users, Jobs, TEAMS, and What's this?. Below these is a badge for 'First 25 Users Free'. The main content area displays a question titled 'What are the differences between these two code fragments? Using `type()` :'. The question has 1178 votes and 446 answers. It shows two code snippets: one using `type()` and another using `isinstance()`. The question is tagged with 'python', 'oop', 'inheritance', and 'types'. It was edited by TylerH on Jan 4 '18 and asked by abbot on Oct 11 '09. The question has 7 answers, with the top one by TylerH having 1203 votes. The bottom of the page shows the start of an answer by TylerH summarizing the differences between `isinstance` and `type`.

stackoverflow Products Customers Use cases Search...

Home PUBLIC Stack Overflow Tags Users Jobs TEAMS What's this? First 25 Users Free

What are the differences between these two code fragments? Using `type()` :

1178

```
import types

if type(a) is types.DictType:
    do_something()
if type(b) in types.StringTypes:
    do_something_else()
```

Using `isinstance()` :

```
if isinstance(a, dict):
    do_something()
if isinstance(b, str) or isinstance(b, unicode):
    do_something_else()
```

python oop inheritance types

share improve this question edited Jan 4 '18 at 22:09 TylerH 17.2k 10 57 75 asked Oct 11 '09 at 3:50 abbot 22.5k 5 43 52

add a comment

7 Answers active oldest votes

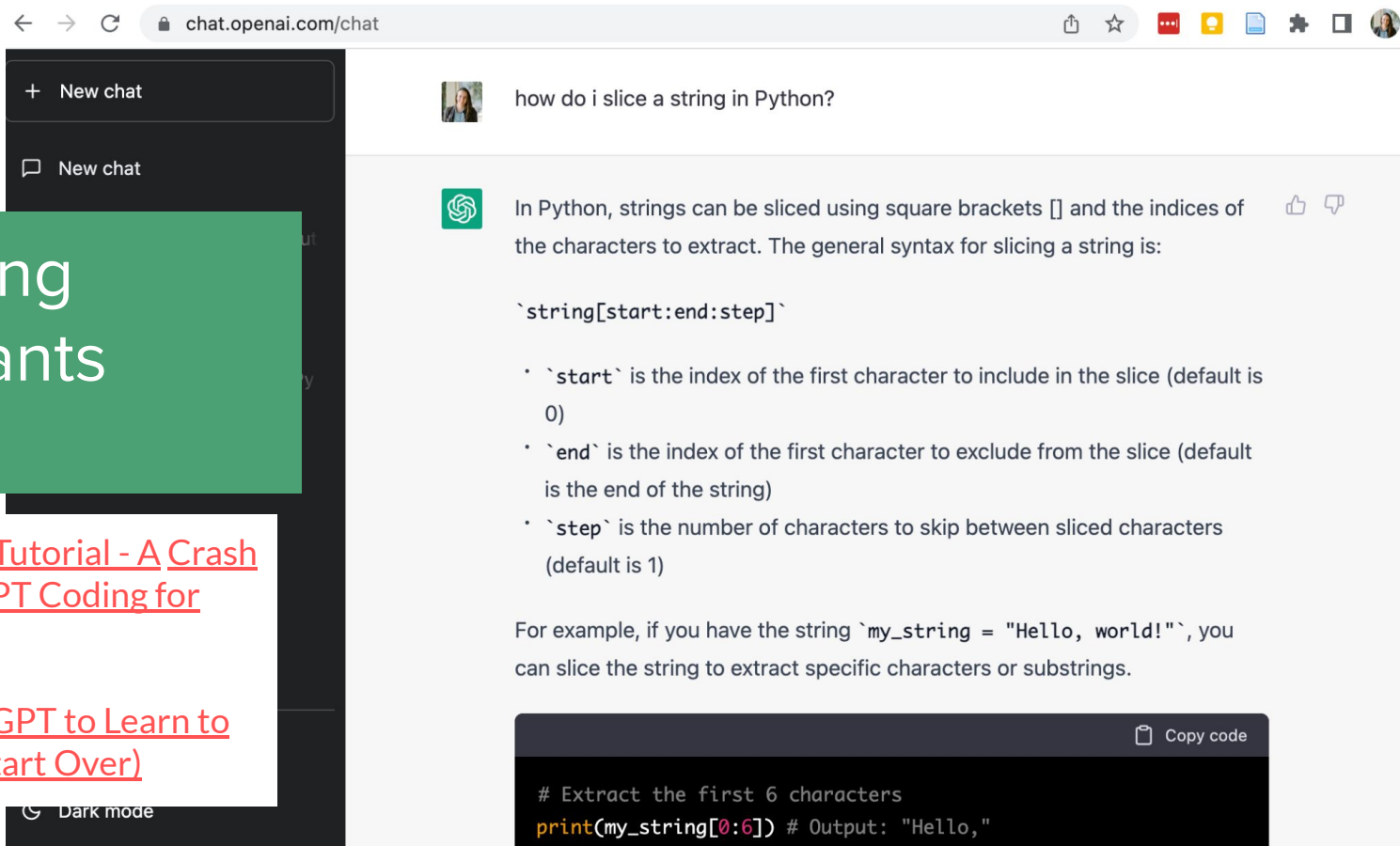
To summarize the contents of other (already good!) answers, `isinstance` caters for inheritance (an instance of a derived class is an instance of a base class, too), while checking for equality of `type` does not (it demands identity of types and rejects instances of subtypes, AKA subclasses).

1203

# Introducing AI assistants

[ChatGPT Coding Tutorial - A Crash Course on ChatGPT Coding for Beginners](#)

[How I'd Use ChatGPT to Learn to Code \(If I Could Start Over\)](#)



The screenshot shows the ChatGPT web interface in a browser. The address bar shows 'chat.openai.com/chat'. On the left, there's a sidebar with a 'New chat' button and a 'Dark mode' toggle. The main chat area shows a user's question: 'how do i slice a string in Python?'. Below it, the AI assistant's response is displayed, starting with the OpenAI logo. The response explains string slicing syntax and provides an example. At the bottom, there's a code block with Python code and a 'Copy code' button.

chat.openai.com/chat

+ New chat

New chat

Dark mode

how do i slice a string in Python?

In Python, strings can be sliced using square brackets [] and the indices of the characters to extract. The general syntax for slicing a string is:

```
`string[start:end:step]`
```

- ``start`` is the index of the first character to include in the slice (default is 0)
- ``end`` is the index of the first character to exclude from the slice (default is the end of the string)
- ``step`` is the number of characters to skip between sliced characters (default is 1)

For example, if you have the string ``my_string = "Hello, world!"``, you can slice the string to extract specific characters or substrings.

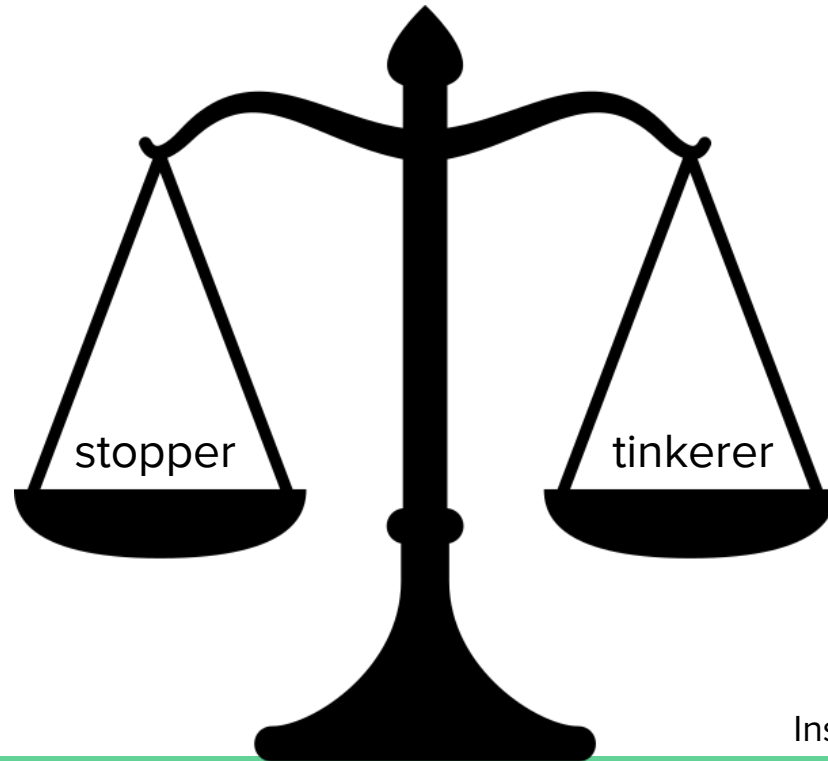
```
# Extract the first 6 characters
print(my_string[0:6]) # Output: "Hello,"
```

Copy code

# Possible levels of AI use

1. **No AI use** - Writing *all code from scratch* using only course materials, documentation, and your own understanding
2. **AI for understanding concepts** - Using AI to explain programming concepts, syntax, etc. (e.g., "Explain what a for loop does" or "What does this error mean?")
3. **AI for code explanation** - Pasting code examples (e.g., from class or online) into AI to understand how they work then writing your own implementation
4. **AI-assisted debugging** - Sharing your code with AI to identify bugs or logic errors, then making corrections yourself after understanding the issue
5. **AI for targeted code snippets** - Asking AI to generate small, specific pieces of code (a few lines) that you integrate into a larger solution
6. **AI for substantial code generation** - Having AI generate major portions of your solution (functions or large blocks), which you review and modify
7. **Full AI generation** - Having AI write complete solutions that you use with minimal modifications or understanding

Be a ***mover***: Make forward progress, & strike a balance between stopping & tinkering forever.



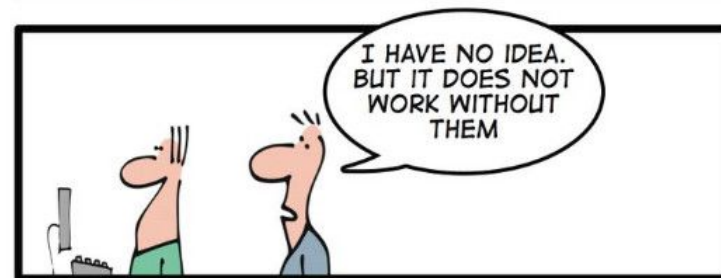
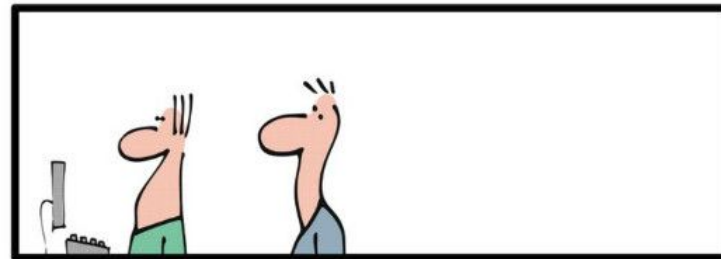
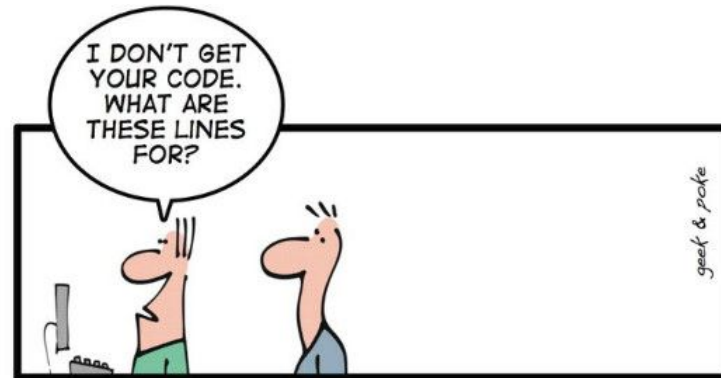
Inspired by Shannon Ellis (COGS18)

# Consider the two hour rule

- If you're stuck, work on a problem for **an hour**.
- If you're still stuck, take a **30 minute break**.
- Then, try again for **30 minutes**.
- If you're still stuck, seek out help.

# Where else can I get practice & help?

- Canvas Discussion Boards
- Office hours
- End-of-lecture resources
- GitHub: programmers' social media platform
  - for “**Issues**” related to specific codes/packages



Geek&Poke

<http://geek-and-poke.com/geekandpoke/2009/7/25/the-art-of-programming-part-2.html>

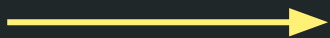
# Objectives for this session

- **Compare & contrast** the types of variables that Python uses to store data points
  - **Understand** the syntax for lists, tuples, and dictionaries
  - **Index, slice, cast, and mutate** lists
-



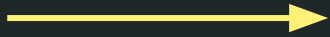
# We're learning how to deal with more and more complex data

```
data_point = 8.02
```



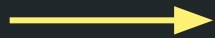
**single variable**  
(int, float,  
string)

```
data = [8.38, 3.34, 6.35]
```



**data structure**  
(list, tuple,  
dictionary)

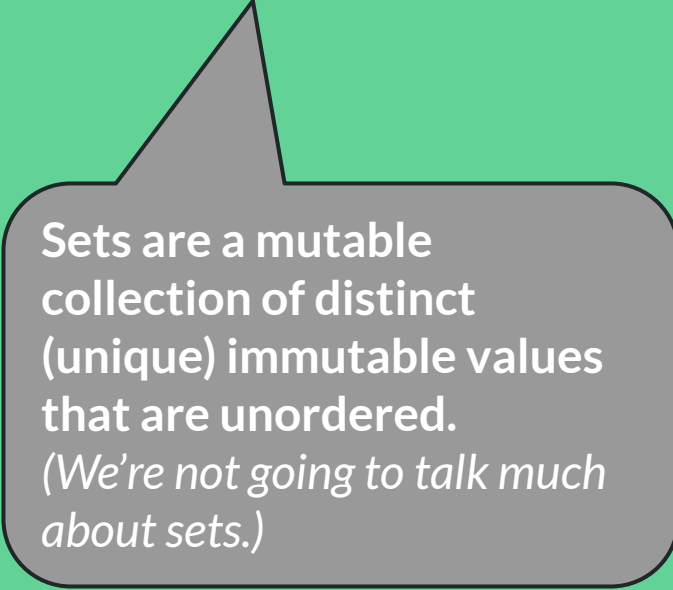
```
big_data = [data_1, data_2, ...]
```



**array**  
or **dataframe**

Python has different ways to store data:  
**lists, tuples, dictionaries, and sets.**

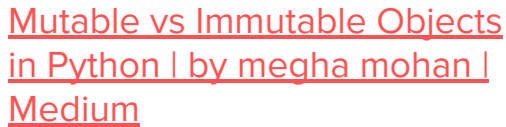
These differ in their  
**syntax, mutability,**  
and use cases.



Sets are a mutable collection of distinct (unique) immutable values that are unordered.  
*(We're not going to talk much about sets.)*

# Which objects are immutable?

Class	Description	Immutable?
<b>bool</b>	Boolean value	✓
<b>int</b>	integer (arbitrary magnitude)	✓
<b>float</b>	floating-point number	✓
<b>list</b>	mutable sequence of objects	
<b>tuple</b>	immutable sequence of objects	✓
<b>str</b>	character string	✓
<b>set</b>	unordered set of distinct objects	
<b>frozenset</b>	immutable form of set class	✓
<b>dict</b>	associative mapping (aka dictionary)	



# Why does Python count from 0?

Programming languages like Fortran, MATLAB and R start counting at 1 because that's what human beings have done for thousands of years!

Languages in the C family (including C++, Java, Perl, and Python) count from 0 because it represents an offset from the first value in the array (the second value is offset by one index from the first value).

This is closer to the way that computers represent arrays.

**Lists** are flexible & efficient containers for heterogeneous data

- Lists are **mutable**: we can change individual elements of the list
- Denoted by brackets & elements are separated by commas

```
my_list = ['apples', 'bananas', 'oranges']
```

Let's do this in the Jupyter Notebook!

- Check the length of your list by using `len(my_list)`
- Use `my_list.append()` to add elements to a list
- Remove elements by index using `del my_list[2]`
- Remove elements by value by using `my_list.remove('oranges')`
- Sort by using `my_list.sort()`

Corresponding notes are here for your reference.

# Lists are flexible & efficient containers for heterogeneous data

- Lists are **mutable**: we can change individual elements of the list
- Denoted by brackets & elements are separated by commas

```
my_list = ['apples', 'bananas', 'oranges']
```

- Check the length of your list by using **len(my\_list)**
- Use **my\_list.append()** to add elements to a list
- Remove elements by index using **del my\_list[2]**
- Remove elements by value by using **my\_list.remove('oranges')**
- Sort by using **my\_list.sort()**



# Indexing lists

```
my_list = [1,2,5,2,3]
```

```
my_list[1] = 2
```


Index number



```
my_list[-1] = 3
```

```
my_list[5] =
```

Allows you to count from the end  
(could be -2, etc.)



**IndexError**

Shown if you try to get an index  
that doesn't exist



# Slicing lists

`my_list[0:2]`

`my_list[1:3]`

`my_list[:3]`

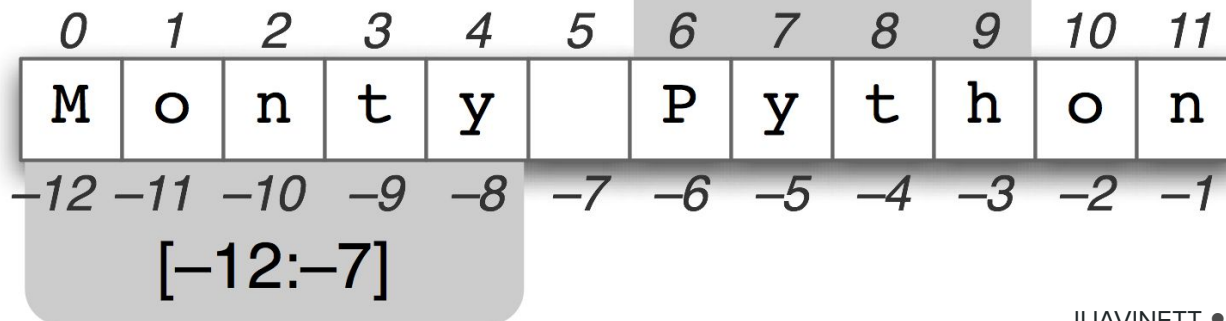
`my_list[3:]`

`my_list[:]`

**[included:excluded]**

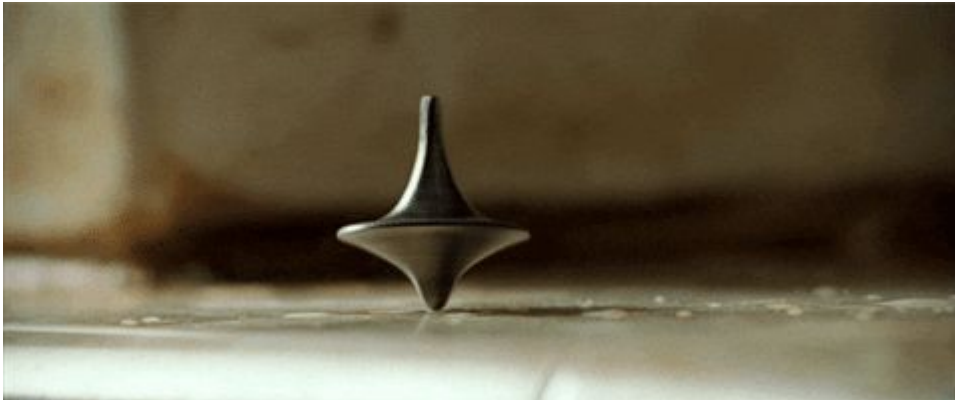
It doesn't show you the stop element (it shows you elements with indices 0 & 1)

One way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of n characters has index n.



# Lists of lists

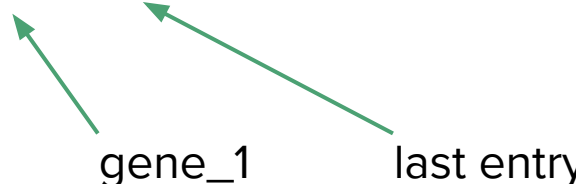
```
>>> gene_1 = ['gene1', 0.48, 0.55]
>>> gene_2 = ['gene2', 0.38, 0.85]
>>> gene_3 = ['gene3', 0.21, 0.81]
>>> all_genes = [gene_1, gene_2, gene_3]
>>> print(all_genes[0][-1])
```



# Lists of lists

```
>>> gene_1 = ['gene1', 0.48, 0.55]
>>> gene_2 = ['gene2', 0.38, 0.85]
>>> gene_3 = ['gene3', 0.21, 0.81]
>>> all_genes = [gene_1, gene_2, gene_3]
>>> print(all_genes[0][-1])

>>> 0.55
```



The diagram illustrates the indexing process for the last element of the first list in a list of lists. Two green arrows originate from the text labels 'gene\_1' and 'last entry' at the bottom. The 'gene\_1' arrow points to the index '0' in the expression 'all\_genes[0][-1]', representing the first list in the 'all\_genes' collection. The 'last entry' arrow points to the index '[-1]', representing the final element within that specific list.

# Tuples

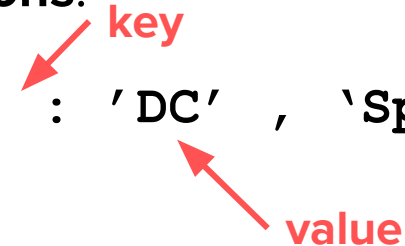
- A tuple is an **immutable** collection of ordered items, that can be of mixed type.
- Tuples are created using parentheses.
- Indexing works similar to lists.

```
>>> my_tuple = ( 3, 'blue', 54.1)
```

# Dictionaries link keys to values

- Denoted by **curly braces** and elements are separated by **commas**. Assignments are done using **colons**.

```
>>> capitals = { 'US' : 'DC' , 'Spain' : 'Madrid' ,  
                 'Italy' : 'Rome' }
```



A diagram with two red arrows. One arrow points from the word 'key' to the string 'US' in the dictionary assignment. The other arrow points from the word 'value' to the string 'DC'.

```
>>> capitals[ 'US' ]
```

```
>>> 'DC'
```

- You'll get a Key Error if you ask for a key that doesn't exist
  - Use **'Germany'** in **capitals** to check

# Working with dictionaries in Python

- Use `capitals.update(morecapitals)` to add another dictionary
- Use `del capitals['US']` to delete entries
- Loop by key or values, or both



# When dictionaries are useful

1. Flexible & efficient way to associate labels with heterogeneous data
2. Use where data items have, or can be given, labels
3. Appropriate for collecting data of different kinds (e.g., name, addresses, ages)

# Resources

[Plotting and Programming in Python: Lists](#)

[Storing Multiple Values in Lists – Programming with Python](#)

[Python 101: Lists, Tuples, and Dictionaries](#)

[Whirlwind Tour of Python: Built-In Data Structures](#)