PYBITES PYTHON TIPS

Real World Python Tips for the Well-Rounded Developer

BOB BELDERBOS JULIAN SEQUEIRA

PyBites Python Tips

Real World Python Tips for
the Well-Rounded Developer

Bob Belderbos & Julian Sequeira

PyBites Python Tips: Real World Python Tips for the Well-Rounded Developer

By Bob Belderbos & Julian Sequeira

Version 4.0.0

For any issues or inquiries email us: support@pybit.es

Thanks to Robin Beer and Chris May for proof reading and a special thanks to our technical reviewer Andrew Jarcho.

Cover design by @ryanurz

Contents

Introduction

Tip: a small but useful piece of practical advice. - Google search

Beautiful is better than ugly. - Zen of Python

Welcome to our Python tips book. It has been a long time coming and we are so proud to finally get it into your hands (digitally speaking).

Great developers read and write a lot of code and our tips have helped thousands of them improve their Python.

Python is a beautiful language with a rich standard library but it's still a huge undertaking to become proficient with it.

It can be difficult to discover awesome features that will make you shine as a developer.

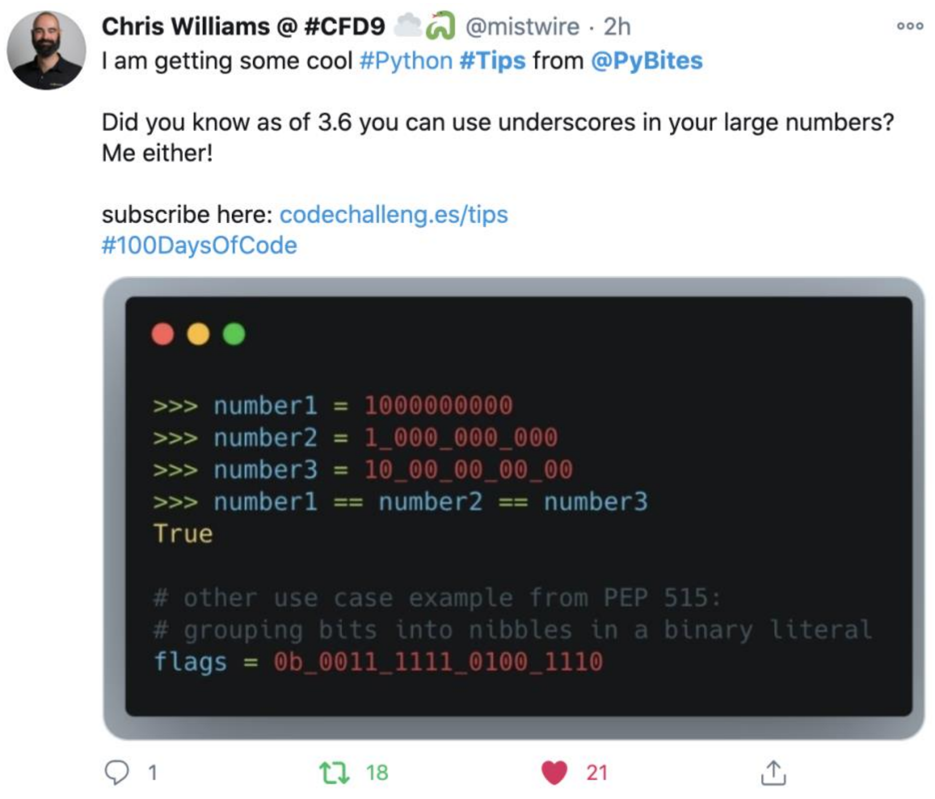
The Zen of Python says: *There should be one-- and preferably only one --obvious way to do it.*

This can potentially take years to figure out which is why we distilled our years of experience into these practical snippets that will help you get there faster.

Regularly reviewing our tips is also the ideal *spaced repetition* that will make you a more effective developer. It will save you lines of code, will make your code more idiomatic ("Pythonic") and you will impress your colleagues and tech recruiters with your increasing knowledge of the language.

A little bit of history

We started sharing tips on *Twitter* roughly 2 years ago and the beautiful code images (produced with Carbon) gained traction immediately. This is what inspired us to write this book.



Chris Williams @ #CFD9 @mistwire · 2h

I am getting some cool #Python #Tips from @PyBites

Did you know as of 3.6 you can use underscores in your large numbers? Me either!

subscribe here: codechalleng.es/tips
#100DaysOfCode

```
>>> number1 = 1000000000
>>> number2 = 1_000_000_000
>>> number3 = 10_00_00_00_00
>>> number1 == number2 == number3
True

# other use case example from PEP 515:
# grouping bits into nibbles in a binary literal
flags = 0b_0011_1111_0100_1110
```

How to read this book

This is not a book you need to read in order. Feel free to jump around to any code snippet or topic that interests you. We want you to get the most out of this book by finding code that excites you. Don't feel you have to read *everything* cover-to-cover. That is, **don't get hit by Tutorial Paralysis**!

We hope you get a lot out of this book. Please hit us up when you use one of our snippets in your code or just to tell us the ones you like the most.

Exercises

Where applicable we link to exercises on our CodeChalleng.es platform. Note that these are not included as part of this book, but you can code them as part of a free 2 week trial.

Feedback

If you encounter any issues while reading this book, however trivial, we'd really appreciate it if you send us your feedback.

For those new to PyBites …

Just in case this is your first interaction with us, welcome!

PyBites was founded 19th of December 2016 as a simple Python blog but has since morphed into a fully-fledged Python training and career coaching business.

Our mission: we are passionate and committed to creating **well-rounded Python developers**, be it through our courses, exercises or mentoring clients individually.

As you'll notice through the tips you're about to read (and hopefully code), what sets us apart is our **practical approach**, **real-world experience** in our materials, and the focus on **leadership (mindset) skills** that are crucial to gaining optimal performance in your career.

So who is behind PyBites?

Meet Bob

Bob studied Business Economics but got fired up about programming early in his career.

He taught himself web design / coding and started living his biggest passion: automate the boring stuff, making other people's lives easier.

Since then he has coded projects accruing millions in cost savings and built a coding platform that has taught Python to thousands of people worldwide.

He deeply cares about helping other people succeed. His biggest win will be your next win!

Meet Julian

With a background in Servers, Data Centers and Communications at 3 of the largest companies on the planet, Julian made the challenging switch to a Python software developer role in 2019.

After learning Python teaching it on PyBites with best friend Bob, Julian has made it his mission to spread not only his love of Python but also his expertise in communication and mindset.

His infectious enthusiasm, motivation and ability to energize those around him greatly assist him in his role of mentoring aspiring Python developers keen on growing their careers.

What we do

To find out more about what we do, make sure you check out the end of this book. With that said, go enjoy the tips!

How our community experiences our tips

"This is truly a vital addition to every Python programmers toolkit: bang up to date with tips that work correctly on the latest versions of Python. Each tip has appropriate references to online documentation if ever further elaboration is needed. Get out the shredder, you're not going to be short of kitty litter for some time after you rid yourself of all those old, dated Python 2.7 hints and tips books: PyBites Python Tips is the premier collection of tips for today's Python. Buy it now: buy it twice and do a friend a favour!" - **Geoff R**

"In terms of knowledge per dollar this is the best investment I've made in the past few years. This is honestly my favorite technical book. Do you ever go on one of those cooking websites for a recipe and have to scroll for what feels like an eternity to get to the ingredients and the 4 steps the recipe actually takes? This is the opposite of that. Each tip is one page: there's some code, an explanation and links to resources if you want to read more. It's not only all you need, it's all you want. I wish there were more books like this!" - **Sergio S**

"Joining the PyBites Platform and utilizing the code challenges in the PyBites Python Tips book has increased my Python skill set is relatively short time while constantly keeping me challenged. Tip #73 helped me with pulling in multiple files to analyze data, a great opportunity to automate a repetitive task I perform on a daily basis. It's very empowering to work through the code challenges and be able to put them into real world practice. Thank you Bob and Julian for bringing together this amazing community!" - **Beth B**

"They did it again. A new cool product! I enjoy everything PyBites does. Minimalistic approach and team work creates value! I read Bob's posts on Python tips and try to put in practice. When Bob and Julian announced their new book, I was so happy – no need to take screenshots as a madman! 😉 The best is that purchasing their book I could download the code and save it in my utils." - **Alex K**

"I have been reading your new Python Tips book for a little while now, just one tip per day along my #100DaysofCode journey. I really love the crisp, practical examples and the links to further reading. The book was indeed a steal. Looking forward to more tips!" - **Andreas J**

"The book is a great and handy reference covering a wide array of topics, from Python's standard library to 3rd party tools, such as howdoi, requests and imagio, just to name a few. The discussions are succinct yet thorough enough to give you a solid grasp of the particular problem. I just wish I would have had this book when I started learning Python." - **Daniel H**

"The PyBites book has me saying, "oh, I used this", "this is new". Great book, for me a Python newcomer and I'll be using it as a reference book!" - **Israel L**

“Bob and Julian are the masters at aggregating these small snippets of code that can really make certain aspects of coding easier. I've been a member of the PyBites site for a few months now, and have always enjoyed getting the tips that you get after completing each bite. I've been taking notes for quite a while and would add each tip into my notes, but having them in book form is like having my own personal index of useful tips. Just today, I used an aspect of sets (set.difference) to help me solve a problem I was working on. Without that, I would have had to do a bunch of looping and conditional statements, but thanks to going through the PyBites Tips book, I remembered this set method that I had forgotten about. There are so many more tips just like this in the book, and they are continually adding to it! I'm still going through the book and am looking forward to the release of the new tips as well.” - **Jesse B**

"The PyBites Python Tips book has been very helpful to me in my coding. Pythonistas of every skill level will find that a small investment of time, with this excellent book, will yield big dividends. Buy it now!” – **Andrew J**

"Need some great, helpful tips? Want quick examples how to use itertools? I've been getting the PyBites tips through twitter for a long time. Now 100 of them (more to come) are easily accessible in this extremely affordable e-book. Great collection! Thanks!" – **Bart H**

"This book should be in every Pythonista's arsenal, regardless if you're a professional developer or hobbyist. I use the PyBites Python Tips book daily during my lunchtime at work; I just open it up and pick a random tip to read. In just a few minutes, I will have:
1) grasped the idiomatic pythonic way of doing something,
2) understood the explanation behind the tip and opened up a heap of suggested browser links to investigate more about the topic,
3) typed out the few lines of code provided, cementing the knowledge gained, and finally,
4) tested myself by applying it to solve a PyBites exercise.
The wealth of knowledge this book provides compounds daily and I've enjoyed learning about the Python built-in standard library and many 3rd party tools too, all in a very approachable and practical way.
This is now my favourite first Python go-to reference. If you don't get this book, you're seriously missing out!" - **Anthony L**

"You know all those tricks you pick up on the way to learning something? All those 'oh neat!' and 'I didn't know that' and 'that will save me a TON of time' tidbits that you accumulate along the way? Well Bob and Julian made an entire book out of their amazing python tips and it's fantastic! Flip to any page and you will learn something new and get an exercise to test out your new knowledge to boot!" - **Chris W**

“I first started really using Python about 5 or 6 years ago when I undertook a MSc in Analytics. Once I’d completed my Masters I let my new Python skills slowly deteriorate through lack of use and this bothered me. When I discovered PyBites nearly two years ago I found a medium through which I could once again immerse myself in Python. I loved the targeted nature of the exercises and how you were encouraged to go and learn about parts of the Python Standard Library that you may not have been aware of. So when I heard that Bob and Julian were bringing out a book I knew that it would be a must have read, and I wasn’t disappointed. The book is packed over 150 tips of Python wisdom, and growing, where each tip can be consumed and understood in a matter of minutes. Each Pythonic nugget gives clear and concise example code and also links to resources where you can go deeper if you want further explanation of the concepts. The tips show you the libraries and patterns you should be using in your everyday Python use and is also a great reference for when you know the command that you want to use but can’t quite remember the right syntax. Now you’ll be able to quickly refer back to your go to solution again and again and again. Congratulation to Bob and Julian on the release of this book which continues their great work promoting Python to an ever increasing audience.” - **David C**

"The book is really awesome. It provides some really great insights on the vast Standard Library and the external packages. Every time I feel stuck on a problem and I need some clarification on possible methods, I refer to the book." - **Gabriel S**

"A great companion! I keep this book on my desktop so I can refer to it quickly and even browse through it for fun in my spare time. Since I'm new to python, it's nice having a collection of useful tips curated by folks with much more experience than me. It gives me the confidence knowing that the solutions within are good ones, which is not always the case when searching the web for answers. For that reason alone, I feel this book gives tremendous value. I've already used several tips in my personal projects and at work. Thank you, Bob and Julian for this!" **- Ed G**

Share your story

We don't want this to be just another book you collect. We want to know the book got you RESULTS! So, if that's the case we'd like to ask you one (or two) favors:

1. Tell people learning Python about this book. You can use our book's landing page which we'll keep up2date as we grow the collection of tips - thanks!

2. Share with us how the book helped you write better Python. We'd be super grateful if you'd email us your story / honest review.

Acknowledgements

We couldn't have achieved any of this if it wasn't for the support of our beautiful families. Lili, Amy, Bryan | Mel, Oliver, Charlie and Lily – thank you for your patience and unwavering support over the years. We love you.

Of course, a huge shout out and thank you to our amazing community, especially those of you that have been here from day one. Without your input and consistent feedback PyBites wouldn't be where it is today.

-- Bob and Julian

Download the code

You can download a zip file with all the code snippets shown in the book here:
http://codechalleng.es/api/books/tips

1. Swap 2 variables

Need to swap 2 variables in Python? No problemo, just 1 line of code:

```
>>> a = 1
>>> b = 2
>>> a, b = b, a
>>> a
2
>>> b
1
```

Explanation

In other languages you need to keep an intermediate variable to accomplish this.

Not so with Python thanks to tuple unpacking.

Resources

https://stackoverflow.com/a/14836456

2. Split a string into a list

An easy way to make a list in Python is to use the `str.split` method on a string which splits on space by default:

```
>>> names = 'bob julian tim sara'.split()

>>> names

['bob', 'julian', 'tim', 'sara']
```

Explanation

Of course nothing wrong creating a list like `['bob', 'julian', 'tim', 'sara']`, but it can be convenient using the `split` on a string.

This only works though if the elements don't have spaces in them, so if we'd include surnames here, you'd have to create the list like this: `['bob belderbos', 'julian sequeira', 'tim ferriss', 'sara blakely']` (or still have your cake by using commas like this: `'bob belderbos,julian sequeira,tim ferriss,sara blakely'.split(',')`).

Resources

https://docs.python.org/3/library/stdtypes.html#str.split

3. Create a dictionary using zip

Create a `dict` of two sequences using the `zip` built-in:

```
>>> names = 'bob julian tim sara'.split()
>>> ages = '11 22 33 44'.split()
>>> zip(names, ages)
<zip object at 0x7fae75920d20>
>>> list(zip(names, ages))
[('bob', '11'), ('julian', '22'), ('tim', '33'), ('sara', '44')]
>>> dict(zip(names, ages))
{'bob': '11', 'julian': '22', 'tim': '33', 'sara': '44'}
```

Explanation

The `dict` constructor can receive a list of tuples.

Here we use `zip` to combine names and ages. This built-in creates an iterator intertwining two or more sequences ("iterables").

By feeding this into `dict` we get a dictionary back where the first elements of each tuple pair are the keys and the second elements are the values.

It only works with 2 element tuples, giving it 3 you'd get a `ValueError: dictionary update sequence element #0 has length 3; 2 is required`.

Resources

https://stackoverflow.com/a/209854

4. Range

In Python you can use the `range` built-in to generate a sequence of numbers:

```
# range = iterable / lazy
>>> range(1, 11)
range(1, 11)
# so casting to list
# also note upper bound is exclusive
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# 3rd arg = stepping
>>> list(range(1, 11, 2))
[1, 3, 5, 7, 9]
# reverse
>>> list(range(11, 1))
[]
# use negative step
>>> list(range(11, 1, -1))
[11, 10, 9, 8, 7, 6, 5, 4, 3, 2]
```

Explanation

The `range` built-in is very useful to construct a sequence of numbers (`int`s).

Note that the first number is inclusive, the second exclusive. An optional 3rd argument can be provided as "step".

A curious fact is that we can iterate over a range object, but it does not return an iterator (calling `next` on it will give you a `TypeError: 'range' object is not an iterator`).

Resources

https://docs.python.org/3/library/functions.html#func-range

Exercise

Bite 1. Sum n numbers

5. Concatenate / multiline strings

In Python string literals will concatenate like this:

```
>>> 'Python' ' is ' 'fun'
'Python is fun'

>>> multiline = ('Great developers read a lot of code.'
...              ' We hope our tips help with that!')
>>>
>>> multiline
'Great developers read a lot of code. We hope our tips help with that!'
```

Explanation

This is a useful technique when you break down long strings over multiple lines, especially useful to comply with PEP8's "Limit all lines to a maximum of 79 characters".

Resources

https://stackoverflow.com/a/1874679
https://pep8.org/#maximum-line-length

6. Tuple unpacking

Python’s tuple unpacking can be really useful:

```
>>> url = ('https://www.amazon.com/War-Art-Through-Creative-Battles'
...        '/dp/1936891026/?keywords=war+of+art')

>>> url.split('/')
['https:', '', 'www.amazon.com', 'War-Art-Through-Creative-Battles', 'dp',
'1936891026', '?keywords=war+of+art']

>>> url.split('/')[2:-1]
['www.amazon.com', 'War-Art-Through-Creative-Battles', 'dp', '1936891026']

>>> domain, *rest, asin = url.split('/')[2:-1]
>>> domain
'www.amazon.com'
>>> rest
['War-Art-Through-Creative-Battles', 'dp']
>>> asin
'1936891026'

# this works too of course
>>> elements = url.split('/')
>>> elements[2]
'www.amazon.com'
>>> elements[-2]
'1936891026'
```

Explanation

Here we split the `url` string by a slash (`/`) and take a slice of the elements we want.

We then use tuple unpacking to get the first and last elements saving them to `domain` and `asin`.

You can use the `*` to assign multiple elements to a list (`*rest`).

Of course regular list indexing works too here.

Resources

https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences

7. Enumerate

If you need the index inside a loop in Python use `enumerate`:

```
>>> names = 'bob julian tim sara'.split()
>>> for i, name in enumerate(names, start=1):
...     print(i, name)
...
1 bob
2 julian
3 tim
4 sara
```

Explanation

Wrapping `enumerate` around an iterator you get a counter for free.

By default it starts at 0, but you can change that using the optional `start` keyword arg.

Resources

https://docs.python.org/3/library/functions.html#enumerate

Exercise

Bite 15. Enumerate 2 sequences

8. Sorted / min / max key argument

In Python, the `sorted` / `min` / `max` builtin functions take an optional "key" keyword argument that receives a callable:

```
>>> ages = {'bob': 23, 'julian': 11, 'tim': 7, 'sara': 37}

>>> sorted(ages.items())
[('bob', 23), ('julian', 11), ('sara', 37), ('tim', 7)]

>>> sorted(ages.items(), key=lambda x: x[1])
[('tim', 7), ('julian', 11), ('bob', 23), ('sara', 37)]

>>> sorted(ages.items(), key=lambda x: x[1], reverse=True)
[('sara', 37), ('bob', 23), ('julian', 11), ('tim', 7)]

# another example

>>> names = 'bob Julian tim sara'.split()
>>> sorted(names)
['Julian', 'bob', 'sara', 'tim']
>>> sorted(names, key=str.lower)
['bob', 'Julian', 'sara', 'tim']
```

Explanation

This is a very useful technique to sort items by a different item in the sequence in case of tuples, or different attribute if you are sorting objects.

A `lambda` is an inline function, it's the only place where we use them. Normally we just use a regular function (`def my_function(...)`).

And to reverse the sorting order, set the `reverse` keyword argument to `True`.

Lastly `sorted` returns a new copy, `sort` however does the sorting in-place.

Resources

https://docs.python.org/3/howto/sorting.html#sortinghowto

Exercise

Bite 5. Parse a list of names

9. Chain comparison operators

In Python you can chain comparison operators like this:

```
>>> a = 5
>>> b = 15
>>> 1 < a < 10
True
>>> 1 < b < 10
False
```

Explanation

Instead of writing `1 < a and a < 10` you can write `1 < a < 10` which is a bit more concise.

Resources

https://docs.python.org/3/reference/expressions.html#comparisons

10. Is vs == (object equality)

The difference in Python between comparing objects and their values:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> c = a
>>> a == b  # same content
True
>>> a == c  # also same content
True
>>> a is c  # same object
True
>>> a is b  # not the same object
False
# to check for equal objects you can check their
# identity using id()
>>> id(a), id(b), id(c)
(140611808855040, 140611819909632, 140611808855040)
```

Explanation

In Python `is` checks that 2 arguments refer to the same object, `==` is used to check that they have the same value.

To check whether variables refer to the same object you can use the `id()` built-in which, as per the docs, returns an “identity” integer which is guaranteed to be unique and constant for the object's lifetime.

Resources

https://stackoverflow.com/a/15008404
https://docs.python.org/3/library/functions.html#id

Exercise

Bite 80. Check equality of two lists

11. F-strings

Starting Python 3.6 you can use f-strings:

```
>>> for i in range(3):
...     '{i} apple{s}'.format(i=i, s="s" if i != 1 else "")
...
'0 apples'
'1 apple'
'2 apples'
>>> for i in range(3):
...     f'{i} apple{"s" if i != 1 else ""}'
...
'0 apples'
'1 apple'
'2 apples'
```

Explanation

F-strings let you embed variables and even expressions.

`format` was definitely nice, but f-strings make it more explicit and concise.

You can even more easily debug variables, but we'll leave that for another tip ...

Resources

https://www.python.org/dev/peps/pep-0498/

Exercise

Intro Bite 01. F-strings and a simple if/else

12. Flatten a list of lists

Two ways of flattening a list of lists:

```
>>> list_of_lists = [[1, 2], [3], [4, 5], [6, 7, 8]]
>>> sum(list_of_lists, [])
[1, 2, 3, 4, 5, 6, 7, 8]

# more explicit
>>> import itertools
>>> list(itertools.chain(*list_of_lists))
[1, 2, 3, 4, 5, 6, 7, 8]

# not recursive though
>>> list_of_lists = [[1, 2], [3], [4, 5], [6, 7, 8, [1, [2, [3, 4]]]]]
>>> list(itertools.chain(*list_of_lists))  # or itertools.chain.from_iterable
[1, 2, 3, 4, 5, 6, 7, 8, [1, [2, [3, 4]]]]
```

Explanation

Using `itertools` seems more explicit to us, but it only goes one level deep. For deeper nesting you will need recursion which you can try in the exercise below.

Resources

https://stackoverflow.com/a/952946

Exercise

Bite 84. Flatten lists recursively (Droste Bite)

13. Get a random sample

Python makes it easy to pick a random sample from a sequence:

```
>>> names = 'bob julian tim sara carmen job martin vero'.split()
>>> from random import sample
>>> sample(names, 2)
['sara', 'julian']
>>> sample(names, 2)
['vero', 'job']
```

Explanation

The `random` module offers great interfaces. Here we take a random sample of 2 items out of the `names` list.

In Python 3.9 a keyword-only `counts` argument was added to repeat values in the sample set.

Resources

https://docs.python.org/3/library/random.html#random.sample

14. Collections.Counter

For counting in Python look no further than `collections.Counter`:

```
>>> from collections import Counter
>>> languages = 'Python Java Perl Python JS C++ JS Python'.split()
>>> Counter(languages)
Counter({'Python': 3, 'JS': 2, 'Java': 1, 'Perl': 1, 'C++': 1})
>>> Counter(languages).most_common(2)
[('Python', 3), ('JS', 2)]
```

Explanation

It does not get more Pythonic than this ;)

`Counter()` can receive an iterable, a mapping or keyword args (nice!)

`most_common` is useful to get, well, the most common elements.

Resources

https://docs.python.org/3.9/library/collections.html#collections.Counter

Exercise

Bite 18. Find the most common word

15. TextBlob (PyPI)

Get a tweet's sentiment in Python using the `textblob` library:

```
>>> from textblob import TextBlob
>>> tweets = ("I was happy with the book", "this is awful", "Python is object
oriented", "Python is awesome")
>>> for tw in tweets:
...     tw, TextBlob(tw).sentiment
...
('I was happy with the book', Sentiment(polarity=0.8, subjectivity=1.0))
('this is awful', Sentiment(polarity=-1.0, subjectivity=1.0))
('Python is object oriented', Sentiment(polarity=0.0, subjectivity=0.0))
('Python is awesome', Sentiment(polarity=1.0, subjectivity=1.0))
```

Explanation

Python comes with batteries included, but don't forget the thousands of packages on PyPI either.

`TextBlob` lets you process textual data. The sentiment property returns a `namedtuple` of the form `Sentiment(polarity, subjectivity)`.

Resources

https://textblob.readthedocs.io/en/dev/
https://pypi.org/

Exercise

Bite 81. Filter and order tweets by polarity values

16. Rotate characters in string

Two ways to rotate a string (left and right) by `n` characters in Python:

```
>>> from collections import deque
>>> s = "PyBites hits 250 tips"
>>> d = deque(s)
>>> n = 7
>>> d.rotate(n)
>>> d
deque(['5', '0', ' ', 't', 'i', 'p', 's', 'P',
       'y', 'B', 'i', 't', 'e', 's', ' ',
       'h', 'i', 't', 's', ' ', '2'])
>>> ''.join(d)
'50 tipsPyBites hits 2'
>>> d = deque(s)
>>> d.rotate(-n)
>>> ''.join(d)
' hits 250 tipsPyBites'
>>> s[n:] + s[:n]
' hits 250 tipsPyBites'
>>> s[-n:] + s[:-n]
'50 tipsPyBites hits 2'
```

Explanation

Slicing might be the most obvious way here, but `collections.deque` has a dedicated method for it.

Resources

https://docs.python.org/3.9/library/collections.html#collections.deque.rotate

17. Deduplicate values using a set

Need unique values from a list in Python? Use a `set`:

```
>>> languages = 'Python Java Perl PHP Python JS C++ JS Python Ruby'.split()
>>> set(languages)
{'Perl', 'JS', 'Python', 'Ruby', 'Java', 'PHP', 'C++'}
```

Explanation

Python has a basic data type called `set` which is an unordered collection with no duplicate elements.

As they are implemented using hash tables, membership checking is fast.

The other use case is eliminating duplicate entries like we've shown here.

Resources

https://docs.python.org/3/tutorial/datastructures.html#sets

Exercise

Bite 130. Analyze some basic Car Data

18. List comprehension to filter out values

Use a list comprehension in Python to filter a list:

```
>>> languages = ['Python', 'Java', 'Perl', 'PHP', 'Python', 'JS', 'C++', 'JS',
'Python', 'Ruby']
>>> [l for l in languages if l.lower().startswith('p')]
['Python', 'Perl', 'PHP', 'Python', 'Python']

# alternative but less readable

>>> list(filter(lambda l: l.lower().startswith('p'), languages))
['Python', 'Perl', 'PHP', 'Python', 'Python']
```

Explanation

You can write a list comprehension from the inside out: place two `[]` brackets, then write `[l for l in languages]`.

You just repeated the list, now at the end put your condition: `if l.lower().startswith('p')` and voilà you are returning a list of all the languages starting with 'p'.

But you don't have to stop there, you can write `set` and `dict` comprehensions too. They are definitely one of our favorite Python features.

Resources

https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

Exercise

Intro Bite 07. Filter numbers with a list comprehension

19. Any / all built-ins

A Pythonic way to check if any / all elements of an iterable are true.

```
>>> languages = ['Java', 'Perl', 'PHP', 'Python', 'JS', 'C++', 'JS', 'Ruby']
>>> all(len(l) >= 2 for l in languages)
True
>>> any('++' in l for l in languages)
True
```

Explanation

Here we use it to see if all language names are at least 2 characters long which is true.

Then we check if there is any language with `++` in it which is also true.

Resources

https://docs.python.org/3/library/functions.html#all

20. Zfill

Give a number leading zeros in Python using `zfill`:

```
>>> for i in range(1, 6):
...     str(i).zfill(3)
...
'001'
'002'
'003'
'004'
'005'

>>> for i in range(1, 6):
...     str(i).zfill(5)
...
'00001'
'00002'
'00003'
'00004'
'00005'

>>> for i in range(-3, 2):
...     str(i).zfill(3)
...
'-03'
'-02'
'-01'
'000'
'001'
```

Explanation

This is a great technique to print titles in your app, e.g. "Bite 02"

Resources

https://docs.python.org/3/library/stdtypes.html?highlight=zfill#str.zfill

21. Get a unique ID

You need a unique ID in Python? Use the `uuid` module:

```
>>> from uuid import uuid4
>>> uuid4()
UUID('fcbb0369-7bd9-464d-b23c-9622d40fdcda')
>>> uuid4()
UUID('3e8e68b4-172b-42db-bcf7-07c12e0c9660')
```

Explanation

`uuid4()` creates a random UUID (Universally Unique Identifier).

Resources

https://docs.python.org/3/library/uuid.html
https://en.wikipedia.org/wiki/Universally_unique_identifier

22. Join strings

Use `join` in Python to combine a list of multiple strings:

```
>>> tweets = ("I was happy with the book", "this is awful",
...           "Python is object oriented", "Python is awesome")

>>> print('\n'.join(tweets))
I was happy with the book
this is awful
Python is object oriented
Python is awesome

>>> print(' >> '.join(tweets))
I was happy with the book >> this is awful >> Python is object oriented >>
Python is awesome

# make sure all elements are strings

>>> '-'.join([100, 'pybites', 'tips', 4, 'you'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int found

>>> '-'.join(str(w) for w in [100, 'pybites', 'tips', 4, 'you'])
'100-pybites-tips-4-you'
```

Explanation

`join` is a string method, so you call it on the string you want to use to concatenate the elements in the iterable you pass into it.

Make sure all these elements are strings or you'll hit a `TypeError`.

Also note that this can be significantly faster than ordinary string concatenation if you have a lot of strings!

Resources

https://docs.python.org/3/library/stdtypes.html#str.join
https://stackoverflow.com/a/3055541

23. Reverse a list

Different ways to reverse a list in Python:

```
>>> numbers = [1, 2, 3, 4, 5]
# in-place
>>> numbers.reverse()
>>> numbers
[5, 4, 3, 2, 1]
>>> list(reversed(numbers))
[1, 2, 3, 4, 5]
>>> numbers
[5, 4, 3, 2, 1]
>>> numbers[::-1]
[1, 2, 3, 4, 5]
```

Explanation

The first example modifies `numbers` in place, `reversed` returns a `list_reverseiterator` iterator which we cast to a `list` to show in the REPL (the original list remains unchanged).

Lastly, using a negative step in the list slicing syntax returns a new object as well.

Resources

https://docs.python.org/3/library/functions.html#reversed

Exercise

Bite 9. Palindromes

24. Calendar.month

Here is how to print a month calendar using Python:

```
>>> import calendar

>>> print(calendar.month(2020, 10))
    October 2020
Mo Tu We Th Fr Sa Su
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

>>> help(calendar.month)
…
formatmonth(theyear, themonth, w=0, l=0) method of calendar.TextCalendar
instance
    Return a month's calendar string (multi-line).

# make the columns wider
>>> print(calendar.month(2020, 10, w=5))
               October 2020
 Mon    Tue    Wed    Thu    Fri    Sat    Sun
                        1      2      3      4
   5      6      7      8      9     10     11
  12     13     14     15     16     17     18
  19     20     21     22     23     24     25
  26     27     28     29     30     31
```

Explanation

Just like the Unix `cal` command you can get a month calendar using `calendar.month` passing in a year and a month.

Optionally you can control the width of the columns with the `w` keyword argument (btw it's amazing what `help` often reveals about modules and methods!)

Resources

https://docs.python.org/3/library/calendar.html#calendar.month

25. Dir and help

How to inspect objects and getting help in Python:

```
>>> a = 'hello'
>>> type(a)
<class 'str'>
>>> str.__mro__
(<class 'str'>, <class 'object'>)
>>> dir(a)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__' ...]
>>> help(a.strip)
Help on built-in function strip:

strip(...) method of builtins.str instance
    S.strip([chars]) -> str
...
```

Explanation

Here are some useful built-ins to know more about your objects:

`type` shows us of what type the object is, a single character is a `str` in Python.

`__mro__` gives you the object's inheritance tree.

`dir` lists … or wait ... why don't you type `help(dir)` in your REPL right now? ...

Resources

https://pybit.es/python-help.html

26. Wrapping text

You can use the `textwrap` module to wrap text to columns:

```
>>> from textwrap import wrap

>>> text = ("Every great developer you know got there by solving "
...         "problems they were unqualified to solve until they "
...         "actually did it. - Patrick McKenzie")

>>> for line in wrap(text, width=80): print(line)
...
Every great developer you know got there by solving problems they were
unqualified to solve until they actually did it. - Patrick McKenzie

>>> for line in wrap(text, width=40): print(line)
...
Every great developer you know got there
by solving problems they were
unqualified to solve until they actually
did it. - Patrick McKenzie
```

Explanation

Here we use `textwrap.wrap` to break down the given text (quote) into a list of lines of width 80 and 40 respectively.

The `width` argument is optional and will default to 70 if not provided.

Resources

https://docs.python.org/3/library/textwrap.html

Exercise

Bite 54. Nicer formatting of a poem or text

27. Copy.deepcopy

Use `deepcopy` to copy compound objects in Python:

```
>>> from copy import copy, deepcopy
>>> items = [dict(id=1, name='laptop')]
>>> items2 = copy(items)
>>> items[0]['name'] = 'macbook'
>>> items
[{'id': 1, 'name': 'macbook'}]
>>> items2
[{'id': 1, 'name': 'macbook'}]  # oops!

>>> items = [dict(id=1, name='laptop')]
>>> items2 = deepcopy(items)
>>> items[0]['name'] = 'macbook'
>>> items
[{'id': 1, 'name': 'macbook'}]
>>> items2
[{'id': 1, 'name': 'laptop'}]  # this object stays intact as per intention
```

Explanation

It's important to know the difference between shallow and deep copy:

- A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.

- A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original

As shown in the example, making a shallow copy of a nested object with `copy` (or using slicing: `[:]`) inner objects are references, so updating one updates all copies.

Not so with `deepcopy`. See our article below for another use case.

Resources

https://pybit.es/mutability.html
https://docs.python.org/3/library/copy.html

Exercise

Bite 32. Don't let mutability fool you

28. Simulate dice rolls

Python's `random`, `range` and `itertools` make it easy to simulate dice rolls:

```
>>> import random
>>> import itertools
>>> dice = range(1, 7)
>>> list(dice)
[1, 2, 3, 4, 5, 6]

>>> combinations = list(itertools.product(dice, repeat=2))  # random.choice
below needs a list
>>> len(combinations)
36
>>> combinations[:5]  # use slicing to get a little sample
[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)]

>>> for _ in range(3):
...     random.choice(combinations)
...
(2, 6)
(1, 4)
(5, 6)
>>> random.sample(combinations, 5)
[(6, 2), (2, 4), (6, 1), (4, 2), (4, 3)]
```

Explanation

Here we use `range` to specify a range of 1-6 (7, the outer bound, is exclusive). As `range` is lazy loading, you have to wrap it in a `list` to see all the values.

Then we use `itertools.product` to get all the possible rolls of the pair of dice (see the second resource link below).

Then we use `random.choice` to get 3 random rolls (`_` indicates we are discarding the use of this loop variable).

We can get rid of the for loop by using `random.sample` which seems designed for this use case. The less code the better :)

Resources

https://stackoverflow.com/a/54873996
https://statweb.stanford.edu/~susan/courses/s60/split/node65.html

29. Itertools.cycle

Use `itertools.cycle` to loop through a sequence ad infinitum:

```
>>> import itertools
>>> lights = itertools.cycle('Red Green Amber'.split())
>>> next(lights)
'Red'
>>> next(lights)
'Green'
>>> next(lights)
'Amber'
>>> next(lights)
'Red'
>>> next(lights)
'Green'
>>> next(lights)
'Amber'
```

Explanation

`itertools.cycle` lets you loop through a sequence indefinitely.

Here Julian used it to simulate traffic lights (as part of our first #100DaysOfCode).

Resources

https://github.com/pybites/100DaysOfCode/blob/master/029/traffic_lights.py
https://docs.python.org/3.8/library/itertools.html#itertools.cycle

Exercise

Bite 63. Use an infinite iterator to simulate a traffic light

30. Pairing friends up

Given a list of friends how many pairs can be formed?

```
>>> import itertools
>>> friends = 'bob tim julian fred'.split()
>>> list(itertools.combinations(friends, 2))
[('bob', 'tim'), ('bob', 'julian'), ('bob', 'fred'), ('tim', 'julian'),
('tim', 'fred'), ('julian', 'fred')]


>>> list(itertools.permutations(friends, 2))
[('bob', 'tim'), ('bob', 'julian'), ('bob', 'fred'), ('tim', 'bob'), ('tim',
'julian'), ('tim', 'fred'), ('julian', 'bob'), ('julian', 'tim'), ('julian',
'fred'), ('fred', 'bob'), ('fred', 'tim'), ('fred', 'julian')]
```

Explanation

This is where Python's `itertools.combinations` shines!

Note that `itertools.permutations` does not work here, because it treats elements as unique based on their position, so it causes two entries: `('bob', 'tim')` and `('tim', 'bob')`.

`combinations` on the other hand takes this duplication out which is what we want here.

Resources

https://docs.python.org/3/library/itertools.html#itertools.combinations

Exercise

Bite 17. Form teams from a group of friends

31. Set operations

You want to compare 2 sequences in Python? Enter set operations:

```
>>> a = {1, 2, 3, 4, 5}  # or use set() on a list
>>> b = {1, 2, 3, 6, 7, 8}
# unique to a
>>> a - b
{4, 5}
# unique to b
>>> b - a
{8, 6, 7}
# in both sets
>>> a & b
{1, 2, 3}
# in either one or the other
>>> a ^ b
{4, 5, 6, 7, 8}
# no need for more verbose (and probably slower) looping
>>> line1 = ['You', 'can', 'do', 'anything', 'but', 'not', 'everything']
>>> line2 = ['We', 'are', 'what', 'we', 'repeatedly', 'do']
>>> for word in line1:
...     if word in line2: print(word)
...
do
>>> set(line1) & set(line2)
{'do'}
```

Explanation

`set` operations are a very powerful feature. As you can see in the code example they can save you a lot of code / looping.

You want to have this trick up your sleeve, so practice the linked exercise below!

Resources

https://docs.python.org/3.8/library/stdtypes.html#set-types-set-frozenset

Exercise

Bite 78. Find programmers with common languages

32. In operator

Do you want to perform multiple membership checks in Python? Just use `in`:

```
>>> color = 'yellow'

# is this a primary color?
>>> color == 'red' or color == 'blue' or color == 'yellow'  # don't do this
True
>>> primary_colors = set(['red', 'yellow', 'blue'])
>>> color in primary_colors
True
```

Explanation

When you do multiple comparisons, it's often more concise to just use the `in` operator.

Special note on membership checking and performance. If you use `in` against a `list` it scans through all items sequentially, which is slower. A `set` (like a `dict`) is a hashable type which makes `in` lookups faster.

Resources

https://docs.python.org/3/reference/expressions.html#in
https://stackoverflow.com/a/14535739

33. What Python version do you use?

Determine the version of Python you are using from the command line and at runtime:

```
$ python -V
Python 3.6.0

$ python3.9 -V
Python 3.9.0

>>> import sys
>>> sys.version_info
sys.version_info(major=3, minor=6, micro=0, releaselevel='final', serial=0)
>>> sys.version_info.major
3
>>> if sys.version_info.major < 3: print("Python 3 only please!")
...
>>> if (sys.version_info.major, sys.version_info.minor) < (3, 9):
...     print("Use Python 3.9")
...
Use Python 3.9
```

Explanation

`sys.version_info` returns version information as a named tuple which is nice, because that lets you access the `major` and `minor` attributes.

Resources

https://docs.python.org/3/library/sys.html#sys.version_info

34. Powerful printing

Python's `print` statement is more powerful than you might think:

```
>>> row = ["1", "bob", "developer", "python"]

>>> print(','.join(str(x) for x in row))
1,bob,developer,python

>>> print(*row, sep=',')
1,bob,developer,python
```

Explanation

You typically would use `print` and `join` together as in the first example.

However the same can be accomplished unpacking the `row` argument list using `*` (see resources below), together with the `sep` keyword argument.

Resources

https://docs.python.org/3/library/functions.html#print
https://docs.python.org/dev/tutorial/controlflow.html#unpacking-argument-lists

35. Prevent overwriting files

Prevent a file from being overwritten in Python:

```
>>> with open('hello', 'w') as f:
...     f.write('hello')
...
5

$ more hello
hello

>>> with open('hello', 'w') as f:
...     f.write('spam')
...
4

# oops

$ more hello
spam

# 'x' prevents this:

>>> with open('hello', 'x') as f:
...     f.write('spam')
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'hello'
```

Explanation

The `bash` shell has a "noclobber" option to protect files from being overwritten.

Python's `open` built-in has the 'x' switch which opens the file for "exclusive creation", failing if the file already exists.

Resources

https://docs.python.org/3/library/functions.html#open

36. Caching API calls (PyPI)

You can cache repeated API calls using the `requests_cache` module:

```
>>> import requests
>>> import requests_cache

# supported backends: sqlite (default), mongodb, redis, memory
# expiring cache in 10 seconds for example sake

>>> requests_cache.install_cache('cache.db', backend='sqlite',
expire_after=10)

>>> resp = requests.get("https://pybit.es/")
>>> resp.from_cache
False
>>> resp = requests.get("https://pybit.es/")
>>> resp.from_cache
True

# waiting for 15 seconds
>>> resp = requests.get("https://pybit.es/")
>>> resp.from_cache
False
# request straight after last one, using cache again
>>> resp = requests.get("https://pybit.es/")
>>> resp.from_cache
True
```

Explanation

If you want to avoid repeated API calls, use `requests_cache`.

It builds up a local persistent cache. This is especially useful when you are developing your app locally or when you are dealing with API rate limits (and possible cost of usage!)

Resources

https://pybit.es/requests-cache.html
https://requests-cache.readthedocs.io/en/latest/index.html

37. Strip punctuation

Two ways to strip punctuation from a string in Python:

```
>>> from string import punctuation
>>> punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'

>>> my_string = "punc;tu.ation!"
>>> table = str.maketrans({key: None for key in punctuation})
>>> my_string.translate(table)
'punctuation'

>>> ''.join([c for c in my_string if c not in punctuation])
'punctuation'
```

Explanation

The first method uses the `maketrans` `str` method which returns a mapping table you can feed into the `translate` method on the string in question.

The second way to accomplish this is to use a list comprehension that discards any characters that are in `punctuation`.

For both methods we leverage the `string` module which has various strings (constants) defined, here `punctuation`, which is defined by the module as "a string containing all ASCII punctuation characters".

Resources

https://docs.python.org/3/library/stdtypes.html?highlight=maketrans#str.maketrans
https://www.w3schools.com/python/ref_string_maketrans.asp

Exercise

Bite 68. Remove punctuation characters from a string

38. Functions are first-class objects

You can access function attributes in Python:

```
>>> def calculate_bmi(weight, length):
...
>>> calculate_bmi.__name__
'calculate_bmi'
>>> calculate_bmi.__doc__
'Given the weight and length, calculate BMI'
>>> calculate_bmi.__code__.co_varnames
('weight', 'length')
```

Explanation

Everything in Python is an object, so are functions.

This means we can access attributes like the example above.

You won't commonly need this but one handy use case is getting the class name for your `__repr__` "dunder" (discussed in another tip): `self.__class__.__name__`

Resources

https://docs.python.org/3/reference/datamodel.html

39. Requests (PyPI)

For web requests use the `requests` library:

```
>>> import urllib.request as req

# oops
>>> url = 'https://pybit.es'
>>> req.urlopen(url).read()
....403

# more work
>>> r = req.Request(url)
>>> r.add_header('User-agent','wswp')

>>> req.urlopen(r).status
200

# requests abstracts all this away
>>> import requests
>>> requests.get(url).status_code
200
```

Explanation

As you can see in this example using `urllib.request` gave some trouble so we had to add headers manually.

`requests` has you covered with its beautiful / elegant interface, so to us it's an external dependency you want to add without thinking twice when you are working with web APIs, scrapers, etc.

Resources

https://requests.readthedocs.io/en/master/

40. Use underscore for big numbers

In Python you can make larger numbers more readable:

```
>>> number1 = 1000000000
>>> number2 = 1_000_000_000
>>> number3 = 10_00_00_00_00
>>> number1 == number2 == number3
True

# other use case example from PEP 515:
# grouping bits into nibbles in a binary literal
flags = 0b_0011_1111_0100_1110
```

Explanation

Thanks to PEP 515 we can now add underscores to numeric literals.

The underscore serves as a visual separator and can be placed anywhere, but to us it makes most sense to separate by thousands when used for big numbers.

Resources

https://www.python.org/dev/peps/pep-0515/

41. Ipaddress module

There is a module in Python to deal with IP addresses:

```
>>> import ipaddress
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')

>>> my_ip = ipaddress.ip_address('fe80:0:0:0:200:f8ff:fe21:67cf')
>>> my_ip
IPv6Address('fe80::200:f8ff:fe21:67cf')
>>> my_ip.version
6

>>> net4 = ipaddress.ip_network('192.0.2.0/24')
>>> net4.netmask
IPv4Address('255.255.255.0')
>>> net4.num_addresses
256
>>> for x in net4.hosts():
...     print(x)
192.0.2.1
...
192.0.2.254
```

Explanation

The `ipaddress` module simplifies various IP address related tasks.

It lets you define IP addresses, networks, host interfaces, and more. It supports both v4 and v6 of the Internet Protocol.

Resources

https://pybit.es/ipaddress.html
https://docs.python.org/3/howto/ipaddress.html

42. Calendar.monthcalendar

Here is how you can create a month calendar in Python:

```
>>> import calendar
>>> from datetime import datetime

>>> now = datetime.now()
>>> now.year, now.month
(2020, 10)

>>> from pprint import pprint as pp
>>> cal = calendar.monthcalendar(now.year, now.month)
>>> pp(cal)
[[0, 0, 0, 1, 2, 3, 4],
 [5, 6, 7, 8, 9, 10, 11],
 [12, 13, 14, 15, 16, 17, 18],
 [19, 20, 21, 22, 23, 24, 25],
 [26, 27, 28, 29, 30, 31, 0]]

>>> cal = calendar.monthcalendar(now.year, now.month+1)
>>> cal[0]
[0, 0, 0, 0, 0, 0, 1]
```

Explanation

`calendar.monthcalendar` returns a matrix representing a month's calendar.

Each row represents a week (starting with Monday); days outside of the month are represented by zeros.

This method really helped us when we built our coding streak calendar on our platform.

Resources

https://docs.python.org/3/library/calendar.html#calendar.monthcalendar

43. Itertools.count

Python's `itertools` makes it easy to make a sequence starting a number:

```
>>> import itertools
>>> seq = itertools.count(11)
>>> next(seq)
11
>>> next(seq)
12

# with step
>>> seq = itertools.count(6, step=3)
>>> for _ in range(3):
...     next(seq)
...
6
9
12

>>> seq = itertools.count(3.75, step=0.25)
>>> for _ in range(3):
...     next(seq)
...
3.75
4.0
4.25
```

Explanation

The functions in the `itertools` module create iterators for efficient looping. Learn and embrace them, they can save you a lot of work.

Here we make an infinite counter starting at 11. You can give it a step and it also works with `float`s.

By the way, the `_` in the `for` loop is a "throwaway" variable (we won't need / use it).

Resources

https://docs.python.org/3.8/library/itertools.html#itertools.count

44. Group dict / mappings

Use `collections.ChainMap` to group multiple dictionaries / mappings together:

```
>>> import argparse
>>> import os
>>> from collections import ChainMap
>>> defaults = {'color': 'red'}

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-c', '--color')
>>> args = parser.parse_args()

>>> cli_args = {k: v for k, v in vars(args).items() if v}
>>> combined = ChainMap(cli_args, os.environ, defaults)
>>> combined['color']
red  # default

>>> os.environ['color'] = 'blue'
>>> combined = ChainMap(cli_args, os.environ, defaults)
>>> combined['color']
blue  # env variable takes precedence

>>> args.color = 'green'  # as if given from the command line
>>> cli_args = {k: v for k, v in vars(args).items() if v}
>>> combined = ChainMap(cli_args, os.environ, defaults)
>>> combined['color']
green  # command line variable takes precedence
```

Explanation

`ChainMap` groups multiple `dict`s or other mappings together to create a single, updateable view.

In this example, extracted from the docs, `ChainMap` allows us to set up a precedence chain: user specified command-line arguments take precedence over environment variables which in turn take precedence over default values.

Resources

https://docs.python.org/3/library/collections.html#chainmap-examples-and-recipes
https://docs.python.org/3/library/collections.html#collections.ChainMap

45. Redirecting standard output

Python's `contextlib` module has a useful context manager for redirecting standard output:

```
from contextlib import redirect_stdout


>>> with open('help.txt', 'w') as f:
...     with redirect_stdout(f):
...         help(pow)
...


>>> with open('help.txt') as f:
...     f.read()
...
'Help on built-in function pow in module builtins: ... ...
```

Explanation

As per the example in the docs, here we capture the output of `help(pow)` in a file and read it back in showing it in the REPL.

To send output to standard error, use `with redirect_stdout(sys.stderr): ...`.

Resources

https://docs.python.org/3/library/contextlib.html#contextlib.redirect_stdout

46. If __name__ == "__main__"

What is the meaning of this commonly used statement in Python?

```
$ more script.py
def func():
    print("Hello from function")



if __name__ == "__main__":
    func()

# main block gets invoked

$ python script.py
Hello from function

$ python
>>> import script  # main block does not get invoked
>>> script.func()
Hello from function
```

Explanation

`if __name__ == "__main__"` is often confusing to people new to Python.

It's used at the end of a script to write code that ONLY executes if the module (script) is called directly.

Code in this `if` does NOT run when the module gets imported into another module or in the REPL.

Resources

https://docs.python.org/3/library/__main__.html

47. Comparing version numbers

Python's standard library keeps amazing us - comparing version numbers made easy:

```
>>> from distutils.version import StrictVersion
>>> StrictVersion('0.12.1') < StrictVersion('1.0.2')
True
```

Explanation

Imagine the amount of nested logic to reliably compare versions. We can tell, because 10 years ago we had to deal with checking Sun Microsystems ILOM firmware releases.

Had we known that the standard library even covers this!

Unfortunately `StrictVersion` seems undocumented, but it's a pretty nifty tool abstracting the complexity away for us.

An alternative is the `packaging` library.

Resources

https://stackoverflow.com/a/6972866
https://packaging.pypa.io/en/latest/

Exercise

Bite 163. Which packages were upgraded?

48. Remove leading whitespace

You can use `textwrap.dedent` to remove any common leading whitespace from every line:

```
from textwrap import dedent


def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
      world
    '''
    print(repr(s))          # prints '    hello\n      world\n    '
    print(repr(dedent(s)))  # prints 'hello\n  world\n'
```

Explanation

This is a useful technique when testing your code.

When adding any multiline string variable in a test function, the indenting of the function body causes extra leading spacing to be included in this variable.

`textwrap.dedent` can be used to remove these leading spaces which helps when writing your `assert` statements.

Alternatively you can use `inspect.cleandoc`.

Resources

https://docs.python.org/3/library/textwrap.html#textwrap.dedent
https://docs.python.org/3/library/inspect.html#inspect.cleandoc

49. The platform module

Python, what OS / system am I coding on today?

```
>>> import platform
>>> platform.machine()
'x86_64'
>>> platform.node()
'Bobs-iMac.local'

>>> platform.platform()
'Darwin-19.6.0-x86_64-i386-64bit'
>>> platform.system()
'Darwin'
>>> platform.release()
'19.6.0'

>>> platform.uname()
uname_result(system='Darwin', node='Bobs-iMac.local', release='19.6.0',
                         version='Darwin Kernel Version 19.6.0: ...
                          ... machine='x86_64', processor='i386')

>>> platform.mac_ver()
('10.15.7', ('', '', ''), 'x86_64')
```

Explanation

Nice module to get some quick information about your OS and hardware.

Resources

https://docs.python.org/3/library/platform.html

50. Object representations

What is the difference between `__str__` and `__repr__` in Python?

```
>>> from datetime import date
>>> today = date.today()
>>> str(today)
'2020-10-31'
>>> repr(today)
'datetime.date(2020, 10, 31)'
```

Explanation

In short, the goal of `__repr__` is to be unambiguous and `__str__` is to be readable.

Or as Ned Batchelder succinctly said: "`__repr__` is for developers, `__str__` is for customers."

As we can see above, the `datetime` module follows this advice nicely.

Resources

https://stackoverflow.com/a/1438297

Exercise

Bite 167. Complete a User class: properties and representation dunder methods

51. Keep a list ordered upon insert

Want to efficiently insert items into a list in sorted order? Use Python's `bisect` module:

```
>>> from bisect import insort

>>> items = [3, 5, 7]

>>> insort(items, 6)
>>> items
[3, 5, 6, 7]

>>> insort(items, 4)
>>> items
[3, 4, 5, 6, 7]

>>> insort(items, 9)
>>> items
[3, 4, 5, 6, 7, 9]

>>> insort(items, 20)
>>> items
[3, 4, 5, 6, 7, 9, 20]

>>> insort(items, 15)
>>> items
[3, 4, 5, 6, 7, 9, 15, 20]
```

Explanation

The `bisect` module provides functions to support maintaining order upon insert.

As per the docs this can improve performance for long lists of items with expensive comparison operations (e.g. repeatedly sorting a list).

Resources

https://docs.python.org/3/library/bisect.html#bisect.insort

Exercise

Bite 181. Keep a list sorted upon insert

52. Inspect objects with the vars built-in

In Python you can use the `vars` built-in to easily access an object's attributes:

```
# script.py
import argparse

parser = argparse.ArgumentParser(description='A simple calculator')
parser.add_argument('-a', '--add', nargs='+', help="Sums numbers")
parser.add_argument('-s', '--sub', nargs='+', help="Subtracts numbers")
parser.add_argument('-m', '--mul', nargs='+', help="Multiplies numbers")
parser.add_argument('-d', '--div', nargs='+', help="Divides numbers")

args = parser.parse_args()

# drop in the debugger
breakpoint()

$ python script.py -a 1 -s 2 -m 3 -d 4

(Pdb) args
(Pdb) vars(args)
{'add': ['1'], 'sub': ['2'], 'mul': ['3'], 'div': ['4']}
```

Explanation

Here we add some command line arguments to a script and set a breakpoint.

Then we run the script and by default `args` does not show anything, hm ...

However `args` is an object and reading up on `vars` we see that without arguments, it's equivalent to `locals()` and with an argument, it's equivalent to `object.__dict__`.

So using `vars` on the object we can inspect the internal attribute dictionary.

Resources

https://docs.python.org/3/library/functions.html#vars
https://www.peterbe.com/plog/vars-argparse-namespace-into-a-function

53. Is a string a number?

You can use the `str.isdigit` method to see if a string is numeric:

```
>>> s = ("It's almost 3 years we launched our PyBites platform"
...      ", which now hosts 300+ exercises, up to the next 100 Bites")
>>>
>>> [int(word) for word in s.split() if word.isdigit()]
[3, 100]
```

Explanation

Here we use it in a list comprehension to extract numbers from a string.

Note that `isdigit` requires all characters in the string to be digits, so `300+` is discarded.

Resources

https://docs.python.org/3/library/stdtypes.html#str.isdigit
https://stackoverflow.com/a/4289557

54. Requests and re.findall (PyPI)

Use `findall` from the `re` module to find all instances that match a regular expression pattern:

```
>>> import re
>>> import requests
>>> html = requests.get('https://pybit.es/archives.html').text

>>> matches = re.findall(r'https://pybit\.es/guest.*?\.html', html)
>>> len(matches)
29
>>> for m in matches: print(m)
...
https://pybit.es/guest-create-aws-lambda-layers.html
https://pybit.es/guest-clean-text-data.html
...
...
https://pybit.es/guest-learning-apis.html
https://pybit.es/guest-making-of-task-manager.html
```

Explanation

Here we first download our archives page's into the `html` variable using `requests`.

Then we find all PyBites guest article links which can be defined as:
- have “/guest” after the root URL (pybit.es),
- after that they have a variable length of any characters (the "?" here means "don't be greedy" = match as little as possible),
- and they end with a literal dot (.) and the string "html".

`matches` will contain a list of all matches, links in this case.

Lastly notice we prepend our regex with "r" which is known as the "raw string notation" which keeps regular expressions sane. Without it, every backslash ('\') in a regular expression would have to be prefixed with another one to escape it.

Resources

https://docs.python.org/3/library/re.html#re.findall
https://docs.python.org/3/howto/regex.html

Exercise

Bite 2. Regex Fun

55. Construct a dict from an iterable

You can create a `dict` using an iterable:

```
>>> nodes
['abcd:22', 'efgh:80', 'ijkl:443']
>>> dict(node.split(':') for node in nodes)
{'abcd': '22', 'efgh': '80', 'ijkl': '443'}

>>> nodes.append('mno:555:bug')
>>> nodes
['abcd:22', 'efgh:80', 'ijkl:443', 'mno:555:bug']

# oops
>>> dict(node.split(':') for node in nodes)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: dictionary update sequence element #3 has length 3; 2 is required

# split has our back though
>>> dict(node.split(':', 1) for node in nodes)
{'abcd': '22', 'efgh': '80', 'ijkl': '443', 'mno': '555:bug'}
```

Explanation

`dict` has different constructors: `dict(**kwarg)`, `dict(mapping, **kwarg)`, and `dict(iterable, **kwarg)`.

As you can see we can give it an iterable which in this example is a sequence of `(node, port)` tuples. Pretty cool!

Of course we rely on a single colon (`:`) in the node strings, so we use `split`'s optional second argument to split only once.

Resources

https://docs.python.org/3/library/functions.html#func-dict

56. Set comprehensions

Using a set comprehension to remove duplicates and operate on each remaining item:

```
>>> names = ['dana', 'tim', 'sara', 'ana', 'joyce', 'dana', 'tim', 'ana']


>>> set(names)
{'tim', 'joyce', 'dana', 'ana', 'sara'}


>>> {name.title() for name in names if "a" in name}
{'Ana', 'Dana', 'Sara'}
```

Explanation

`set`s are great to filter out duplicated elements.

Next we use a set comprehension to also filter on names containing the letter "a", title casing the matching names.

Resources

https://docs.python.org/3/tutorial/datastructures.html#sets

Exercise

Bite 77. New places to travel to

57. Call help inside pdb

Need help inside `pdb` (Python’s debugger)? You can:

```
(Pdb) elem
<Element {http://www.worldbank.org}country at 0x106610908>
(Pdb) p help(elem)
Help on _Element object:
...
```

Explanation

We already saw the power of the `vars` built-in to inspect objects.

`pdb` supports pretty printing using `pp` so we use this often when debugging: `pp(vars(object))`.

But what if you want to get help for an object without leaving the debugger? You can use `p help(...)`.

Resources

https://stackoverflow.com/a/29523730
https://docs.python.org/3/library/pdb.html#debugger-commands

58. Working with file system paths

Since Python 3.4 there is a more elegant way to work with file system paths:

```
>>> from pathlib import Path
>>> tmp = Path('/tmp')
>>> countries = tmp / 'countries.xml'  # before: os.path.join
>>> countries.exists()  # before: os.path.exists
True
>>> countries.is_dir()
False
```

Explanation

`pathlib` makes working with the file system more elegant.

Wonder how `tmp / 'countries.xml'` actually works? According to the Stack Overflow answer below, the `Path` class has a `__truediv__` (dunder) method that returns another `Path`. The power of OOP and Python's data model!

Resources

https://docs.python.org/3/library/pathlib.html
https://stackoverflow.com/a/53085465

59. Validate JSON

Pretty print / validate JSON from the command line using Python's `json.tool`:

```
$ cat MOCK_DATA.json
[{"id":1,"first_name":"Myrle","email":"mleport0@t.co"},
{"id":2,"first_name":"Lynnette","email":"lchurchward1@seattletimes.com"}]

$ python -m json.tool MOCK_DATA.json
[
    {
        "id": 1,
        "first_name": "Myrle",
        "email": "mleport0@t.co"
    },
    {
        "id": 2,
        "first_name": "Lynnette",
        "email": "lchurchward1@seattletimes.com"
    }
]

# validation error

$ echo '[{id: 1}]' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 3 (char 2)
```

Explanation

The tools seems under active development - new command line switches:
- >= 3.5: `--sort-keys`: sort the output of dictionaries alphabetically by key.
- >= 3.9: `--no-ensure-ascii`: disable escaping of non-ascii characters.
- >= 3.8: `--json-lines`: parse every input line as separate JSON object.
- >= 3.9: `--indent`, `--tab`, `--no-indent`, `--compact`: mutually exclusive options for whitespace control.

Resources

https://docs.python.org/3/library/json.html#module-json.tool

60. Create a gif image (PyPI)

Here is how you can create a gif image using the `imageio` library:

```python
import imageio



def create_gif(file_names, out_file="image.gif", duration=1.5):
    images = []
    for filename in file_names:
        images.append(imageio.imread(filename))
    imageio.mimsave(out_file, images, duration=duration)
```

Explanation

`imageio` makes it easy to create simple GIFs.

After pip installing the module, just feed a list of images to the `mimsave` method.

You can define the output file name and the duration between the images as well. You get best results if all images have the same dimensions.

Resources

https://imageio.github.io/
https://github.com/pybites/100DaysOfCode/blob/master/003/create_gif.py

61. Convert str to datetime in pandas (PyPI)

Convert a `DataFrame` column from `str` to `Timestamp`:

```
>>> from io import StringIO
>>> from dateutil.parser import parse
>>> import pandas as pd

>>> data = StringIO("""username;date_joined
... bbelderbos;2017-11-02 09:48:22
... pybites;2017-11-04 11:37:10
... ...
... """)

>>> df = pd.read_csv(data, sep=";")
>>> df.date_joined[0]
'2017-11-02 09:48:22'  # str

# convert from str to Timestamp (whole column!)
>>> df.date_joined = df.date_joined.apply(parse)
>>> df.date_joined[0]
Timestamp('2017-11-02 09:48:22')

# or use:
>>> df.date_joined = pd.to_datetime(df.date_joined)
```

Explanation

First we use `io.StringIO` to create (pandas) `DataFrame` from a string.

But wait, the `date_joined` column returns `str`, not `Timestamp` objects :(

No worries, we can use the `DataFrame`'s `apply` method which applies a function (here: `dateutil.parser.parse`) along an axis of the `DataFrame`. Pretty powerful!

Alternatively use: `df.date_joined = pd.to_datetime(df.date_joined)` which somebody timed to be faster.

Resources

https://stackoverflow.com/a/22605281
https://stackoverflow.com/a/16414011

62. Splitting by newline

Using `split("\n")` to split a text into lines can return odd results if you are on Windows or an (older) Apple computer which use `\r\n` and `\r` for newlines respectively. In those instances, `splitlines` has you covered:

```
>>> "first line\r\nsecond line".split("\n")
['first line\r', 'second line']

# includes \r (Carriage Return) in the splitting
>>> "first line\r\nsecond line".splitlines()
['first line', 'second line']

>>> "first line\rsecond line".splitlines()
['first line', 'second line']

>>> "first line\nsecond line".splitlines()
['first line', 'second line']
```

Explanation

You probably won’t hit this but it's good to know about what Python's glossary defines as "universal newlines":
- the Unix end-of-line convention = `\n`,
- the Windows convention = `\r\n`,
- and the old Macintosh convention = `\r`

`splitlines` accounts for all 3 scenarios and some more.

Resources

https://docs.python.org/3/library/stdtypes.html#str.splitlines
https://docs.python.org/3/glossary.html

63. Group by query in Django's ORM (PyPI)

Let's use some Django ORM magic to get the most common first names on the platform:

```
>>> from django.db.models import Count
>>> from django.contrib.auth.models import User

>>> User.objects.exclude(first_name__exact='').values(
...     'first_name').annotate(
...     name_count=Count('first_name')
... ).order_by('-name_count')[:5].values_list(
...     'first_name', flat=True
... )

<QuerySet ['David', 'Daniel', 'Michael', 'Chris', 'John']>
```

Explanation

Here we use Django ORM's `annotate` which will result in a `GROUP BY` SQL query in the database.

The `order_by('-name_count')` gives us the most common names first. We have a lot of Davids and Daniels coding on our platform!

Resources

https://books.agiliq.com/projects/django-orm-cookbook/en/latest/duplicate.html
https://simpleisbetterthancomplex.com/tutorial/2016/12/06/how-to-create-group-by-queries.html

64. Convert markdown to html (PyPI)

Much nicer to write our Bites in markdown, when done just convert them into HTML with one simple command:

```
(venv) $ pip install markdown

# which creates this console entry script: venv/bin/markdown_py

(venv) $ markdown_py -h
Usage: markdown_py [options] [INPUTFILE]
       (STDIN is assumed if no INPUTFILE is given)

# let's try an example doc

(venv) $ more doc.md
## pybites
this is some markdown
here is a [link](http://example.com).

# bounce html to standard output

(venv) $ markdown_py < doc.md
<h2>pybites</h2>
<p>this is some markdown</p>
<p>here is a <a href="http://example.com">link</a>.</p>

# or save it to a file

(venv) $ markdown_py < doc.md > doc.html
```

Explanation

The code should be self-explanatory, although if you're still new to Unix / command line, standard input / output redirection is a valuable bonus: command line tools are a must in your developer arsenal!

Resources

https://python-markdown.github.io/

65. Get multiple items from a sequence

Another standard library gem: `operator.itemgetter` lets you grab multiple items from a `list`, `dict`, etc.:

```
>>> from operator import itemgetter

>>> days = ['mon', 'tue', 'wed', 'thurs', 'fri', 'sat', 'sun']
>>> f = itemgetter(3, 6)
>>> f(days)
('thurs', 'sun')

# same as:
>>> days.__getitem__(3), days.__getitem__(6)
('thurs', 'sun')

# works with strings too:
>>> f("hello world")
('l', 'w')

# using with dict keys and in one go:
>>> workouts
{'mon': 'chest+biceps', 'tue': 'legs', 'wed': 'cardio', 'thurs':
'back+triceps', 'fri': 'legs', 'sat': 'rest', 'sun': 'rest'}
>>> itemgetter('mon', 'tue')(workouts)
('chest+biceps', 'legs')
```

Explanation

Slicing gets you consecutive items, but what if you want to get non-consecutive items?

Enter `itemgetter` which lets you specify multiple lookup indices as shown above.

`itemgetter` is also a nice alternative for `lambda` when sorting. For example, to sort by the second item of each tuple in a list
(say `inventory = [('apple', 3), ('banana', 2), ...]`), you can do: `sorted(inventory, key=itemgetter(1))`.

Resources

https://docs.python.org/3/library/operator.html#operator.itemgetter

66. Strip takes multiple characters

Python's `str.strip` can remove multiple leading and trailing characters at once:

```
# real use case on our platform
>>> pytest_summary = "=== 20 passed in 0.05 seconds ===\n"
>>> pytest_summary.strip("= \n")
'20 passed in 0.05 seconds'

# another example from the docs
>>> 'www.example.com'.strip('cmowz.')
'example'
>>> comment_string = '#....... Section 3.2.1 Issue #32 .......'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

Explanation

Note it only strips mentioned characters at the beginning and end, not inside the string.

It will eagerly strip till it hits a character that is not in the sequence of characters specified.

If you want to strip on only one side use the `lstrip` method for leading and the `rstrip` method for trailing characters.

Resources

https://docs.python.org/3/library/stdtypes.html#str.strip

67. Instant coding answers via the command line (PyPI)

Need to check Stack Overflow, but don’t want to leave the terminal? Use a neat tool called `howdoi`:

```
$ mkdir ~/howdoi && cd $_
$ python -m venv venv && source venv/bin/activate
(venv) $ pip install howdoi

# I made a useful alias (you could also use vim-howdoi)
$ alias howdoi
alias howdoi='source $HOME/howdoi/venv/bin/activate && howdoi'

# enjoy!
$ howdoi zip enumerate
for index, (value1, value2) in enumerate(zip(data1, data2)):
    ...

$ howdoi argparse
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("a")
…

# or just get (and open) the link to the answer / code
$ howdoi decorator -l
https://stackoverflow.com/questions/2435764/how-to-differentiate-between-
method-and-function-in-a-decorator
$ open `howdoi decorator -l`
```

Explanation

`howdoi` is very useful tool to quickly look for code snippets from the command line.

Here we show you how to install the package into a virtual environment and make a shell alias. Then just enjoy :)

As nice as `howdoi` is on the outside, the code on the inside is elegant too. It was used as an example in The Hitchhiker's Guide to Python's "Reading Great Code" chapter.

Resources

https://github.com/gleitz/howdoi
https://pybit.es/developer-tools.html

68. Match files with Path.glob

Use `Path.glob` to list matching files in a directory (structure):

```
>>> from pathlib import Path

# get python files in current directory
>>> for file_path in Path('.').glob('*.py'): print(file_path)

# get python files one level deep
>>> for file_path in Path('.').glob('*/*.py'): print(file_path)

# get python files recursively (might be slow!)
>>> for file_path in Path('.').glob('**/*.py'): print(file_path)
```

Explanation

Previously we used the `glob` module for this tip, but using `pathlib` it returns `pathlib.PosixPath` instead of `str` objects, which are way easier (Pythonic?) to manipulate as per the `Path` interface.

Note that using the `**` pattern makes it recursive which can significantly slow down the operation on large directory trees.

Resources

https://docs.python.org/3/library/pathlib.html#pathlib.Path.glob

Exercise

Bite 116. List and filter files in a directory

69. String module

Using the `string` module to make a random string:

```
>>> import random
>>> import string

>>> ''.join(random.choice(string.ascii_lowercase) for i in range(20))
'jjkjsgyksqfmtgpujmud'

# "sample" seems more adequate:
>>> ''.join(random.sample(string.ascii_lowercase + string.digits, 20))
'oytkq3u4l6hnbpz19r85'

# other constants you can use:
>>> help(string)
...
DATA
    __all__ = ['ascii_letters', 'ascii_lowercase', 'ascii_uppercase', 'cap...
    ascii_letters = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
    ascii_lowercase = 'abcdefghijklmnopqrstuvwxyz'
    ascii_uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    digits = '0123456789'
    hexdigits = '0123456789abcdefABCDEF'
    octdigits = '01234567'
    printable = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU...
    punctuation = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
    whitespace = ' \t\n\r\x0b\x0c'
```

Explanation

The `string` module has some useful constants.

Here we use two to build random strings. We literally used this to create license keys.

Again, get into the habit of calling `help` on objects to quickly reference documentation.

Resources

https://docs.python.org/3/library/string.html

Exercise

Bite 47. Write a new password field validator

70. Use case fold for Unicode string matching

Case folding is similar to lowercasing but more aggressive:

```
>>> a, b = "der Fluß", "der Fluss"

>>> a.lower() == b.lower()
False

# casefold converts 'ß' "ss" which makes the 2 strings identical
>>> a.casefold() == b.casefold()
True

>>> 'ß'.casefold()
'ss'
```

Explanation

`str.casefold` (new since Python 3.3) is similar to `str.lower` but more aggressive because it is intended to remove all case distinctions in a string.

This is especially helpful for Unicode characters, often quoted is the example of matching a German 'ß' with 'ss'.

Resources

https://docs.python.org/3/library/stdtypes.html#str.casefold
https://pythonbytes.fm/episodes/show/168/race-your-donkey-car-with-python

71. Access your virtual environment in a Jupyter Notebook (PyPI)

How to work with a virtual environment (dependencies) in a Jupyter notebook:

```
# in your virtual environment (assuming it's "venv" in your current directory)
$ pip install ipykernel


# install a new kernel
$ ipykernel install --user --name venv --display-name "My Project (venv)"


$ jupyter notebook
# in your notebook's file browser select "My Project (venv)" from the "New"
dropdown
```

Explanation

It can be annoying to open a notebook and not being able to work with external modules you just `pip install`ed in your virtual environment.

But there is a fix: get `ipykernel` and install a dedicated kernel for your virtual environment.

Then when you open a notebook, you can select your venv and voilà: your virtual env's dependencies are now available for import.

Resources

https://anbasile.github.io/posts/2017-06-25-jupyter-venv/
https://ipython.readthedocs.io/en/stable/install/kernel_install.html

72. Callable and getattr built-ins

You can use the `callable` built-in to see if an object is callable (e.g. a function / method):

```
>>> import string
>>> attrs = [attr for attr in dir(string) if not attr.startswith('_')]
>>> for a in attrs: a, callable(getattr(string, a))
...
('Formatter', True)
('Template', True)
('ascii_letters', False)
('ascii_lowercase', False)
('ascii_uppercase', False)
('capwords', True)
('digits', False)
…

# why getattr?
>>> attrs[0], type(attrs[0])
('Formatter', <class 'str'>)
# oops
>>> callable(attrs[0])
False
# we need to give callable the actual object:
>>> callable(getattr(string, attrs[0]))
True
```

Explanation

Here we call `dir` on the `string` module to see all the "public" attributes.

Then we loop over seeing if they are "callable". Note that we need to use `getattr` to get the actual object (as opposed to the `str` that `dir` returns), because `callable` receives an object.

So there you go, now you also know about the handy `getattr` built-in.

Fun fact: the callable function was removed in Python 3.0 but came back in Python 3.2.

Resources

https://docs.python.org/3/library/functions.html#callable
https://docs.python.org/3/library/functions.html#getattr

73. Open 2 files in parallel

Who said you could only open one file at a time using the `with` statement?

```
$ more f1.txt
a
f
g
s
s

$ more f2.txt
a
scseww
sa
23
saf

>>> with open('f1.txt') as f1, open('f2.txt') as f2:
...     for i, j in zip(f1, f2):
...             print(f'{i.strip()}={j.strip()}')
...
a=a
f=scseww
g=sa
s=23
s=saf
```

Explanation

Here we open two files using the `with` context manager.

Then we use the `zip` built-in to loop over them in parallel, concatenating the lines.

Resources

https://docs.python.org/3/reference/compound_stmts.html#the-with-statement
https://www.python.org/dev/peps/pep-0343/

74. Strip out vowels counting replacements

Here is how to replace all vowels from a text while keeping a count of the number of replacements made:

```
>>> import re
>>> vowels = 'aeiou'
>>> text = """
... The Zen of Python, by Tim Peters
...
... Beautiful is better than ugly.
... [truncated]
... """
>>> new_string, number_of_subs_made = re.subn(f'[{vowels}]', '*', text,
flags=re.I)
>>> new_string[:10]
'\nTh* Z*n *'
>>> number_of_subs_made
262
```

Explanation

Python has a robust `re` module.

You can use `re.subn` to do regex replacing. It returns a `tuple` of the new (replaced) string and the number of replacements made.

This can be a bit cryptic (what if somebody does not know much regex?). So, no shame in using a classic `for` to loop through the string and manually count the replacements. This might be more readable / intuitive.

Or what do you think? Solve the exercise below and hop in the forum to tell us ...

Resources

https://docs.python.org/3/library/re.html#re.subn

Exercise

Intro Bite 06. Strip out vowels and count the number of replacements

75. Freeze a portion of a function

Python's `functools.partial` lets you put a basic wrapper around an existing function:

```
>>> from functools import partial
>>> print_no_newline = partial(print, end=', ')
>>> for _ in range(3): print('test')
...
test
test
test
>>> for _ in range(3): print_no_newline('test')
...
test, test, test, >>>
```

Explanation

Python's `functools.partial` lets you put a basic wrapper around an existing function so that you can set a default value where there normally wouldn't be one.

Here we make our own `print` defaulting the `end` keyword to a comma (overwriting `print`'s default of adding a newline (`\n`) to the end).

So this is a nice way to make a "shortcut" if you always call a function with the same arguments.

Resources

https://docs.python.org/3/library/functools.html#functools.partial

Exercise

Bite 172. Having fun with Python Partials

76. Logging debug info

Log Python errors with debug information:

```
>>> import logging
>>> try:
...     1/0
... except ZeroDivisionError:
...     logging.exception("message")
...
ERROR:root:message
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

Explanation

`logging.exception` will output a stack trace alongside the error message you specify.

Note that this method should only be called from an exception handler to get relevant output.

Resources

https://docs.python.org/3/library/logging.html#logging.Logger.exception
https://stackoverflow.com/a/5191885

77. Use __all__ for encapsulation

Limit module imports with `__all__`:

```
$ more mod.py
__all__ = ['a', 'b']

def a():
    pass

def b():
    pass

def c():
    pass

$ python
>>> from mod import *
>>>
>>> a()
>>> b()
>>> c()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

Explanation

It's considered bad practice to use `from module import *` to import everything into the current namespace, don't do that.

Of course you cannot prevent somebody else from doing this though. So, as a package author you can be proactive about this and use `__all__` to list just the modules you allow to be imported.

Resources

https://docs.python.org/3/tutorial/modules.html#importing-from-a-package

78. Integer caching

Python curiosity: integers in the range -5...256 are cached:

```
>>> a = [5, 200, 256, 257, 300, 500]
>>> b = [5, 200, 256, 257, 300, 500]
>>> for i, j in zip(a, b):
...   print(i, j, i is j)
...
5 5 True
200 200 True
256 256 True
257 257 False
300 300 False
500 500 False
```

Explanation

`is` checks whether 2 things are the same object. As per the Integer Objects docs:

"The current implementation keeps an array of integer objects for all integers between -5 and 256, when you create an int in that range you actually just get back a reference to the existing object."

That's the reason that 256 `is` 256 but not so for bigger integers.

Resources

https://docs.python.org/3/c-api/long.html
https://wsvincent.com/python-wat-integer-cache/

79. Round to the next 1000

The `round` built-in can round before the decimal point as well:

```
>>> round(28, -1)
30
>>> round(288, -1)
290
>>> round(288, -2)
300
>>> round(2888, -3)
3000
```

Explanation

You can use `round` to round to the next 10, 100, 1000, ... using the negative `ndigits` argument.

Resources

https://docs.python.org/3/library/functions.html#round

80. Defaultdict

No more `KeyError` exceptions using a `collections.defaultdict`:

```
>>> from collections import defaultdict
>>> data = """Tim,ID
... Sara,BR
... Thelma,CN
... Chris,RU
... Fina,ID
... Juliana,SE
... Roberto,CN
... Mario,PL
... Paul,CN"""
>>> countries = defaultdict(list)
>>> for line in data.splitlines():
...     name, country_code = line.split(',')
...     countries[country_code].append(name)
...
>>> countries
defaultdict(<class 'list'>,
            {'BR': ['Sara'], 'CN': ['Thelma', 'Roberto', 'Paul'],
             'ID': ['Tim', 'Fina'], 'PL': ['Mario'],
             'RU': ['Chris'], 'SE': ['Juliana']})
```

Explanation

If you're always checking if a key is already in a dictionary before adding a value, use `defaultdict` which makes sure the key is in the `dict` before adding values.

How it works: you give `defaultdict` its `default_factory` argument, in this case a `list`.

When a new key gets inserted into the dictionary it creates a default value for the given key (in our example a `list`), then it inserts the actual value.

Resources

https://docs.python.org/3/library/collections.html#collections.defaultdict

Exercise

Bite 123. Find the user with most friends

81. Parametrization of tests (PyPI)

The `pytest.mark.parametrize` decorator enables parametrization of arguments for a test function:

```python
import pytest

@pytest.mark.parametrize("day, expected", [
    ('Monday', 'Go train Chest+biceps'),
    ('monday', 'Go train Chest+biceps'),
    ('Tuesday', 'Go train Back+triceps'),
    ('TuEsdAy', 'Go train Back+triceps'),
    ('Wednesday', 'Go train Core'),
    ('wednesdaY', 'Go train Core'),
    ('Thursday', 'Go train Legs'),
    ('Friday', 'Go train Shoulders'),
    ('Saturday', CHILL_OUT),
    ('Sunday', CHILL_OUT),
    ('sundAy', CHILL_OUT),
    ('nonsense', INVALID_DAY),
    ('monday2', INVALID_DAY),
])
def test_get_workout_valid_case_insensitive_dict_lookups(day, expected):
    assert get_workout_motd(day) == expected
```

Explanation

`pytest.mark.parametrize` prevents you from writing repeated `assert` statements in your tests, which leads to less redundant and DRY ("don't repeat yourself") code.

Side benefits: it runs the test for each `(day, expected)` parameter `tuple` (= more green dots).

Defining the parameters in a `list` also makes it easier to abstract and re-use them in other tests.

Resources

https://docs.pytest.org/en/stable/parametrize.html
https://pybit.es/pytest-coding-100-tests.html

Exercise

Bite 239. Test FizzBuzz

82. Mocking out web services

Here we mock out `tweepy.API`'s `get_status` API call, swapping it with our own test data:

```
from unittest.mock import patch


...


class WhoTweetedTestCase(unittest.TestCase):

    @patch.object(tweepy.API, 'get_status', return_value=get_tweet('AU'))
    def test_julian(self, mock_method):
        ...

    @patch.object(tweepy.API, 'get_status', return_value=get_tweet('ES'))
    def test_bob(self, mock_method):
        ...
```

Explanation

The idea of mocking is to prevent real access to something, in this case an external API, because it creates a dependency and slows down the tests.

The example code imitates `tweepy.API`'s `get_status` method which would make a call to the Twitter API.

We use `@patch.object`'s `return_value` keyword argument to load in alternative response data using the `get_tweet` helper function.

Resources

https://pybit.es/twitter-api-geodata-mocking.html
https://docs.python.org/3/library/unittest.mock.html#unittest.mock.patch

Exercise

Bite 247. Mocking a standard library function

83. Amazon affiliation URL (PyPI)

Create an affiliation link from an Amazon URL on our clipboard pasting it back to the clipboard:

```
>>> import os
>>> import re
>>> import pyperclip
>>> def gen_affiliation_link(url):
...     if not re.search(r"amazon.*/dp/", url):
...         raise ValueError(f"{url} is not a valid Amazon product link")
...     asin = re.sub(r".*/dp/([^/]+).*", r"\1", url)
...     code = os.environ.get("AMAZON_AFFILIATE_CODE", "pyb0f-20")
...     return f"http://www.amazon.com/dp/{asin}/?tag={code}"
...
>>> def copy_affiliation_link():
...     url = pyperclip.paste()
...     link = gen_affiliation_link(url)
...     pyperclip.copy(link)
...
# going to Amazon, we copy this link to our clipboard:
# https://www.amazon.com/Pragmatic-Programmer-journey-mastery-Anniversary
# /dp/0135957052/ref=sr_1_1?dchild=1&keywords=pragmatic+programmer&sr=8-1
# then we call
>>> copy_affiliation_link()
# which sends this generated affiliation URL back to our clipboard:
# http://www.amazon.com/dp/0135957052/?tag=pyb0f-20
```

Explanation

The `pyperclip` module is a cross-platform Python module for copy and paste clipboard functions. Here we use it to convert an Amazon book URL, sitting on the OS clipboard, to an affiliation link, copying it back to the clipboard.

Note that we broke out the `gen_affiliation_link` helper to make it easier to test this code.

The `.*/dp/([^/]+).*` regex means: " match all up until and including `/dp/`, then capture (using `()`) as many characters that are not equal `/` ". The `\1` in `re.sub` references this captured match which is the asin number (= "0135957052" here).

Resources

https://pypi.org/project/pyperclip/
https://docs.python.org/3/library/re.html#re.sub

84. Working around invalid JSON

`JSONDecodeErrors` can be annoying, but look at what the `ast` module offers us:

```
>>> import ast
>>> import json
>>> a = "{u'person': u'Julian', u'token': u'abc123'}"
>>> type(a)
<class 'str'>
>>> json.loads(a)
...
json.decoder.JSONDecodeError: Expecting property name enclosed in double
quotes: line 1 column 2 (char 1)
>>> ast.literal_eval(a)
{'person': 'Julian', 'token': 'abc123'}
>>> b = ast.literal_eval(a)
>>> type(b)
<class 'dict'>
>>> b['token']
'abc123'

# also used in
https://github.com/plone/plone.schema/blob/master/plone/schema/jsonfield.py

class JSONField(Field):
    ...
    def fromUnicode(self, value):
        ...
        try:
            v = json.loads(value)
        except JSONDecodeError:
            v = ast.literal_eval(value)
```

Explanation

If `json.loads` complains about invalid JSON, you can try to convert your string to a `dict` using `ast.literal_eval` which "safely evaluates an expression node or a string containing a Python literal or container display".

Resources

https://docs.python.org/3/library/ast.html#ast.literal_eval
https://github.com/plone/plone.schema/blob/master/plone/schema/jsonfield.py

85. Parsing dates (PyPI)

The `dateutil` module has wonderful support for converting date strings into `datetime` objects:

```
>>> from datetime import datetime
>>> from dateutil.parser import parse
>>> logline = "INFO 2014-07-03T23:31:22 supybot Killing Driver objects."
>>> date = logline.split()[1]
>>> date
'2014-07-03T23:31:22'
>>> datetime.strptime(date, '%Y-%m-%dT%H:%M:%S')
datetime.datetime(2014, 7, 3, 23, 31, 22)
>>> parse(date)
datetime.datetime(2014, 7, 3, 23, 31, 22)
```

Explanation

It can be hard to remember the right "format code" to feed to `datetime.strptime`.

Luckily though `dateutil` (external library) makes this much easier.

Fun fact: we learned about this technique through one of our Bite forums, so make sure you check those out on our platform!

Resources

https://dateutil.readthedocs.io/en/stable/parser.html
https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior

Exercise

Bite 7. Parsing dates from logs

86. When to use a deque over a list

A `collections.deque` is a list-like container with fast appends and pops on either end:

```
>>> from collections import deque
>>> import random
>>> from timeit import timeit

>>> lst = list(range(10_000_000))
>>> deq = deque(range(10_000_000))

>>> def insert_and_delete(ds):
...     for _ in range(10):
...         index = random.choice(range(100))
...         ds.remove(index)
...         ds.insert(index, index)

# wow
>>> timeit("insert_and_delete(lst)", "from __main__ import insert_and_delete,
lst", number=10)
1.6085020060000375
>>> timeit("insert_and_delete(deq)", "from __main__ import insert_and_delete,
deq", number=10)
0.00024063900002602168
```

Explanation

In any programming language it's important that you know the right data structure for the job.

`deque`s can come in handy depending the list operations. As per the docs: "deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction."

In the example above we see how this can seriously improve the performance of your code!

Resources

https://docs.python.org/3/library/collections.html#collections.deque

Exercise

Bite 45. Keep a queue of last n items

87. Fetch attributes from an object

The `operator.attrgetter` class lets you fetch one or more attributes:

```
>>> from operator import attrgetter
>>> class Person:
...     def __init__(self, name, age, profession):
...         self.name = name
...         self.age = age
...         self.profession = profession

>>> jane = Person("Jane Goodall", 86, "primatologist")
>>> attrgetter('name')(jane)
'Jane Goodall'

# get multiple attributes
>>> ag = attrgetter('name', 'age', 'profession')
>>> ag(jane)
('Jane Goodall', 86, 'primatologist')

# only works for valid attributes
>>> attrgetter('name2')(jane)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Person' object has no attribute 'name2'
```

Explanation

As per the documentation `operator.attrgetter` returns a callable object that fetches attr from its operand. If more than one attribute is requested, it returns a tuple of attributes.

So here we fetch one attribute from the `jane` instance first, following by all 3 in one go.

Resources

https://docs.python.org/3/library/operator.html#operator.attrgetter

Exercise

Bite 135. Sort a list of book objects

88. Adding a __str__ aids in debugging

Always add a `__str__` method to your Django models (and Python classes for that matter):

```
# no __str__ method
class Book(models.Model):
    bookid = models.CharField(max_length=20)  # google bookid
    title = models.CharField(max_length=200)
    ...

$ python manage.py shell
>>> from books.models import Book
>>> Book.objects.count()
2
>>> Book.objects.last()
<Book: Book object (2)>

# with __str__ method
class Book(models.Model):
    bookid = models.CharField(max_length=20)  # google bookid
    title = models.CharField(max_length=200)
    ...

    def __str__(self):
        return f'{self.id} {self.bookid} {self.title}'

$ python manage.py shell
>>> from books.models import Book
>>> Book.objects.last()
<Book: 2 1ZxxDwAAQBAJ Think & Grow Rich>
>>> Book.objects.first()
<Book: 1 1KuwDwAAQBAJ The Coaching Habit>
```

Explanation

As you can see in the code above adding a `__str__` (dunder) method makes it easier to inspect objects in the Django shell or when debugging your code in general.

Resources

https://stackoverflow.com/a/56339054
https://dbader.org/blog/python-dunder-methods

89. Assert_called_with

Here is how to assert a mocked out function was called:

```python
from unittest.mock import patch


@patch.object(facebook.GraphAPI, 'put_object', autospec=True)
def test_post_message(self, mock_put_object):
    sf = simple_facebook.SimpleFacebook("fake oauth token")
    sf.post_message("Hello World!")
    mock_put_object.assert_called_with(message="Hello World!")
```

Explanation

The `assert_called_with` method (`unittest.mock`/ mock object library) is a convenient way of asserting that the last call has been made with the specified arguments.

Here is an example we found in the linked "An Introduction to Mocking in Python" article which checks a Facebook message was posted without hitting FB's API.

Resources

https://docs.python.org/3/library/unittest.mock.html#unittest.mock.Mock.assert_called_with
https://www.toptal.com/python/an-introduction-to-mocking-in-python

90. Access the exception info in your test (PyPI)

Here is how to access the actual exception info in your `pytest` test:

```
import pytest

def test_new_score_should_be_higher(yellow_belt):
    assert yellow_belt.score == 50
    with pytest.raises(ValueError) as excinfo:
        yellow_belt.score = 40
    assert str(excinfo.value) == "Cannot lower score"


(Pdb) excinfo.value
ValueError('Cannot lower score')
(Pdb) str(excinfo.value)
'Cannot lower score'
```

Explanation

To obtain the exception value string, cast the exception to `str` outside of the `with pytest.raises` block.

Resources

https://docs.pytest.org/en/stable/assert.html#assertions-about-expected-exceptions

91. Dictionary access without KeyError

Ready for something pretty idiomatic / elegant? Meet `dict`'s `get` method:

```
>>> workouts = {
...     "Mon": "upper body 1",
...     "Tue": "lower body 1",
...     "Thu": "upper body 2",
...     "Fri": "lower body 2"
... }
>>> workouts['Mon']
'upper body 1'
>>> workouts['Fri']
'lower body 2'
>>> workouts['Sat']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Sat'
>>> ret = workouts.get('Sat')
>>> ret is None
True
>>> workouts.get('Sat', 'not a workout day')
'not a workout day'
```

Explanation

The `dict.get` method checks if the provided key is in the dictionary. If not it returns `None`, unless you give it a default value as second argument.

So if the `Sat` key is not in the dictionary, `workouts.get('Sat')` returns `None`, but `workouts.get('Sat', 'default_value')` returns `default_value` as that is defined as default.

Resources

https://docs.python.org/3/library/stdtypes.html#dict.get

Exercise

Intro Bite 09. Workout dictionary lookups

92. Itertools.takewhile

Get all the ninja belts up until a certain score using `itertools.takewhile`:

```
>>> ninja_belts
{10: 'white', 50: 'yellow', 100: 'orange', 175: 'green', 250: 'blue', 400:
'brown',
 600: 'black', 800: 'paneled', 1000: 'red'}
>>> from itertools import takewhile
>>> def get_belts(user_score):
...     return takewhile(lambda x: x[0] <= user_score,
...                      ninja_belts.items())
...
>>> for i in (7, 18, 51, 174, 176):
...     i, list(get_belts(i))
...
(7, [])
(18, [(10, 'white')])
(51, [(10, 'white'), (50, 'yellow')])
(174, [(10, 'white'), (50, 'yellow'), (100, 'orange')])
(176, [(10, 'white'), (50, 'yellow'), (100, 'orange'), (175, 'green')])
```

Explanation

`itertools.takewhile` receives a predicate and an iterable. It returns a generator that returns elements from the iterable as long as the predicate is true.

So here our predicate checks if the subsequent belt scores are less than the user's score, effectively returning all the belts this user has earned so far.

This code runs on our platform as we speak ☺

Resources

https://docs.python.org/3/library/itertools.html#itertools.takewhile
https://pybit.es/itertools-examples.html

93. Swap dictionary keys and values

Swap a `dict`'s keys and values in just one line of code (!) using a dictionary comprehension:

```
>>> ninja_belts
{10: 'white', 50: 'yellow', 100: 'orange', 175: 'green', 250: 'blue', 400:
'brown',
 600: 'black', 800: 'paneled', 1000: 'red'}

>>> {v:k for k, v in ninja_belts.items()}
{'white': 10, 'yellow': 50, 'orange': 100, 'green': 175, 'blue': 250, 'brown':
400,
 'black': 600, 'paneled': 800, 'red': 1000}
```

Explanation

Here we see the power of comprehensions which allow us to loop through the `dict`'s (key, value) pairs using `items()`, swapping them in the "expression" part of the comprehension.

We just love Python :)

Resources

https://www.python.org/dev/peps/pep-0274/
https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

94. Itertools.groupby

Group car data by manufacturer using `itertools.groupby`:

```
>>> from itertools import groupby
>>> from operator import itemgetter
>>> cars = [
...     ('Toyota', 'Avalon'), ('Ford', 'Bronco'),
...     ('Chevrolet', 'Cavalier'), ('Chevrolet', 'Corvette'),
...     ('Volkswagen', 'GTI'), ('Toyota', 'Highlander'),
...     ('Chevrolet', 'Impala'), ('Nissan', 'Maxima'),
...     ('Ford', 'Mustang'), ('Kia', 'Optima'),
...     ('Volkswagen', 'Passat'), ('Nissan', 'Pathfinder'),
...     ('Ford', 'Taurus'), ('Nissan', 'Titan'),
... ]
>>> cars.sort()  # required
>>> for manufacturer, models in groupby(cars, key=itemgetter(0)):
...     print(f'* {manufacturer}')
...     print(", ".join(model[1] for model in models))
...
* Chevrolet
Cavalier, Corvette, Impala
* Ford
Bronco, Mustang, Taurus
… …
* Volkswagen
GTI, Passat
```

Explanation

Here we have a `cars` list of (manufacturer, model) tuples and we want to group the models by manufacturer.

We can use `itertools.groupby` which takes an iterable (`cars`) and a key (function) to group the items by. Here we use `itemgetter` as key which is a callable that gets the first item of each tuple (manufacturer) as the grouping key.

The only requirement for this to work is that the data is sorted, hence the `cars.sort()` call which sorts the cars in-place.

Resources

https://docs.python.org/3/library/itertools.html#itertools.groupby
https://docs.python.org/3/library/operator.html#operator.itemgetter

95. Mocking datetimes (PyPI)

You can use the `freezegun` library to mock out the `datetime` module:

```python
import datetime

from freezegun import freeze_time

from tomorrow import tomorrow


@freeze_time('2020-07-09')
def test_no_args():
    assert tomorrow() == datetime.date(2020, 7, 10)
```

Explanation

Mocking out `datetimes` is tough.

Say you'd use `@patch('datetime.date', your_datetime_mock_object)` in your test, only to have it break when the import changes from:
`import datetime`
to:
`from datetime import date, timedelta`

`FreezeGun` works around this issue by faking Python `datetimes` thoroughly, keeping your tests simple.

Resources

https://pybit.es/guest-freezegun.html
https://github.com/spulec/freezegun

Exercise

Bite 283. Like there's no tomorrow?

96. Read in source code

Get a list of source lines from an object using `inspect.getsourcelines`:

```python
# challenge.py module
from abc import ABC, abstractmethod

class Challenge(ABC):

    def __init__(self, number):
        self.number = number

    @abstractmethod
    def pretty_title(self):
        return 'Subclass should implement'


# test module
import inspect

from challenge import Challenge

def test_baseclass_methods_are_abstract():
    lines = [line.strip() for line in
             inspect.getsourcelines(Challenge)[0]]
    ...
    pretty_title_index = lines.index('def pretty_title(self):')
    assert lines[pretty_title_index - 1] == "@abstractmethod"
```

Explanation

The `inspect` module is very useful to get information about live objects such as modules, classes, methods, etc.

In this real world scenario (taken from our platform), we use it in our tests to verify that the submitted code makes the `pretty_title` method abstract by applying the `@abstractmethod` decorator to it.

Resources

https://docs.python.org/3/library/inspect.html#inspect.getsourcelines

97. A decorator best practice

Preserve docstrings of decorated functions using `functools.wraps`:

```
>>> def mydecorator(function):
...     def wrapped(*args, **kwargs):
...         result = function(*args, **kwargs)
...         return result
...     return wrapped
...
>>> def func():
...     """ajksnd"""
...
>>> @mydecorator
... def func():
...     """my docstring"""
...     print("hello")
...
>>> func.__doc__
>>>
>>> from functools import wraps
>>> def mydecorator(function):
...     @wraps(function)  # adding this line
...     # ... rest the same ...
...
>>> @mydecorator
... def func():
...     """should be preserved now"""
...     print("hello")
...
>>> func.__doc__
'should be preserved now'
```

Explanation

When writing decorators, it's best practice to use `functools.wraps` to not lose the docstring and other metadata of the function you are decorating.

Resources

https://docs.python.org/3/library/functools.html#functools.wraps
https://pybit.es/decorators-by-example.html

98. Django Extensions (PyPI)

Save time preloading all your models using `shell_plus`:

```
(venv) $ pip install django-extensions ipython

# settings/local.py
INSTALLED_APPS = INSTALLED_APPS + [
    'django_extensions',
]

(venv) $ python manage.py shell_plus --ipython
# Shell Plus Model Imports
from books.models import Badge, Book, BookNote, Search, UserBook
from django.contrib.admin.models import LogEntry
...
from goal.models import Goal
from pomodoro.models import Pomodoro
# Shell Plus Django Imports
from django.core.cache import cache
from django.conf import settings
from django.contrib.auth import get_user_model
from django.db import transaction
from django.db.models import Avg, Case, Count, F, Max, Min, Prefetch, Q, Sum,
When
...
In [1]: Book  # already imported
Out[1]: books.models.Book
In [2]: Book.objects.count()
Out[2]: 2
In [3]: Book.objects.last()
Out[3]: <Book: 2 1ZxxDwAAQBAJ Think & Grow Rich>
```

Explanation

Django Extensions (`django-extensions`) comes with some really powerful tools, for example `shell_plus`, which autoloads all your Django models and ORM helpers in a shell which can be IPython.

Resources

https://django-extensions.readthedocs.io/en/latest/shell_plus.html

99. What day of the week is it?

If you want the day of the week as an integer, call `weekday` on a `date` object:

```
>>> from datetime import date
>>> help(date.today().weekday)

Help on built-in function weekday:

weekday(...) method of datetime.date instance
    Return the day of the week represented by the date.
    Monday == 0 ... Sunday == 6



# using it for a Django command for example

from django.core.management.base import BaseCommand

IS_MONDAY = date.today().weekday() == 0



class Command(BaseCommand):

    def handle(self, *args, **options):
        # logic for every day

        if IS_MONDAY:
            # additional logic only to be run on Monday ...
```

Explanation

We use this often in our Django commands to run them only on particular days.

For example our 4 weekly tip emails we use match `WEEKDAY = date.today().weekday()` against these constants: `MONDAY, TUESDAY, THURSDAY, FRIDAY = 0, 1, 3, 4`, very useful.

Resources

https://docs.python.org/3/library/datetime.html#datetime.date.weekday

Exercise

Bite 74. What day of the week were you born on?

100. What week of the year is it?

If you want the week of the year, call `isocalendar` on a `date` object:

```
>>> from datetime import date
>>> date.today().isocalendar()[1]
30
>>> date(2020, 1, 3).isocalendar()[1]
1
>>> date(2020, 12, 25).isocalendar()[1]
52

# get a range of remaining weeks:
>>> start_week = date.today().isocalendar()[1]
>>> start_week
45
>>> weeks = range(start_week, 53)
>>> list(weeks)
[45, 46, 47, 48, 49, 50, 51, 52]
```

Explanation

`datetime.date.isocalendar` returns a named tuple object with three components: year, week and weekday. Here we just want the week, so we grab the second item (`[1]`).

In the second snippet we use this with `range` to get the remaining weeks of this year.

Resources

https://docs.python.org/3/library/datetime.html#datetime.date.isocalendar

101. Properties, computed fields

Python makes it easy to create read-only properties using the property decorator:

```
>>> from datetime import datetime, timedelta

>>> class Promo:
...     def __init__(self, name, expires=None):
...         self.name = name
...         self.expires = expires or datetime.now()
...     @property
...     def expired(self):
...         return datetime.now() > self.expires
...

>>> promo = Promo('Halloween', expires=datetime.now() + timedelta(seconds=5))
>>> promo.expired
False
# wait 5 seconds
>>> promo.expired
True
```

Explanation

An elegant way to define computed attributes in Python is by using the `@property` decorator (similar to getters and setters in other languages).

These are attributes that are not actually stored, but computed on demand.

In the code above, we use it to see if a promotion has expired by calling the `expired` property on the object. Pretty neat, no?

Resources

https://docs.python.org/3/library/functions.html#property
https://pybit.es/property-decorator.html

102. Ternary operator

Python's ternary operator lets you do an if/else on a single line:

```
>>> my_list = []
>>> my_list[0] if my_list else None
>>> ret = my_list[0] if my_list else None
>>> ret is None
True
>>> my_list = [1, 2, 3]
>>> ret = my_list[0] if my_list else None
>>> ret
1
```

Explanation

If you just want to do something simple as setting a variable in Python, you can reduce a long if/else (4 lines of code) to a one liner conditional expression.

Not everybody likes this construct though, because it contains two branches on a single line of code, which testing tools like `coverage` might not flag as requiring two tests.

Resources

https://docs.python.org/3/reference/expressions.html#conditional-expressions

103. Method chaining

You can chain methods in Python by having them return self:

```
>>> class Calculator:
...
...     def __init__(self, number):
...         self.number = number
...
...     def __str__(self):
...         return f"Number = {self.number}"
...
...     @property
...     def half(self):
...         self.number /= 2
...         return self
...
...     @property
...     def double(self):
...         self.number *= 2
...         return self
...

>>> c = Calculator(10)
>>> print(c)
Number = 10
>>> print(c.double.double)
Number = 40
>>> print(c.double.half.half)
Number = 20.0
```

Explanation

As elegant as it reads (libraries like `pandas` and `sqlalchemy` use method chaining), do keep in mind that it can be more difficult to debug methods that are chained.

Resources

https://stackoverflow.com/a/41817688

104. Create dictionaries with zip

Let's map the Intro Bite ids 101-110 to the more sensible numbers 01-10:

```
>>> bites = range(101, 111)
>>> dict(
...     zip(
...         bites, (str(val)[1:] for val in bites)
...     )
... )
{101: '01', 102: '02', 103: '03', 104: '04', 105: '05',
 106: '06', 107: '07', 108: '08', 109: '09', 110: '10'}
```

Explanation

Here we use the `dict` constructor which can receive a list of 2 element tuples intertwined together by the `zip` built-in.

The `[1:]` slice just strips off the first character of the converted string, so `101` -> `01`.

Again, this is code in use on our platform today :)

Resources

https://docs.python.org/3/library/functions.html#func-dict

105. Make a class callable

Adding `__call__` to your class makes it callable:

```
MAX_GUESSES = 5



class GuessingGame:

    def __init__(self, max_guesses=MAX_GUESSES):
        self.guesses = []
        self.max_guesses = max_guesses
        ...

    def guess(self):
        ...

    def __call__(self):
        """Entry point to the game"""
        while len(self.guesses) < self.max_guesses:
            ...

if __name__ == '__main__':
    game = GuessingGame()
    game()  # __call__ is invoked
```

Explanation

The `__call__` special (magic) method lets you write Python classes where the instances are "callable", that is they can be called like functions.

Here we use it as an entry point to a guessing game class.

Resources

https://docs.python.org/3/reference/datamodel.html#object.__call__
https://dbader.org/blog/python-dunder-methods

106. Get close matches

`difflib.get_close_matches` returns a list of the best “good enough” matches:

```
>>> from difflib import get_close_matches
>>> names = ['julian', 'pybites', 'bob', 'tim', 'python', 'sara', 'james',
'ana']
>>> get_close_matches('pythonista', names)
['python']
>>> get_close_matches('pybit', names)
['pybites']
>>> get_close_matches('jul', names)
['julian']
>>> get_close_matches('ara', names)
['sara', 'ana']
```

Explanation

`difflib` is yet another standard library gem.

Interestingly, Django’s manage.py adopted this command (PR linked below) which you can see in action by typing:

$ python manage.py run-server
Unknown command: 'run-server'. Did you mean runserver?

Resources

https://docs.python.org/3/library/difflib.html#difflib.get_close_matches
https://github.com/django/django/pull/9703/files

107. A dunder shortcut

Write 2 comparison magic methods, get 5 in return, thanks to `functools.total_ordering`:

```
from functools import total_ordering

@total_ordering
class Account:
    ...
    def __eq__(self, other):
        return self.balance == other.balance

    def __lt__(self, other):
        return self.balance < other.balance

>>> acc = Account()
>>> acc2 = Account()
... account activity ...

# all comparison operators work
>>> acc2 > acc
True

>>> acc2 < acc
False

>>> acc == acc2
False
```

Explanation

To not have to implement all of the Python comparison dunder (magic) methods, you can use the `functools.total_ordering` decorator.

It allows you to take a shortcut implementing `__eq__()` together with just one of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`.

Resources

https://docs.python.org/3/library/functools.html#functools.total_ordering
https://dbader.org/blog/python-dunder-methods

108. Date range generator

Here is a generator that returns weekly Monday-Sunday date ranges:

```
>>> from pprint import pprint as pp
>>> from datetime import date, timedelta

>>> def get_week_dates(start_date, num_weeks=10):
...     if start_date.weekday() != 0:
...         raise ValueError("Start date needs to be a Monday")
...
...     for i in range(num_weeks):
...         start = start_date + timedelta(days=i*7)
...         end = start + timedelta(days=6)
...         yield start, end
...
# oops it's Wed now
>>> pp(list(get_week_dates(date.today())))
...
ValueError: Start date needs to be a Monday

# so let's go 2 days back
>>> mon = date.today() - timedelta(days=2)
>>> pp(list(get_week_dates(mon)))
[(datetime.date(2020, 11, 16), datetime.date(2020, 11, 22)),
 (datetime.date(2020, 11, 23), datetime.date(2020, 11, 29)),
...
```

Explanation

`get_week_dates` receives a start `datetime` and an `int` for the number of weeks.

First we check if the start date is a Monday (`weekday` = 0), if not we raise a `ValueError` (it's common to do these pre-checks / throw an exception early in a function).

Then we loop over the weeks building up (start, end) date tuples. The `yield` keyword turns this into a generator.

We used this code as part of creating our WINS.md file so we can document our wins every week.

Resources

https://pybit.es/generators.html
https://docs.python.org/3.8/library/datetime.html#timedelta-objectspy

109. Robust sorting with a function

Sort a list of words, putting words with a digit at the end:

```
>>> ll =  ['Hello1', 'pyth0n', 'coding']
>>> sorted(ll)
['Hello1', 'coding', 'pyth0n']

>>> sorted(ll, key=lambda x: any(s.isdigit() for s in x))
['coding', 'Hello1', 'pyth0n']

# What if we also want to sort the digits? Use a helper:
>>> import re
>>> def get_digit(s):
...     m = re.search(r"(\d+)", s)
...     return int(m.group()) if m else -1
...

>>> ll = ['python', 'py4you', 'coding', 'pyth0n', 'Hello3', 'hello1']
>>> sorted(ll, key=get_digit)
['python', 'coding', 'pyth0n', 'hello1', 'Hello3', 'py4you']
```

Explanation

As we've seen before `sorted` has an optional `key` argument that takes a callable.

For simple "digit last" sorting the inline `lambda` (anonymous function) works just fine.

But to also sort the digits, things get a bit more complex so we move to using a helper function.

In this `get_digit` helper we match the digits returning them (casted to `int`) or `-1` if there are none.

This makes that words with digits go after words without digits, and the words that contain digits are sorted by this digit. We hope you agree this is pretty powerful!

Resources

https://docs.python.org/3/library/functions.html#sorted

110. Pluralizing words

Here are 2 ways to pluralize words:

```
>>> for num_games in (0, 1, 2, 10):
...     num_games, f"game{'' if num_games == 1 else 's'}"
...
(0, 'games')
(1, 'game')
(2, 'games')
(10, 'games')

>>> from gettext import ngettext
>>> for num_games in (0, 1, 2, 10):
...     num_games, ngettext("game", "games", num_games)
...
(0, 'games')
(1, 'game')
(2, 'games')
(10, 'games')
```

Explanation

1. Use an f-string that embeds expressions, so here we use a ternary statement.

2. Use `gettext.ngettext` which takes the `singular` and `plural` words, and the number `n` to apply it to (the `gettext` module provides internationalization (I18N) and localization (L10N) services).

Resources

https://www.python.org/dev/peps/pep-0498/
https://docs.python.org/3/library/gettext.html#gettext.ngettext

111. The str.isalpha method

Here we loop through a string of characters filtering out non-letters:

```
>>> text = 'a 😀 b C é 2 㑧'
>>> [(c, c.isalpha()) for c in text.split()]
[('a', True), ('😀', False), ('b', True), ('C', True), ('é', True), ('2',
False), ('㑧', True)]
```

Explanation

We discovered the `str`'s `isalpha` method the other day which is useful to filter out any non-alphabetic letters.

As per the docs, *alphabetic characters* are those characters defined in the Unicode character database as “Letter”.

Resources

https://docs.python.org/3/library/stdtypes.html#str.isalpha

112. The dictionary's setdefault method

Python `dict`s have a useful method called `setdefault` that can retrieve or set a value in one statement:

```
>>> workouts = {
...     "Mon": "upper body 1",
...     "Tue": "lower body 1",
... }
>>> workouts.setdefault("Mon", "new workout")
'upper body 1'
>>> workouts
{'Mon': 'upper body 1', 'Tue': 'lower body 1'}
>>> workouts.setdefault("Wed", "new workout")
'new workout'
>>> workouts
{'Mon': 'upper body 1', 'Tue': 'lower body 1', 'Wed': 'new workout'}
```

Explanation

The `setdefault` method returns the value of a key if it exists, otherwise it inserts the key, value pair.

Resources

https://docs.python.org/3/library/stdtypes.html#dict.setdefault

113. Debugging Django templates (PyPI)

Why is our Django registration form (plugin) not showing any validation errors? Let's debug inserting a breakpoint into the template:

```
# [your_app]/templatetags/tags.py

from django import template
register = template.Library()

@register.simple_tag(takes_context=True)
def set_breakpoint(context):
    breakpoint()


# in template
{% load tags %}
...
{% set_breakpoint %}
{% for error in form.non_field_errors %}
  {{error}}
{% endfor %}


# inspecting form in pdb ... ok I need to look at errors, not only
non_field_errors

(Pdb) vars(context)['dicts'][3]['form'].errors.as_ul()
'<ul class="errorlist"><li>password2<ul class="errorlist"><li>The two password
fields didn&#39;t match.</li></ul></li></ul>'
```

Explanation

This is a real world scenario where we could not figure out why a form was not displaying errors.

You cannot set a breakpoint directly in the template, but what you can do is write a template tag (here `set_breakpoint`) and use that instead.

Now debugging became pretty easy, it turned out that we were referencing the wrong Django form attribute :)

Resources

https://stackoverflow.com/a/63761743

114. Dictionary union operators

Python 3.9 adds union operators to dicts making dict merging more readable:

```
>>> dict1 = {'a': 10, 'b': 5, 'c': 3}
>>> dict2 = {'d': 6, 'c': 4, 'b': 8}

# not the "obvious way" to merge two dicts
>>> {**dict1, **dict2}
{'a': 10, 'b': 8, 'c': 4, 'd': 6}

# 3.9 now supports union operators
# returning new dict
>>> dict1 | dict2
{'a': 10, 'b': 8, 'c': 4, 'd': 6}
>>> dict2 | dict1
{'d': 6, 'c': 3, 'b': 5, 'a': 10}

# or modifying in-place
>>> dict1
{'a': 10, 'b': 5, 'c': 3}
>>> dict1 |= dict2
>>> dict1
{'a': 10, 'b': 8, 'c': 4, 'd': 6}
```

Explanation

Much cleaner!

One thing to note, as per the PEP's rationale, is that key conflicts will be resolved by keeping the rightmost value. This matches the existing behavior of similar dict operations, where the last seen value always wins.

Resources

https://www.python.org/dev/peps/pep-0584/

115. Removesuffix and removeprefix

Python 3.9 offers two useful new string methods:

```
>>> "this is PyBites".strip("PyBites")
'his is '  # oops
>>> "this is PyBites".removesuffix("PyBites")
'this is '
>>> "pybites shares tips".strip("pybites")
' shares '  # oops
>>> "pybites shares tips".removeprefix("pybites")
' shares tips'
```

Explanation

Sometimes `strip` is a too greedy or does not do what developers expect. Again the PEP's rationale is insightful:

There have been repeated issues ... related to user confusion about the existing `str.lstrip` and `str.rstrip` methods. These users are typically expecting the behavior of `removeprefix` and `removesuffix`, but they are surprised that the parameter for `lstrip` is interpreted as a set of characters, not a substring.

... As another testimonial for the usefulness of these methods, several users on Python-Ideas [2] reported frequently including similar functions in their code for productivity. The implementation often contained subtle mistakes regarding the handling of the empty string, so a well-tested built-in method would be useful.

So use them to make your code more concise and potentially more reliable as well.

Resources

https://docs.python.org/3/library/stdtypes.html#str.removeprefix
https://www.python.org/dev/peps/pep-0616/

116. Timezones are now in the standard library

Starting Python 3.9 we can use them via the `zoneinfo` module:

```
>>> from datetime import datetime
>>> from zoneinfo import ZoneInfo, available_timezones

>>> aus, cet = [tz for tz in available_timezones() if 'Sydney' in tz or
'Madrid' in tz]
>>> aus
'Australia/Sydney'
>>> cet
'Europe/Madrid'

# hours Australia ahead of Spain
>>> def dt(year, month, tz):
...     return datetime(year, month, 1, 0, 0, tzinfo=ZoneInfo(tz))

>>> for month in range(1, 13):
...     delta = dt(2021, month, aus) - dt(2021, month, cet)
...     diff_hours = int(abs(delta.total_seconds()) / 3600)
...     print(f"Month {month} => {diff_hours} hours")
...
Month 1 => 10 hours
Month 2 => 10 hours
Month 3 => 10 hours
Month 4 => 9 hours
Month 5 => 8 hours
Month 6 => 8 hours
…
```

Explanation

Starting Python 3.9 you can make a time zone aware `datetime` (before they were "naive") passing in a `ZoneInfo` object via its new `tzinfo` keyword arg.

Here we use it to figure out the ever changing time difference between Australia vs Spain.

Resources

https://docs.python.org/3/library/zoneinfo.html
https://www.python.org/dev/peps/pep-0615/

117. Type annotations built-ins support

The `list` built-in is now part of type annotations:

```
$ python3.8
>>> def greet_all(names: list[str]) -> None:
...     for name in names:
...         print("Hello", name)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'type' object is not subscriptable
>>> from typing import List
>>> def greet_all(names: List[str]) -> None:
...     pass
...
>>> ^D
$ python3.9
>>> def greet_all(names: list[str]) -> None:
...     for name in names:
...         print("Hello", name)
...
>>>
```

Explanation

Since Python 3.9 type annotations now support using the built-in collection types (`list` / `dict`) as generic types instead of importing the corresponding capitalized types (e.g. `List` or `Dict`) from typing.

PEP rationale: this change removes the necessity for a parallel type hierarchy in the typing module, making it easier for users to annotate their programs and easier for teachers to teach Python.

Resources

https://docs.python.org/3.9/whatsnew/3.9.html#type-hinting-generics-in-standard-collections
https://www.python.org/dev/peps/pep-0585/

118. Decorators support expressions now

PEP 614 (>= Python 3.9) brought relaxing grammar restrictions on decorators:

```
buttons = [QPushButton(f'Button {i}') for i in range(10)]

# old way of working with multiple decorators
button_0 = buttons[0]

@button_0.clicked.connect
def spam():
    ...

button_1 = buttons[1]

@button_1.clicked.connect
def eggs():
    ...

# now you can use any valid expression, including dictionary access
@buttons[0].clicked.connect
def spam():
    ...

@buttons[1].clicked.connect
def eggs():
    ...
```

Explanation

This example we took from this PEP. So decorators now support expressions, going from this grammar:

```
decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
```

to:

```
decorator: '@' namedexpr_test NEWLINE
```

We have not found a use case ourselves yet, but it's a useful feature to know about.

Resources

https://www.python.org/dev/peps/pep-0614/

119. Install a project in editable mode

`pip install` has an `-e` or `--editable` mode:

```
# hm… my package is not in my path
>>> import sys
>>> [p for p in sys.path if p.endswith('pbreadinglist')]
[]

# creating a minimal setup.py (this is not enough for PyPI of course)
(venv) $ more setup.py
import setuptools
setuptools.setup(name='pbreadinglist', version='1.0')
# now I can install it in editable mode
(venv) $ pip install --editable .
# alternatively add this to your requirements.txt
-e .
...

# now I got this in my virtual environment
$ more venv/lib/python3.8/site-packages/pbreadinglist.egg-link
/Users/bobbelderbos/code/pbreadinglist
.
# and voila, my current directory is in my path now
>>> import sys
>>> [p for p in sys.path if p.endswith('pbreadinglist')]
['/Users/bobbelderbos/code/pbreadinglist']
```

Explanation

The easiest way to have your code available on the Python path when using a virtual environment and `pip` is to have a `setup.py` file and install your project in "editable mode" when developing.

Credit: we picked this up when working with the excellent `pytest-django` plugin (link below).

Resources

https://pip.pypa.io/en/stable/reference/pip_install/#install-editable
https://pytest-django.readthedocs.io/en/latest/managing_python_path.html

120. F-strings debugging

Starting Python 3.8 f-strings make it easier to debug / print your variables:

```
$ python3.7

>>> a = 1
>>> b = 'julian'
>>> c = [2, 3, 4]
>>> f"{a=} {b=} {c=}"
  File "<fstring>", line 1
    (a=)
      ^
SyntaxError: invalid syntax

$ python3.8

>>> a = 1
>>> b = 'julian'
>>> c = [2, 3, 4]
>>> f"{a=} {b=} {c=}"
"a=1 b='julian' c=[2, 3, 4]"
```

Explanation

As per docs: an `=` specifier was added to f-strings. An f-string such as f'{expr=}' will expand to the text of the expression, an equal sign, then the representation of the evaluated expression.

Resources

https://docs.python.org/3/whatsnew/3.8.html#f-strings-support-for-self-documenting-expressions-and-debugging
https://docs.python.org/3/glossary.html#term-f-string

121. Download a file (PyPI)

Here are 2 ways to download a file in Python:

```
>>> from urllib.request import urlretrieve

>>> url = ("https://pybites-tips.s3.eu-central-1.amazonaws.com/"
...        "python-download-file.png")

>>> urlretrieve(url)
('/var/folders/r1/p3qtn9t55tvdz8t1l47jhpgc0000gn/T/tmpu2ui82tl',
 <http.client.HTTPMessage object at 0x7fc60a5a1d00>)
>>> urlretrieve(url, 'localfile.png')
('localfile.png', <http.client.HTTPMessage object at 0x7fc60a5a1190>)

>>> import requests
>>> r = requests.get(url)
>>> with open("localfile2.png", "wb") as f:
...     f.write(r.content)
...
200032
```

Explanation

`urlretrieve` is part of the standard library and actually does the job pretty well.

However if you want to do everything with `requests` it's pretty straightforward as well.

Both work here, but as we've seen in tip #39, once you need headers `urlretrieve` might not work, so defaulting to `requests` is usually a good decision.

Resources

https://stackoverflow.com/a/34957875

122. Extract zipfile in memory

`Zipfile` supports reading content without writing to disk:

```
>>> from zipfile import Zipfile
>>> my_zip = ZipFile('pybites_bite306.zip')

>>> my_zip.namelist()
['README.md', 'bite.html', 'translate_cds.py', 'test_translate_cds.py',
'git.txt']

>>> readme = my_zip.namelist()[0]
>>> readme
'README.md'

>>> my_zip.read(readme)
b'## [Bite 306. Translate coding sequences to proteins] ...
```

Explanation

`Zipfile`'s `extractall` extracts all members from the archive to the current working directory.

If you don't want that, you can use a combination of `namelist` and `read` to check the zipfile's content in memory, as we learned from below Stack Overflow answer.

Resources

https://stackoverflow.com/a/10909016
https://docs.python.org/3/library/zipfile.html#zipfile.ZipFile.namelist

123. Concurrent file downloads

Speed up your downloads significantly using `concurrent.futures`:

```
# full code: https://gist.github.com/pybites/6a15bfe006057b6d82e85b4fd1240beb
import concurrent.futures
import os
import requests

def _download_page(url):
    fname = os.path.basename(url)
    r = requests.get(url)
    with open(f'downloads/{fname}', 'wb') as outfile:
        outfile.write(r.content)

def download_urls_sequentially(urls):
    for url in urls:
        _download_page(url)

def download_urls_concurrently(urls):
    with concurrent.futures.ThreadPoolExecutor(max_workers=32) as executor:
        future_to_url = {executor.submit(_download_page, url): url
                         for url in urls}
        for future in concurrent.futures.as_completed(future_to_url):
            future_to_url[future]

$ time python dl.py -s
real    0m51.322s

$ time python dl.py -c
real    0m2.878s
```

Explanation

Here we adapted the `ThreadPoolExecutor` example from the `concurrent.futures` docs to speed up downloading 200+ PyBites articles. The time saving is significant!

Resources

https://docs.python.org/3/library/concurrent.futures.html#threadpoolexecutor-example
https://gist.github.com/pybites/6a15bfe006057b6d82e85b4fd1240beb

124. Timing a function

You can use Python's `timeit` module to time your code.

```
# dl.py
...
from timeit import timeit

URLS = 'urls'

# code from previous tip #123

if __name__ == '__main__':
    with open(URLS) as f:
        urls = [u.rstrip() for u in f.readlines()]
        funcs = 'download_urls_sequentially, download_urls_concurrently'
        for func in funcs.split(', '):
            print(func)
            print(timeit(f"{func}(urls)",
                         f"from __main__ import {funcs}, urls",
                         number=1))

$ python dl.py
download_urls_sequentially
53.683453743
download_urls_concurrently
2.6830952339999996
```

Explanation

Here we use the `timeit` module to time the performance of `download_urls_sequentially` and `download_urls_concurrently` (see previous tip #123).

Note that first we give `timeit` the function to run with its arguments. Then we give it the second `setup` argument which imports the function and the `urls` variable into the current namespace. Lastly we specify the number of executions.

The complete function signature is: `timeit.timeit(stmt='pass', setup='pass', timer=, number=1000000, globals=None)`.

Resources

https://docs.python.org/3/library/timeit.html#timeit.timeit
https://stackoverflow.com/a/19010935

125. Tqdm progress bar (PyPI)

`tqdm` lets you create a fast progress meter:

```
import concurrent.futures
import os

import requests
from tqdm import tqdm

def _download_page(url):
    …

def download_urls_concurrently(urls):
    with concurrent.futures.ThreadPoolExecutor(max_workers=32) as executor:
        future_to_url = {executor.submit(_download_page, url): url
                         for url in urls}
        for future in concurrent.futures.as_completed(future_to_url):
            # make this a generator for tqdm
            yield future_to_url[future]

if __name__ == '__main__':
    with open(URLS) as f:
        urls = [u.rstrip() for u in f.readlines()]

        # all you need to do is wrap an iterable with tqdm, sweet!
        for _ in tqdm(download_urls_concurrently(urls), total=len(urls)):
            pass

$ python dl.py
100%|██████████████████████| 228/228 [00:02<00:00, 83.07it/s]
```

Explanation

Here we use it to show the progress of downloading blog posts from our blog (as introduced in the previous 2 tips #123 and #124).

Fun fact: `tqdm` == Spanish for "te quiero demasiado" or "I love you too much".

Resources

https://github.com/tqdm/tqdm

126. Convert bytes to str

You can use the decode method to turn a Python byte-string into a regular string:

```
(Pdb) expected
'<a href="/books/nneBa6-mWfgC">Coders at Work</a>'
(Pdb) expected in response.content
*** TypeError: a bytes-like object is required, not 'str'
(Pdb) type(response.content)
<class 'bytes'>
(Pdb) type(response.content.decode())
<class 'str'>
(Pdb) expected in response.content.decode()
True
```

Explanation

Another real world scenario. We hit this while writing some tests for our PyBites Reading List app (linked below).

The `response.content`, as returned from `pytest-django`'s `client.get('/')`, comes back as `bytes`, but our `expected` is a `str`. The `in` check between those two types is incompatible.

To solve this we converted `response.content` to a `str`.

The other way around, comparing `bytes` against `bytes`, would work too: `str.encode(expected) in response.content`.

Resources

https://stackoverflow.com/a/606205
https://github.com/PyBites-Open-Source/pbreadinglist

127. The advantage of isinstance

How to compare types in Python?

```
# let's subclass a list, a better way:
# https://treyhunner.com/2019/04/why-you-shouldnt-inherit-from-list-and-dict-in-python/

>>> from collections import UserList
>>> class MyList(UserList):
...     pass
...
>>> my_list = MyList()
>>> MyList.__mro__
(<class '__main__.MyList'>, <class 'collections.UserList'>,
<class 'collections.abc.MutableSequence'>, <class 'collections.abc.Sequence'>,
<class 'collections.abc.Reversible'>, <class 'collections.abc.Collection'>,
<class 'collections.abc.Sized'>, <class 'collections.abc.Iterable'>,
<class 'collections.abc.Container'>, <class 'object'>)

# isinstance = faster and considers inheritance
>>> type(my_list)
<class '__main__.MyList'>
>>> from collections.abc import Sequence
>>> type(my_list) is Sequence
False
>>> isinstance(my_list, Sequence)
True
>>> isinstance(my_list, object)
True
```

Explanation

If you use `isinstance` (over `type`), it takes inheritance into account too (see resource below).

Two other things to notice in the code above:
1. The `collections` module provides wrappers around objects for easier subclassing.
2. A useful magic (or "dunder") method is `__mro__` (which stands for "Method Resolution Order") which lets you see a class' inheritance hierarchy.

Resources

https://switowski.com/blog/type-vs-isinstance
https://stackoverflow.com/questions/2010692/what-does-mro-do/

128. Printable characters

Which (ASCII) characters are considered printable?

```
>>> from string import (printable, digits, ascii_letters,
...                     punctuation, whitespace)
>>> digits
'0123456789'
>>> ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> whitespace
' \t\n\r\x0b\x0c'
>>> for line in wrap(printable, width=60): print(line)
...
0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX
YZ!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
>>> digits + ascii_letters + punctuation + whitespace == printable
True
```

Explanation

In Python you can use the `string` module's `printable` to see if characters are printable.

This is a combination of `digits`, `ascii_letters`, `punctuation`, and `whitespace`.

Resources

https://docs.python.org/3.8/library/string.html#string.printable

129. Getpass module

Two useful functions of the `getpass` module:

```
>>> from getpass import getpass, getuser
>>> from inspect import getsource
>>> import os
>>> from pprint import pprint as pp

>>> username = input('Enter your username: ')
Enter your username: bob
>>> password = getpass('Enter your password: ')
Enter your password:
>>> username, password
'bob', 'password'

>>> getuser()
'bobbelderbos'
>>> os.environ['LOGNAME']
'bobbelderbos'

>>> pp(getsource(getuser))
...
 "    for name in ('LOGNAME', 'USER', 'LNAME', 'USERNAME'):\n"
 '        user = os.environ.get(name)\n'
 '        if user:\n'
 '            return user\n'
...
```

Explanation

Here we ask for a user and password. As we type the user it is visible in the terminal.

For the password though we use the `getpass` function (note the module is also called `getpass`) instead of `input` which prompts the user for a password without echoing.

Another cool function of the `getpass` module is the `getuser` function that retrieves the logged in username checking various environment variables.

Resources

https://docs.python.org/3/library/getpass.html
https://docs.python.org/3/library/inspect.html#inspect.getsource

130. Fixtures in pytest (PyPI)

The `@pytest.fixture` decorator provides an easy yet powerful way to setup and teardown resources for your tests:

```
# module to test (full code linked below)
class Car:
    def __init__(self, model, year):
        self.model = model
        self.year = year

    def __str__(self):
        return f"Car: {self.model} ({self.year})"
     ...
# test module
import pytest
from cars import Car

# by default fixtures use "function" scope = they run for each test function
(use "module" / "session" scope to persist across tests)

@pytest.fixture
def car():
    return Car("Sierra", 2020)

def test_str(car):  # fixtures are passed in as test function arguments
    assert str(car) == 'Car: Sierra (2020)'
```

Explanation

As per the pytest docs: “fixtures provide a fixed baseline so that tests execute reliably and produce consistent, repeatable, results. Initialization may setup services, state, or other operating environments."

We use them everywhere in our tests for our Bites of Py exercises, and so should you. Check out our article linked below.

Still not convinced? Let’s quote somebody that knows a lot about testing: "... one of the great reasons to use fixtures: to focus the test on what you’re actually testing, not on what you had to do to get ready for the test." - Brian Okken ("Python Testing with pytest")

Resources

https://pybit.es/pytest-fixtures.html
https://gist.github.com/pybites/29e6bd91344b4ff747c2f18daf73d00e

131. A powerful regex repeat qualifier

Here we use the `{}` repeat qualifier to match a PyBites license key:

```
>>> import re

# using [A-Z0-9] instead of \w because that would also match an underscore (_)
>>> pat = re.compile(r'^PB(-[A-Z0-9]{8}){4}$')

>>> pat.match('PB-U8N435EH-PG65PW87-IXPWQG5T-898XSZI4')
<re.Match object; span=(0, 38), match='PB-U8N435EH-PG65PW87-IXPWQG5T-
898XSZI4'>

# ^$ = end-to-end match, so adding an extra space, no match
>>> pat.match('PB-U8N435EH-PG65PW87-IXPWQG5T-898XSZI4 ')

# replacing a character with an underscore, no match
>>> pat.match('PB-U_N435EH-PG65PW87-IXPWQG5T-898XSZI4 ')

# leaving the leading PB- off, no match
>>> pat.match('U8N435EH-PG65PW87-IXPWQG5T-898XSZI4')
```

Explanation

In the regular expression meta language, curly braces are useful to match any number of the previous character class or, as is the case here, grouped character sets.

You can specify a range `{x,y}` which means: at least `x`, and at most `y` repetitions. Or use `{z}`, which means: match exactly `z` repetitions.

So as the example in the docs states: `a/{1,3}b` will match `a/b`, `a//b`, and `a///b`, but it won't match `ab`, which has no slashes, or `a////b`, which has four slashes.

Here our regular expression pattern matches: a literal PB, then 4 times the sub pattern of dash (`-`), followed by 8 characters in the range `A-Z0-9`.

Lastly, the `^` and `$` match the beginning and end of the string, so nothing can come before and after the string pattern (license key).

Resources

https://docs.python.org/3/howto/regex.html#repeating-things

132. Take the randomness out

Let's make `random` predictable using `random.seed`:

```
import random

# make it predictable
>>> random.seed(12)
>>> random.sample([1,2,3,4,5], 2)
[4, 3]
>>> random.sample([1,2,3,4,5], 2)
[5, 3]
>>> random.sample([1,2,3,4,5], 2)
[2, 4]

# same results
>>> random.seed(12)
>>> random.sample([1,2,3,4,5], 2)
[4, 3]
>>> random.sample([1,2,3,4,5], 2)
[5, 3]
>>> random.sample([1,2,3,4,5], 2)
[2, 4]
```

Explanation

Here we use `random.sample` to take 2 random items from the `list` passed in.

Using `random.seed` we can take the randomness out, so the `random.sample` calls after the second `random.seed` produce the same results.

This is useful for testing where you want this predictability.

Resources

https://docs.python.org/3/library/random.html#random.seed
https://stackoverflow.com/a/52343354

133. A powerful parser

What if you have to split by words, but need to keep words together that are in double quotes? Enter the `shlex` module:

```
>>> s = """pybites "code challenges" community "data science" development"""

# split on space won't work
>>> s.split()
['pybites', '"code', 'challenges"', 'community', '"data', 'science"',
'development']

# enter shlex
>>> import shlex
>>> shlex.split(s)
['pybites', 'code challenges', 'community', 'data science', 'development']

# if it was not for shlex
>>> import re
>>> [element.strip('"') for element in re.findall(r'"[^"]*"|[^"\W]+', s)]
['pybites', 'code challenges', 'community', 'data science', 'development']
```

Explanation

This might seem easier than it really is, but in reality solving this problem often leads to complex regular expressions like shown in the code block above.

As you can see `shlex` is ideal for this scenario and it is yet another gem of Python's Standard Library.

Resources

https://docs.python.org/3/library/shlex.html
https://pymotw.com/2/shlex/

134. Only keyword arguments

Here is how you can force keyword arguments:

```
>>> def divide_numbers(*, numerator, denominator):
...     try:
...         return int(numerator)/int(denominator)
...     except ZeroDivisionError:
...         return 0
...

>>> divide_numbers(10, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: divide_numbers() takes 0 positional arguments but 2 were given

>>> divide_numbers(numerator=10, denominator=2)
5.0
```

Explanation

You can enforce "keyword-only" arguments in Python by adding a * in the function arguments, succeeding arguments can only be supplied by keyword:

Resources

https://www.python.org/dev/peps/pep-3102/
https://docs.python.org/3/glossary.html#term-argument

135. Emojis in Python (PyPI)

You want emojis in Python? Check out the `emoji` library:

```
# pip install emoji
>>> import emoji
>>> emoji.emojize(":snake:")
'🐍'
>>> emoji.demojize('👍')
':thumbs_up:'

>>> [emo for name, emo in emoji.EMOJI_UNICODE.items() if 'Spain' in name]
['ES']

>>> from pprint import pprint as pp
>>> pp([(name, emo) for name, emo in emoji.EMOJI_ALIAS_UNICODE.items()
        if 'flag_for' in name])
[(':flag_for_Afghanistan:', 'AF'),
 (':flag_for_Albania:', 'AL'),
...
...
 (':flag_for_Zimbabwe:', 'ZW'),
 (':flag_for_Åland_Islands:', 'AX')]
```

Explanation

There are the `emojize` method to convert a string to emoji and `demojize` to go vice versa.

You can look for emojis using the `EMOJI_UNICODE` and `EMOJI_ALIAS_UNICODE` dictionaries.

Resources

https://github.com/carpedm20/emoji

136. Split on last delimiter

In Python you can use `rpartition` or `rsplit` to split a string starting at the end:

```
>>> domains = ["pybit.es", "codechalleng.es", "shop.pybit.es",
"something.longer.pybit.es"]


# hm this gives me different sized lists
>>> [domain.split(".") for domain in domains]
[['pybit', 'es'], ['codechalleng', 'es'], ['shop', 'pybit', 'es'],
['something', 'longer', 'pybit', 'es']]


# rpartition for the win:
>>> [domain.rpartition(".") for domain in domains]
[('pybit', '.', 'es'), ('codechalleng', '.', 'es'), ('shop.pybit', '.', 'es'),
('something.longer.pybit', '.', 'es')]


# to clean up the dot you can also split from the end
>>> [domain.rsplit(".", 1) for domain in domains]
[['pybit', 'es'], ['codechalleng', 'es'], ['shop.pybit', 'es'],
['something.longer.pybit', 'es']]
```

Explanation

Here is a typical problem where you want to split from the end but only a max number of occurrences.

If you want to include the separator use `rpartition` else use `rsplit` which works the same as `split` but starts at the end.

The trick here is to use the `maxsplit` argument to limit the amount of splits to perform which now returns equal length lists of "(subdomain.)domain" and "country code".

Resources

https://docs.python.org/3/library/stdtypes.html#str.rsplit
https://docs.python.org/3/library/stdtypes.html#str.rpartition

137. When zip truncates

How to combine two iterators of different lengths without losing data? Use `zip_longest` from the `itertools` module:

```
>>> names = 'Julian Bob PyBites Dante Martin Rodolfo'.split()
>>> countries = 'Australia Spain Global Argentina'.split()

# truncation - oops!
>>> list(zip(names, countries))
[('Julian', 'Australia'), ('Bob', 'Spain'), ('PyBites', 'Global'), ('Dante',
'Argentina')]

# adding None to shortest iterator = countries
>>> from itertools import zip_longest
>>> list(zip_longest(names, countries))
[('Julian', 'Australia'), ('Bob', 'Spain'), ('PyBites', 'Global'), ('Dante',
'Argentina'), ('Martin', None), ('Rodolfo', None)]

# same but with a fillvalue other than None
>>> list(zip_longest(names, countries, fillvalue="Worldwide"))
[('Julian', 'Australia'), ('Bob', 'Spain'), ('PyBites', 'Global'), ('Dante',
'Argentina'), ('Martin', 'Worldwide'), ('Rodolfo', 'Worldwide')]
```

Explanation

Once more `itertools` helps us out!

Its `zip_longest` takes one or more iterables and a `fillvalue` (default `None`) which it uses to supplement missing values from the shortest iterator.

Resources

https://docs.python.org/3/library/itertools.html#itertools.zip_longest

Exercise

Bite 64. Fix a truncating zip function

138. Flatten a list of lists - part II

Here is how you can flatten a list of lists recursively:

```
>>> numbers = [[1], [2, 3], [4, [5, 6, [7, 8, [9, [10, 11, 12]]]]]]


>>> from itertools import chain
>>> list(chain.from_iterable(numbers))
[1, 2, 3, 4, [5, 6, [7, 8, [9, [10, 11, 12]]]]]


>>> def flatten(numbers):
...     for num in numbers:
...         if isinstance(num, int):
...             yield num
...         else:
...             yield from flatten(num)
...


>>> flatten(numbers)
<generator object flatten at 0x7fae0b17b510>


>>> list(flatten(numbers))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>> sum(flatten(numbers))
78
```

Explanation

As we've seen in tip #12, `itertools.chain` only flattens one level deep.

Here we wrote a `flatten` function that uses recursion yielding all the numbers (`yield` turns this into a generator).

Always be wary of the "base case" (here: `isinstance(num, int)`) with recursion to not go in an infinite loop.

Resources

https://pybit.es/grokking_algorithms.html
https://pybit.es/generators.html

139. Properties part II

Here is how you can use a `@propery` for validation:

```
>>> class Ninja:
...     def __init__(self):
...         self._score = 0
...
...     @property
...     def score(self):
...         return self._score
...
...     @score.setter
...     def score(self, new_score):
...         if new_score < 0:
...             raise ValueError('Score cannot be negative')
...         self._score = new_score
...
>>> n = Ninja()
>>> n.score
0
>>> n.score = 2
>>> n.score = -1
...
ValueError: Score cannot be negative
>>> n.score
2
```

Explanation

In Python there are no private variables and writing getters and setters for all of them is not the way to go. The Pythonic way to do getters and setters is using the `@property` decorator.

Here we used the `setter` on our `@score` property to do extra validation. This will kick in when we assign a value to the `score` attribute (e.g. `n.score = 2`).

You could also add a `deleter` on your property to add behavior to `del n.score`.

Resources

https://pybit.es/property-decorator.html

140. Skip tests in pytest (PyPI)

Here is how to skip a test based on a condition:

```
# comments.py (code with a syntax error)
def time_printer():
    this line should be commented

# test_comments.py
import pytest

def _can_import():
    try:
        import comments  # noqa F401
        return True
    except IndentationError:
        return False

def test_import_fails_because_not_all_garbage_commented():
    if not _can_import():
        raise pytest.fail(…

@pytest.mark.skipif(not _can_import(), reason="Only run if import works")
def test_output_time_printer_with_time_arg_returns_string(capfd):
    # tests past successful import ...

# nicer output + skip other test
$ pytest [output truncated]
E           Failed: comments.py raised an IndentationError, did you comment it
properly?
=== 1 failed, 1 skipped in 0.05 seconds ===
```

Explanation

Sometimes you want to skip a test based on the success of another test. In this real world example, will already fail at import time because of an `IndentationError`.

To provide a better error to the end user, we catch that error first with a `try / except` failing `test_import_fails_because_not_all_garbage_commented`. The second test then only runs if there is no error upon importing the comments module.

Resources

https://docs.pytest.org/en/latest/skipping.html

141. What is included in the standard library? (PyPI)

See what modules are part of the standard library using the Python Standard Library List (`stdlib-list`) module:

```
>>> from stdlib_list import stdlib_list
>>> libs_39 = stdlib_list("3.9")
>>> len(libs_39)
334

>>> libs_39[:5]
['__future__', '__main__', '_thread', 'abc', 'aifc']
>>> libs_39[-5:]
['zipapp', 'zipfile', 'zipimport', 'zlib', 'zoneinfo']

>>> libs_38 = stdlib_list("3.8")
>>> set(libs_39) - set(libs_38)
{'test.support.bytecode_helper', 'zoneinfo', 'graphlib',
'test.support.socket_helper'}

>>> libs_37 = stdlib_list("3.7")
>>> libs_36 = stdlib_list("3.6")
>>> 'dataclasses' in (set(libs_37) - set(libs_36))
True
```

Explanation

Here we use the `stdlib_list` library to get all the standard library modules in a particular Python version.

Comparing versions (using a set operation), we can even see what got added.

We need this to create our module index at the end of this book. For our first edition we coded this manually. From now on though, we definitely will use this library, because it - as any good library - abstracts a lot of unnecessary complexity away (= less code).

Resources

https://pypi.org/project/stdlib-list/

142. Convert XML to a dictionary (PyPI)

Are you tired of parsing XML? What if you can turn it into a `dict`?

```
# pip install xmltodict
>>> import xmltodict
>>> xml = '''<?xml version="1.0" encoding="UTF-8"?>
... <root response="True">
...   <movie title="The Prestige" year="2006" released="20 Oct 2006"
runtime="130 min" />
...   <movie title="The Dark Knight" year="2008" released="18 Jul 2008"
runtime="152 min" />
...   <movie title="Interstellar" year="2014" released="07 Nov 2014"
runtime="169 min" />
... </root>'''

# convert xml to dict
>>> movies = xmltodict.parse(xml)
>>> type(movies)
<class 'collections.OrderedDict'>

# retrieve titles and years
>>> for m in movies['root']['movie']:
...     print(m['@title'], m['@year'])
...
The Prestige 2006
The Dark Knight 2008
Interstellar 2014
```

Explanation

Not sure about you, but parsing XML leads to a lot of googling and trial and error in the debugger.

`xmltodict` is a nice Python module that makes working with XML feel like you are working with JSON, which is a format we much prefer.

Resources

https://github.com/martinblech/xmltodict

143. Named tuples give you attribute access

We love `namedtuples`, a tuple subclass with named fields:

```
>>> from collections import namedtuple
>>> from datetime import date
>>> Transaction = namedtuple('Transaction', 'giver points date')
>>> tr = Transaction('tim', 5, date.today())
# or: Transaction(giver='tim', points=5, date=date.today())
>>> tr
Transaction(giver='tim', points=5, date=datetime.date(2020, 11, 14))
>>> tr[0]
'tim'
>>> tr.giver
'tim'
>>> tr.points
5
>>> tr.points = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Explanation

`namedtuples` (from the `collections` module) are one of our favorite data structures.

They work like tuples so they are immutable, but instead of indexing you can access the elements by attribute which leads to really elegant code.

Do notice that the lookup by attribute is slightly slower than by index, but usually this is not a concern and the increased readability is much worth it.

Resources

https://docs.python.org/3/library/collections.html#collections.namedtuple
https://stackoverflow.com/a/31200651

144. Beautiful web scraping (PyPI)

Beautiful Soup is an awesome library to scrape web pages:

```
>>> import bs4
>>> import requests

>>> resp = requests.get('https://pybit.es/archives')
>>> soup = bs4.BeautifulSoup(resp.text, 'html.parser')

>>> matches = [a for a in soup.find_all('a', href=True)
...            if 'beautiful' in a.text.lower()]

>>> for m in matches:
...     print(m.text)
...     print(m['href'])
...
Generating Beautiful Code Snippets with Carbon and Selenium
https://pybit.es/python-tips-carbon-selenium.html
Create a Simple Web Scraper with BeautifulSoup4
https://pybit.es/simplewebscraper.html
Beautiful, idiomatic Python
https://pybit.es/beautiful-python.html
```

Explanation

Beautiful Soup is a Python library for pulling data out of HTML and XML files.

Here we use it to find all the articles that have the string “'beautiful” in the link text.

(Always check the conditions of the site you are scraping to see if it’s cool to do so.)

Resources

https://pybit.es/simplewebscraper.html
https://beautiful-soup-4.readthedocs.io/en/latest/

145. Pretty print

Using `pprint` to pretty print a nested data structure:

```
>>> from collections import Counter
>>> import pprint
>>> from itertools import chain

# from a previous tip
>>> articles = [a.text for a in soup.find_all('a', href=True)
...                 if 'https://pybit.es' in a['href']]
# get all words from articles
>>> words = chain(*[art.split() for art in articles])
# get the most common words
>>> most_common = Counter(words).most_common(20)

# print them nicely
>>> pp = pprint.PrettyPrinter(width=40, compact=True)
>>> pp.pprint(most_common)
[('-', 214), ('Code', 141),
 ('Challenge', 122), ('Twitter', 101),
 ('to', 73), ('Python', 70), ('a', 67),
 ('Digest', 60), ('Review', 54),
 ('2017', 52), ('and', 45),
 ('PyBites', 43), ('2018', 39),
 ('of', 37), ('Week', 36),
 ('digest', 36), ('week', 35),
 ('How', 32), ('With', 29),
 ('the', 28)]
```

Explanation

We use `pprint` a lot! Simply do a `from pprint import pprint as pp` and it becomes a lot easier to look at a (nested) data structure. `PrettyPrinter` adds some more options as you can see in the code above.

Note that `pp` is available in `pdb` so that is yet another reason to live and breathe in your debugger :)

Resources

https://docs.python.org/3/library/pprint.html#pprint.PrettyPrinter
https://docs.python.org/3/library/pdb.html#pdbcommand-pp

146. Dataclasses for the win

The `@dataclass` decorator helps reduce boilerplate code when writing your Python classes:

```
>>> from dataclasses import dataclass
>>> @dataclass
... class Bite:
...     number: int
...     title: str
...     level: str = 'Beginner'
...
...     def __post_init__(self):
...         self.title = self.title.capitalize()
...
>>> bite = Bite(1, "sum n numbers")
>>> repr(bite)
"Bite(number=1, title='Sum n numbers', level='Beginner')"
>>> bite.level = 2  # dataclasses are mutable
>>> bite
Bite(number=1, title='Sum n numbers', level=2)
# make an immutable dataclass
>>> @dataclass(frozen=True)
... class Bite:
...     # same but without __post_init__ (dataclasses.FrozenInstanceError)

>>> bite = Bite(1, "sum n numbers")
>>> bite.level = 2
...
dataclasses.FrozenInstanceError: cannot assign to field 'level'
```

Explanation

Dataclasses save you code. They automatically add special methods like `__init__()` and `__repr__()` to your classes. You can default values like we did here with `level`. The `__post_init__` method lets you do some extra construction.

Dataclasses are mutable by default, but passing `frozen=True` as an argument to the decorator, you can make them immutable (aka have a `namedtuple` on steroids). This does mean you can't use `__post_init__`. Another cool feature is `order` which generates comparison *dunders*.

Resources

https://docs.python.org/3/library/dataclasses.html

147. Nested defaultdicts

Ever wondered how to nest `defaultdicts`?

```
# using pdb to have my cars json (source: Mockeroo) loaded into the data list
(Pdb) from collections import Counter, defaultdict
(Pdb) from pprint import pprint as pp
(Pdb) pp data
[{'automaker': 'Dodge', 'id': 1, 'model': 'Ram Van 1500', 'year': 1999},
 {'automaker': 'Chrysler', 'id': 2, 'model': 'Town & Country', 'year': 2002},
 {'automaker': 'Porsche', 'id': 3, 'model': 'Cayenne', 'year': 2008},
... many more records ...

(Pdb) cars = defaultdict(lambda: Counter())  # or:
defaultdict(functools.partial(Counter))
(Pdb) for row in data: cars[row["automaker"]][row["model"]] += 1
(Pdb) pp cars
defaultdict(<function <lambda> at 0x7fe3aaa22f70>,
            {'Acura': Counter({'CL': 3,
                               'NSX': 3,
                               'RL': 3,
                               'Vigor': 2,
        ...
             'Audi': Counter({'A8': 4,
                              'TT': 4,
        ...
                              'Q7': 1,
                              'A3': 1}),
        ...
        ... many more ...
```

Explanation

`defaultdicts` are awesome (see tip #80), but what if we need various levels?

The `defaultdict`' constructor's `default_factory` can be a function which will be used for building new elements. That means that we can introduce nesting by using `lambda` (which is an anonymous / inline function). If you don't like `lambda`, you could also use `functools.partial` which might be a bit more elegant (see linked Stack Overflow answer).

Resources

https://docs.python.org/3/library/collections.html#collections.defaultdict
https://stackoverflow.com/a/5030081

148. Yield from iterable

Here we use `yield from` to create a simple generator:

```
>>> def gen():
...     for i in range(1, 11):
...         yield i
...
>>> g = gen()
>>> next(g)
1
>>> next(g)
2
...
...
>>> next(g)
10
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

# same result, more concise

>>> def gen():
...     yield from range(1, 11)
...

# note that a for loop "catches" the StopIteration for you
>>> [i for i in gen()]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Explanation

`yield` turns a function into a generator. Here our generator produces the numbers 1 to 10.

PEP 380 (Python 3.3) introduced `yield from` which saves you the extra for loop.

Resources

https://pybit.es/generators.html
https://docs.python.org/3/whatsnew/3.3.html#pep-380

149. How to slice a generator?

You can use `itertools.islice` for this purpose:

```
>>> def gen():
...     yield from range(1, 11)
...
>>> g = gen()

>>> g[:2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not subscriptable

>>> from itertools import islice
>>> my_slice = islice(g, 2)
>>> my_slice
<itertools.islice object at 0x7fb2ab084540>
>>> list(my_slice)
[1, 2]
>>> [i for i in g]
[3, 4, 5, 6, 7, 8, 9, 10]

# another example of generator exhaustion:
>>> g = gen()
>>> ', '.join(str(i) for i in g)
'1, 2, 3, 4, 5, 6, 7, 8, 9, 10'
>>> ', '.join(str(i) for i in g)
''
```

Explanation

Pretty cool! One word of caution with generators, they are consumed only once!

As you can see here, once we materialized a slice, you cannot get it twice. So best is to store it into a variable for reuse.

Of course doing `g = gen()` again, you can consume the values again.

Resources

https://docs.python.org/3/library/itertools.html#itertools.islice
https://pybit.es/generators.html

150. Rollback context manager

Here we use a context manager to rollback a transaction based on a condition:

```
>>> class Account:
...     def __init__(self):
...         self._transactions = []
...
...     @property
...     def balance(self):
...         return sum(self._transactions)
...
...     def add_amount(self, amount):
...         self._transactions.append(amount)
...
...     def __enter__(self):
...         self._copy_transactions = list(self._transactions)
...         return self
...
...     def __exit__(self, exc_type, exc_val, exc_tb):
...         if self.balance < 0:
...             self._transactions = self._copy_transactions
...             print("ERROR: transaction causes negative balance, rollback!")
...
>>> acc = Account()
>>> with acc as a:
...     a.add_amount(-20)
...
ERROR: transaction causes negative balance, rollback!
>>> acc.balance
0
```

Explanation

Adding the `__enter__` and `__exit__` (dunder) methods to an object, you make it work as a context manager. The example above is adapted from: Enriching Your Python Classes With Dunder (Magic, Special) Methods (linked below).

Resources

https://dbader.org/blog/python-dunder-methods
https://docs.python.org/3/library/stdtypes.html#typecontextmanager

151. Configuration variables (PyPI)

Use `python-decouple` and `dj-database-url` to manage configuration variables:

```
# pip install python-decouple dj-database-url

from decouple import config, Csv
import dj_database_url

SECRET_KEY = config('SECRET_KEY')
DEBUG = config('DEBUG', default=False, cast=bool)
ALLOWED_HOSTS = config('ALLOWED_HOSTS', cast=Csv())

DATABASES = {
    'default': dj_database_url.config(
        default=config('DATABASE_URL')
    )
}
```

Explanation

The only thing you need to do is keep an `.env` file in your project folder.

We recommend ignoring it (using `.gitignore`) and commit an `.env-example` with empty variables to communicate to other developers what is required for the project.

`python-decouple`'s `config` object (and `dj-database-url`'s `config` method) will read in config variables from that file. Note that environment variables set in the shell will take precedence over the `.env` file.

Resources

https://pypi.org/project/python-decouple/
https://pypi.org/project/dj-database-url/

152. Take a screenshot of a website (PyPI)

You can use the headless chrome/chromium automation library `pyppeteer` for this:

```python
import asyncio
import re
import sys

from pyppeteer import launch


async def create_screenshot(url):
    browser = await launch()
    page = await browser.newPage()
    await page.goto(url)
    file_name = re.sub(r'[^0-9a-z]+', r'-', url.lower())
    await page.screenshot({'path': f'{file_name}.png'})
    await browser.close()


if __name__ == '__main__':
    url = sys.argv[1] if len(sys.argv) > 1 else 'https://pybit.es'
    asyncio.get_event_loop().run_until_complete(
        create_screenshot(url))
```

Explanation

`pyppeteer` is the unofficial Python port of the `puppeteer` JavaScript (headless) chrome/chromium browser automation library.

Here we slightly adapted its docs example to make a screenshot of a given URL from the command line (e.g. `python script.py https://codechalleng.es`)

Resources

https://pypi.org/project/pyppeteer/

153. Fake test data (PyPI)

Use `faker` to generate random test data:

```
# pip install Django && python manage.py shell
In [1]: from django.contrib.auth.models import User
In [2]: from faker import Faker
In [3]: fake = Faker()
# create some fake data
In [4]: first_name, last_name, domain = fake.first_name(), fake.last_name(),
fake.safe_domain_name()
In [5]: first_name, last_name, domain
Out[5]: ('Douglas', 'Tapia', 'example.net')
In [6]: username = f"{first_name.lower()}.{last_name.lower()}"
In [7]: email = f"{username}@{domain}"
# create a Django User object
In [8]: u = User(first_name=first_name, last_name=last_name, email=email,
username=username)
In [9]: u
Out[9]: <User: douglas.tapia>
In [10]: vars(u)
Out[10]:
{'_state': <django.db.models.base.ModelState at 0x7fef04fe23d0>,
 'id': None,
…
 'username': 'douglas.tapia',
 'first_name': 'Douglas',
 'last_name': 'Tapia',
 'email': 'douglas.tapia@example.net',
 …
 'date_joined': datetime.datetime(2020, 11, 25, 7, 38, 44, 52703,
tzinfo=<UTC>)}
```

Explanation

Here we use `Faker()` to create and initialize a faker generator (`fake`).

We can then create a Django `User` object using `fake`'s useful `username`, `email`, etc. methods.

This is very useful to quickly fill a DB with fake data for testing or doing a demo.

Resources

https://faker.readthedocs.io/en/master/index.html

154. Import a module from a string

You can use Python's `importlib.import_module` (or `__import__`) to import a module from a string:

```
>>> import importlib
>>> import inspect

>>> module = importlib.import_module('random')  # or __import__('random')
>>> sample = getattr(module, 'sample')

>>> sample([1, 2, 3, 4, 5], 2)
[1, 3]

>>> inspect.getsource(sample)[:50]
'    def sample(self, population, k, *, counts=None'
```

Explanation

Here we use `importlib` to import `random` from a string. This is a useful technique if we'd want to specify any module form the command line (see gist linked below).

As we've seen in tip 72, we can use `getattr` to get an attribute from a module, here we get the `sample` function from the `random` module this way.

Lastly we get the source code from this function using the `inspect` module (discussed in tips 96 and 129). It blows our minds that all this is readily available in Python's Standard Library!

Resources

https://stackoverflow.com/a/8790232
https://github.com/PyBites-Open-Source/pysource

155. Match word boundaries

Use boundary matching (regex) for more robust text replacements:

```
>>> import re
>>> s = "@P and @PyBites should both work :)"

# oops!
>>> s.replace("@P", "<a href='#P'>{at_name}</a>")
"<a href='#P'>{at_name}</a> and <a href='#P'>{at_name}</a>yBites should both
work :)"

# match right word boundary
>>> s = re.sub(r"@P\b", "<a href='#P'>@P</a>", s)

# good: only @P not replaced
>>> s
"<a href='#P'>@P</a> and @PyBites should both work :)"

# yes, you could do this with one re.sub (\w+), but here we are matching
# exact usernames, one by one
>>> s = re.sub(r"@PyBites\b", "<a href='#PyBites'>@PyBites</a>", s)

# all good
>>> s
"<a href='#P'>@P</a> and <a href='#PyBites'>@PyBites</a> should both work :)"
```

Explanation

Here is another real world scenario / PyBites Platform bug fix. We had to replace an overlapping username and our simple `str.replace` would not suffice.

So if you do a `re.sub` instead you have the opportunity to match the ending word boundary. As you can see in this code snippet, this was the ideal solution here.

Resources

https://docs.python.org/3/library/re.html#regular-expression-syntax

156. Doctest

Did you know that Python allows you to write tests in your docstrings?

```
def greeting(name=None):
    """
    >>> greeting()
    'Hello Stranger!'
    >>> greeting('bob')
    'Hello Bob!'
    """
    name = "Stranger" if name is None else name
    return f"Hello {name.title()}!"


if __name__ == "__main__":
    import doctest
    doctest.testmod()


# no output on a passing test, removing the exclamation point in the expected
output produces:
$ python script.py
**********************************************************************
File "script.py", line 5, in __main__.greeting
Failed example:
    greeting('bob')
Expected:
    'Hello Bob'
Got:
    'Hello Bob!'
**********************************************************************
1 items had failures:
   1 of   2 in __main__.greeting
***Test Failed*** 1 failures.
```

Explanation

As per the docs linked below, the `doctest` module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work as shown.

Resources

https://docs.python.org/3/library/doctest.html

157. Urlretrieve and ZipFile

Download and extract a zipfile in Python in just 2 lines of code (not counting imports):

```
>>> from os import listdir
>>> from zipfile import ZipFile
>>> from urllib.request import urlretrieve

>>> data_zip = "temp.zip"
>>> urlretrieve("https://bites-data.s3.us-east-2.amazonaws.com/311-data.zip",
data_zip)
('temp.zip', <http.client.HTTPMessage object at 0x7f87d55c43d0>)

>>> ZipFile(data_zip).extractall("/tmp/temp")

>>> listdir('/tmp/temp')
['tf_idf.py', 'tf-idf.csv', 'samples.pkl', 'samples.txt', 'stop_words.py']
```

Explanation

First we retrieve a zip file we have stored in an AWS S3 bucket, storing it in `data_zip`.

Then we instantiate a `ZipFile` object - basically loading in this zipfile - and run `extractall` on it which you can give an alternative path.

So this extracts the zip's content in `/tmp/temp`, a subdirectory `ZipFile` creates because it’s not present yet.

Lastly, we use `os.listdir` to list the files in this subdirectory.

Resources

https://docs.python.org/3/library/zipfile.html#zipfile.ZipFile.extractall

158. String module's capwords

Seldom used but here is a good use case for `capwords` (`string` module):

```
>>> s = "Here's food for thought, let's say title casing isn't easy ..."

# oops
>>> s.title()
"Here'S Food For Thought, Let'S Say Title Casing Isn'T Easy ..."

>>> import string
>>> string.capwords(s)
"Here's Food For Thought, Let's Say Title Casing Isn't Easy ..."

# similar to:
>>> s.split()
["Here's", 'food', 'for', 'thought,', "let's", 'say', 'title', 'casing',
"isn't", 'easy', '...']
>>> [w.capitalize() for w in s.split()]
["Here's", 'Food', 'For', 'Thought,', "Let's", 'Say', 'Title', 'Casing',
"Isn't", 'Easy', '...']
>>> ' '.join(w.capitalize() for w in s.split())
"Here's Food For Thought, Let's Say Title Casing Isn't Easy ..."
>>> ' '.join(w.capitalize() for w in s.split()) == string.capwords(s)
True

# use a different string to split on:
>>> string.capwords(s, sep="'")
"Here'S food for thought, let'S say title casing isn'T easy ..."
```

Explanation

In this case the apostrophe makes `str.title` inadequate.

`string.capwords` is smarter about this, because it splits the sting into words, then does a `str.capitalize` on each word, and then joins them back together.

The splitting is done on a whitespace by default, but that can be changed by using the optional `sep` argument, like we did in the last statement.

Resources

https://docs.python.org/3/library/string.html#string.capwords

159. Alternate class constructors

You can use a `@classmethod` to add alternative constructors to your class:

```
>>> from datetime import datetime, date
>>> class MyDate:
...
...     def __init__(self, year, month, day):
...         self.date = date(year, month, day)
...
...     @classmethod
...     def from_str(cls, date_str):
...         try:
...             year, month, day = date_str.split("-")
...             return cls(int(year), int(month), int(day))
...         except ValueError:
...             raise
...
>>> d1 = MyDate(2016, 12, 19)
>>> d1.date
datetime.date(2016, 12, 19)

# an alternative way to instantiate a MyDate object:
>>> d2 = MyDate.from_str("2016-12-19")
>>> d2.date
datetime.date(2016, 12, 19)
>>> d1.date == d2.date
True
```

Explanation

Here we used a `@classmethod` to provide a second way to construct a `MyDate` object, namely by giving it a date string.

We try to unpack it to `year`, `month`, `day` and feed that to `cls`. This will make the same type of object.

Also notice we take the EAFP ("it’s easier to ask for forgiveness than permission") approach (instead of "look before you leap" or LBYL), which is often considered more idiomatic.

Resources

https://docs.python.org/3/library/functions.html#classmethod
https://code-maven.com/slides/python/class-methods-alternative-constructor

160. More robust timedeltas (PyPI)

Use `relativedelta` (`python-dateutil` module) for more specific date calculations:

```
>>> from datetime import date
>>> pybites_born = date(year=2016, month=12, day=19)
>>> today = date.today()
>>> (today-pybites_born).days
1447

# oops
>>> (today-pybites_born).years  # same for .months
...
AttributeError: 'datetime.timedelta' object has no attribute 'years'

>>> from dateutil.relativedelta import relativedelta, MO
>>> diff = relativedelta(today, pybites_born)
>>> diff.years, diff.months
3, 11

# there are many more options:
>>> today
datetime.date(2020, 12, 5)  # Saturday

>>> today + relativedelta(months=+1, weeks=+1, hour=10)
datetime.datetime(2021, 1, 12, 10, 0)

>>> today + relativedelta(weekday=MO)
datetime.date(2020, 12, 7)
```

Explanation

`datetime`'s `timedelta` is pretty awesome, but `python-dateutil`'s `relativedelta` offers many more options.

Here it allows us to define how old PyBites is in years and months at the time of writing this. We then see 2 more examples of being more exact in defining our future / past dates.

Check out the documentation linked below for more examples.

Resources

https://dateutil.readthedocs.io/en/stable/relativedelta.html
https://pybit.es/python-dateutil.html

161. Starts- and endswith

2 more elegant string methods:

```
>>> s = "hello world from pybites"
>>> s[:5] == "hello"  # no need
True
>>> s.startswith("hello")
True
>>> s[-2:] == 'es'
True
# better:
>>> s.endswith('es')
True

# optional args
>>> s.startswith("from")
False
>>> s[12:]
'from pybites'
>>> s.startswith("from", 12)
True
>>> s[:-8]
'hello world from'
>>> s.endswith("from", 0, -8)
True
# tuples are supported too:
>>> t = "hello world from spain"
>>> s.endswith(("pybites", "spain"))
True
>>> t.endswith(("pybites", "spain"))
True
```

Explanation

It's more idiomatic to use these string methods than slicing. It turns out you can use it with tuples and specify start and end positions as well, nice!

Resources

https://docs.python.org/3/library/stdtypes.html#str.startswith
https://docs.python.org/3/library/stdtypes.html#str.endswith

162. Enumerations

In Python you can group constants using the `enum` module:

```
>>> from enum import Enum, IntEnum
>>> class BiteLevel(Enum):
...     INTRO = 1
...     BEGINNER = 2
...     INTERMEDIATE = 3
...     ADVANCED = 4
...
>>> print(repr(BiteLevel.BEGINNER))
<BiteLevel.BEGINNER: 2>
>>> adv = BiteLevel.ADVANCED
>>> adv.name, adv.value
('ADVANCED', 4)
>>> for name, member in BiteLevel.__members__.items(): name, member
...
('INTRO', <BiteLevel.INTRO: 1>)
('BEGINNER', <BiteLevel.BEGINNER: 2>)
...
# Do you want to compare to integers? Use IntEnum
>>> BiteLevel.INTRO == 1
False
>>> class SpecialLevel(IntEnum):
...     ...
>>> SpecialLevel.INTRO == 1
True
```

Explanation

Enumerations are a great way to group constants like we do here with Bite levels.

Each constant `enum` object has `name` and `value` attributes and the enumeration as a whole can be iterated over. For comparing integers, use `IntEnum`.

Resources

https://docs.python.org/3/library/enum.html

Exercise

Bite 82. Define a Score Enum and customize it adding methods

163. Create a temporary directory

Use `tempfile.TemporaryDirectory` to create temporary directories, in `pytest` you can use its `tmp_path` fixture:

```python
# initial code
from pathlib import Path
from tempfile import TemporaryDirectory

# tree.py > https://gist.github.com/pybites/248d6190c27defc3b832ca6cef9ac495
from tree import count_dirs_and_files

def test_only_files():
    with TemporaryDirectory(dir="/tmp") as dirname:
        for i in range(1, 6):
            filename = f'{i}.txt'
            path = Path(dirname) / filename
            with open(path, 'w') as f:
                f.write('hello')
        assert count_dirs_and_files(dirname) == (0, 5)

# refactored using pytest's tmp_path fixture
def test_only_files_refactored(tmp_path):
    for i in range(1, 6):
        path = tmp_path / f'{i}.txt'
        with open(path, 'w') as f:
            f.write('hello')
    assert count_dirs_and_files(tmp_path) == (0, 5)
```

Explanation

`TemporaryDirectory` lets you create a temporary directory which you can use in a context manager. As per the docs, on completion of the context or destruction of the temporary directory object the newly created temporary directory and all its contents are removed.

Our refactoring to use `pytest`'s `tmp_path` reduced the code quite a bit :)

Resources

https://docs.python.org/3/library/tempfile.html
https://docs.pytest.org/en/stable/tmpdir.html

164. How to mute a flake8 E501 (PyPI)?

3 ways to mute certain flake8 errors:

```
# 1. ignore one off violations in-line
MSG = """
Hey {name},
...
<a href="{link}" target="_blank"><img src="{img}" alt="{title}" style="width:
100%;"></a>
...
...
"""  # noqa E501


# 2. ignore from the command line
flake8 --ignore=E501 path/to/files/


# 3. ignore per project (~/.config/flake8 = global, be careful!)
(project_folder) $ more .flake8
[flake8]
ignore = E501
```

Explanation

Use at your own risk / judgement.

Sometimes E501 ("lines no greater than 79 characters") can be annoying so you want to mute them.

You can do it in-line for a single instance or project wide or even global using a `~/.config/flake8`. The more globally muted though, the less obvious, so be cautious with that.

By the way, Vim lovers, you can use this shortcut in your .vimrc to invoke `flake8` when pressing `,f`: `autocmd FileType python map ,f :call Flake8()` (or use Syntastic to run it automatically upon save).

Resources

https://flake8.pycqa.org/en/latest/user/violations.html

165. Smart csv parsing

Does my csv file have a header?

```
>>> import csv

$ cat names-with-header.csv
id,first_name,last_name
1,Bibbye,Wield
2,Hewett,Yushachkov
3,Jory,Broune

$ cat names-without-header.csv
1,Pavia,Soro
2,Marna,Gwatkin
3,Melanie,Gribble

>>> def csv_has_header(file):
...     with open(file, 'r') as csvfile:
...         sniffer = csv.Sniffer()
...         return sniffer.has_header(csvfile.read(2048))
...

>>> csv_has_header('names-with-header.csv')
True
>>> csv_has_header('names-without-header.csv')
False
```

Explanation

What if you are not sure your `csv` file has a header?

It turns out that the `csv` module has a `Sniffer` class that can figure it out for you!

A quick an easy way to get test data like in this example is to use the free Mockaroo service.

Resources

https://docs.python.org/3/library/csv.html#csv.Sniffer
https://stackoverflow.com/a/40193509

Exercise

Bite 79. Parse a csv file and create a bar chart

166. Reading csv files

Use `csv`'s `DictReader` if you want to process rows as dictionaries:

```
>>> import csv
>>> with open('names-with-header.csv') as csvfile:  # see Tip 165
...     reader = csv.reader(csvfile)
...     for row in reader:
...             print(row)
...
['id', 'first_name', 'last_name']
['1', 'Bibbye', 'Wield']
['2', 'Hewett', 'Yushachkov']
['3', 'Jory', 'Broune']

>>> with open('names-with-header.csv') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...             print(row)
...
{'id': '1', 'first_name': 'Bibbye', 'last_name': 'Wield'}
{'id': '2', 'first_name': 'Hewett', 'last_name': 'Yushachkov'}
{'id': '3', 'first_name': 'Jory', 'last_name': 'Broune'}
```

Explanation

You can process the rows in a `csv` file as lists or dictionaries by using `reader` or `DictReader` respectively.

The problem with lists though is that if you have many columns, referring to a column as `row[25]` is not very readable. Compare that to say `row["attribute"]` which is more readable.

In case of `DictReader` you can rely on the order of the columns (in Python 3.6 the rows were of type `OrderedDict`, in 3.8 this was changed to regular `dict`, because since 3.7 dictionary order is guaranteed to be insertion order).

Resources

https://docs.python.org/3/library/csv.html#csv.reader
https://docs.python.org/3/library/csv.html#csv.DictReader

167. Keywords and builtins

You can use the `keyword` and `builtins` modules to see if a string is either:

```
>>> import keyword
>>> import builtins

>>> keyword.kwlist
['False', 'None', 'True', ..., 'with', 'yield']
>>> len(keyword.kwlist)
36

>>> keyword.iskeyword('file')
False
>>> keyword.iskeyword('if')
True
>>> keyword.iskeyword('locals')
False
>>> keyword.iskeyword('nonlocal')
True

>>> 'locals' in dir(builtins)
True
>>> 'property' in dir(builtins)
True

>>> set(dir(builtins)) & set(keyword.kwlist)
{'None', 'False', 'True'}
>>> None = 10
...
SyntaxError: cannot assign to None
```

Explanation

This is pretty self-explanatory. Interestingly `None`, `False` and `True` are both builtins and keywords. As per the linked Stack Overflow thread they are builtins (constants) that are protected from assignment by the parser which prevents accidental overwriting. Keywords don't allow value assignments.

Resources

https://stackoverflow.com/q/8204542

168. Python 2 vs 3 builtins

What Python 2 builtins did not make it to Python3?

```
# the following line is from a python 2.7 REPL session
# if you want to try it you might need to pip install the "future" library
>>> py2_builtins = ['ArithmeticError', ..., 'xrange', 'zip']

>>> import builtins
>>> from pprint import pprint as pp
>>> py3_builtins = dir(builtins)
>>> pp(set(py2_builtins) - set(py3_builtins))
{… 'absolute_import',
 'apply',
 'basestring',
 'buffer',
 'cmp',
 'coerce',
 'execfile',
 'file',
 'intern',
 'long',
 'raw_input',
 'reduce',
 'reload',
 'sys',
 'unichr',
 'unicode',
 'xrange'}
```

Explanation

Here we copied in the output of `dir(builtins)` from Python 2.7 (for which we had to go back to the future, pun intended!) - note this output might differ slightly depending the OS you are on.

Then we used a set operation (see Tip #31) to see which of the Python 2 builtins are no longer in Python 3. We actually discovered this week that `file` is no longer a builtin so no need to use variables like `file_`, `file` does not clash with anything anymore.

Resources

https://docs.python.org/3/library/builtins.html
https://docs.python.org/3.8/library/stdtypes.html#set-types-set-frozenset

169. Print a list to N columns

Here is how to split a list into evenly sized chunks:

```
# credit / thanks to Ned Batchelder
>>> def chunks(lst, n):
...     """Yield successive n-sized chunks from lst."""
...     for i in range(0, len(lst), n):
...         yield lst[i:i + n]
...

# using keyword for a bit shorter output
>>> import keyword
>>> from pprint import pprint as pp
>>> pp(list(chunks(keyword.kwlist, 5)))
[['False', 'None', 'True', '__peg_parser__', 'and'],
 ['as', 'assert', 'async', 'await', 'break'],
 ['class', 'continue', 'def', 'del', 'elif'],
 ['else', 'except', 'finally', 'for', 'from'],
 ['global', 'if', 'import', 'in', 'is'],
 ['lambda', 'nonlocal', 'not', 'or', 'pass'],
 ['raise', 'return', 'try', 'while', 'with'],
 ['yield']]
```

Explanation

In the last tip the output of `dir(builtins)` took up a lot of vertical space which inspired us to see how to use the horizontal space better.

`chunks` is a generator making use of `range`'s third "step" argument (see Tip #4).

Similarly to the `list_reverseiterator` in Tip #23 we need to cast the generator to a `list` to consume it all at once. Then we pretty print it.

Another option is to use `numpy.array_split`.

Resources

https://stackoverflow.com/a/312464
https://stackoverflow.com/a/29679492

170. Pickling your objects

Here is how to persist your objects using the `pickle` module:

```
>>> from collections import namedtuple
>>> from datetime import date
>>> import keyword
>>> import pickle

>>> Transaction = namedtuple('Transaction', 'giver points date')
>>> tr = Transaction(giver='tim', points=5, date=date.today())
>>> class MyClass:
...     def is_keyword(self, name):
...             return keyword.iskeyword(name)
...
>>> mc = MyClass()
>>> mc.is_keyword('None')
True
>>> with open('data.pkl', 'wb') as outfile:  # note the b
...     pickle.dump(tr, outfile)
...     pickle.dump(mc, outfile)
...
>>> with open('data.pkl', 'rb') as infile:  # again note the b
...     tr = pickle.load(infile)
...     mc = pickle.load(infile)
...
>>> tr
Transaction(giver='tim', points=5, date=datetime.date(2020, 12, 17))
>>> mc.is_keyword('None')
True
```

Explanation

Here we use `pickle` to persist a `namedtuple` and even our own little class, really cool! As compared to JSON `pickle` is a binary serialization format, so it's not human-readable. It can represent an extremely large number of Python types. On the flip side though, deserializing untrusted JSON does not in itself create an arbitrary code execution vulnerability, `pickle` does which makes it unsecure.

Resources

https://docs.python.org/3/library/pickle.html

171. Template strings

`Template` strings are a nice way to build up your strings beforehand:

```
>>> from string import Template
>>> s = Template("Hey $name, congratulations on your $belt Ninja Belt")
>>> s.substitute(name="Tim", belt="Orange")
'Hey Tim, congratulations on your Orange Ninja Belt'
>>> s = Template("$name, your balance is $$ $amount")  # escape $

>>> s.substitute(name="Julian", amount=10000)
'Julian, your balance is $ 10000'

# by default need to substitute all
>>> s.substitute(name="Julian")
Traceback (most recent call last):
...
KeyError: 'amount'
# if you want to allow for partial replacements
>>> s = s.safe_substitute(name="Julian")
>>> s
'Julian, your balance is $ $amount'

# works with files too (activate.sh has: source $venv_path/bin/activate)
>>> with open('activate.sh') as f:
...     Template(f.readline().strip()).substitute(venv_path='venv')
...
'source venv/bin/activate'
```

Explanation

Once you actually want to materialize the template, you can use `substitute` method to interpolate the variables. By default `substitute` requires you to replace them all, to allow for missing variables, use `safe_substitute`. To escape literal `$` signs, double them.

You might wonder what the advantage is over `format` which supports variable interpolation? It turns out `Template` strings are safer if you're handling format strings generated from user input (see the second link below).

Resources

https://docs.python.org/3/library/string.html#template-strings
https://realpython.com/python-string-formatting/#4-template-strings-standard-library

172. Upload a file to an S3 bucket (PyPI)

You can use the `boto3` module to manage AWS S3 buckets:

```python
import os
import boto3

FILE_URL = 'https://{bucket}.s3.us-east-2.amazonaws.com/{filename}'
S3_BUCKET = 'pybites-tips'
AWS_ACCESS_KEY_ID = os.environ.get('AWS_ACCESS_KEY_ID')
AWS_SECRET_ACCESS_KEY = os.environ.get('AWS_SECRET_ACCESS_KEY')

def upload_to_s3(filepath, bucket=S3_BUCKET):
    session = boto3.Session(
        aws_access_key_id=AWS_ACCESS_KEY_ID,
        aws_secret_access_key=AWS_SECRET_ACCESS_KEY
    )
    s3 = session.resource('s3')
    ret = s3.Bucket(bucket).put_object(
        Key=os.path.basename(filepath),
        Body=open(filepath, 'rb'),
        ACL='public-read')
    return FILE_URL.format(bucket=bucket, filename=ret.key)

if __name__ == '__main__':
    upload_to_s3('my-tip-image.png')
```

Explanation

Here we use the `boto3` module to automatically upload an image to an S3 bucket. This actually turns out to be a huge time saver managing our tip images. Talking about scratching our own itch.

Resources

https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Client.put_object

173. Parse git logs in Python

Use `subprocess` to run a shell command in Python:

```
>>> import subprocess


>>> cmd = 'git log -5 --oneline'
>>> output = subprocess.check_output(cmd, shell=True).splitlines()


>>> output_lines = [line.decode() for line in output]
>>> print(*output_lines, sep='\n')
9e61a85 Fix nox temp error on windows (#69)
db4f571 refactor tests - separate files and conftest.py for fixtures (#61)
e705392 fix #51 - avoids compiling by using psycopg2-binary (#67)
c93168f Feature notes - users can store public and private notes
bc654f7 Merge pull request #66 from pmayd/poetry_and_conda
```

Explanation

This is pretty useful if you want to parse outputs from existing shell commands or scripts. Why write bash code when you can use Python no? ;)

Be wary of the security risk using `shell=True` like we did here. In that case you are responsible for ensuring that all whitespace and metacharacters are quoted appropriately to avoid shell injection vulnerabilities (`shlex.quote()` can help you with this!)

Note that the returned lines are `bytes` so we use `decode` to cast them to strings (see Tip #126). And as we've seen in Tip #34, we can use `print` to print a list with a different separator.

Resources

https://docs.python.org/3/library/subprocess.html#subprocess.check_output
https://docs.python.org/3/library/subprocess.html#security-considerations

174. Defining custom exceptions

Capture and log standard error output using a custom exception:

```python
import logging
import subprocess


# define my own exception
class MyException(Exception):
    def __init__(self, cmd, stdout, stderr):
        super().__init__(f'{cmd} error (see stderr output for detail)')
        self.stdout = stdout
        self.stderr = stderr


# raise the exception
def run(...):
    ...
    out, err = subprocess.Popen(...)
    if err:
        raise MyException('my_command', out, err)


# catch the exception
try:
    run(...)
except MyException as exc:
    logging.error(exc.stderr)
```

Explanation

Here we first define a custom exception `MyException` by subclassing `Exception`. We have it receive standard output and error outputs.

Next when `subprocess.Popen` fails we raise our new exception.

Elsewhere in the code we catch this exception and log the standard error which is an attribute we now can access on the exception object.

Resources

https://docs.python.org/3/tutorial/errors.html#user-defined-exceptions
https://stackoverflow.com/a/1319675

175. A/B/C test your email campaigns

Here is a real world use case of a generator we used to send different emails:

```
>>> def gen_subjects():
...     while True:
...         yield "Keep your Python muscles strong"
...         yield ("You have not redeemed any Bite Exercises in a while, "
...                "get back on track!")
...         yield "You still have Bite Exercises to claim!"
...
>>> abc_subjects = gen_subjects()
>>> for _ in range(8):
...     next(abc_subjects)
...
'Keep your Python muscles strong'
'You have not redeemed any Bite Exercises in a while, get back on track!'
'You still have Bite Exercises to claim!'
'Keep your Python muscles strong'
'You have not redeemed any Bite Exercises in a while, get back on track!'
'You still have Bite Exercises to claim!'
'Keep your Python muscles strong'
'You have not redeemed any Bite Exercises in a while, get back on track!'
```

Explanation

Here we define a infinite generator that loops through 3 email subject variations. You can call `next` on the generator to retrieve the next subject.

For more generators see Tips #108 and #148. Really elegant, one of the many things that make us love Python!

Resources

https://pybit.es/generators.html

176. Zip and ship!

You can use `zipapp` to make an executable zipfile:

```
# get the code (= adapted script from Tip #83)
$ curl
https://gist.githubusercontent.com/pybites/c4b688fdf69a9f517086ac8cb2ba6b61/ra
w/0fb0c647b3329d8c2d04fbbfa4aa9fd56be5e839/amzlink.py --output amzlink.py


$ mkdir amz
# install required dependency
$ python -m pip install pyperclip --target amz
...


# create an entry point
$ cp amzlink.py amz/__main__.py


# make the zip executable
$ python -m zipapp -p "/usr/bin/env python3" amz -o amzlink


# copy this link to your clipboard:
# https://www.amazon.com/Essentialism-Disciplined-Pursuit-Greg-
McKeown/dp/0804137382
# get your affiliation link
$ ./amzlink
# generated link on your clipboard:
http://www.amazon.com/dp/0804137382/?tag=pyb0f-20
```

Explanation

Here we took the Amazon affiliation link creator (Tip #83) and added an entry point (`if __name__ == "__main__"`, see Tip #46).

As `pyperclip` is not part of the Standard Library we need to `pip install` it to a subdirectory (by the way, if you hit a `pyperclip.PyperclipException`, check out pyperclip.readthedocs.io for possible missing OS dependencies).

We use `zipapp`'s `-p` to point to a Python interpreter, and `-o` to call the script `amzlink`. Copying an Amazon book URL to our clipboard, running `./amzlink` it generates our affiliation link and puts it back on our clipboard, pretty neat!

Resources

https://docs.python.org/3/library/zipapp.html
https://pybit.es/zip-and-ship.html

177. Impose rules on your subclasses

You can use the `abstractmethod` decorator to force derived classes to implement certain methods:

```
>>> from abc import ABCMeta, abstractmethod


>>> class Developer(metaclass=ABCMeta):
...     @abstractmethod
...     def get_post_days(self):
...         """Subclasses of Developer must implement this method"""
...
>>> class Julian(Developer):
...     pass
...
# oops
>>> jul = Julian()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Julian with abstract method
get_post_days


>>> class Julian(Developer):
...     def get_post_days(self):
...         return 'Tue Wed'.split()
...
# ok
>>> jul = Julian()
>>>
```

Explanation

If you go OOP in Python abstract base classes (ABCs) cannot be missed. Here we use the `abstractmethod` decorator in the parent class to enforce children to implement `get_post_days`.

As you can see in the code, if we don't implement this method, we get a `TypeError`. This is a powerful way to enforce a particular interface.

Resources

https://docs.python.org/3/library/abc.html#abc.abstractmethod
https://pybit.es/oop-primer.html

178. Watch out for mutable default arguments

This is a common anti-pattern in Python to look out for:

```
>>> def sum_numbers(number, all_numbers=[]):
...     all_numbers.append(number)
...     return sum(all_numbers)
...
>>> sum_numbers(1, [2, 3])
6
>>> sum_numbers(1)
1
>>> sum_numbers(2)  # oops, should return 2!
3
>>> def sum_numbers(number, all_numbers=None):
...     if all_numbers is None:
...         all_numbers = []
...     all_numbers.append(number)
...     return sum(all_numbers)
...
>>> sum_numbers(1)
1
# ok now
>>> sum_numbers(2)
2
>>> sum_numbers(3)
3
```

Explanation

Default argument values are evaluated once upon module load. This means that if `all_numbers` is not provided; it keeps appending to the same default list that was initiated when the program started! Another `datetime` example is linked in the gist below.

The fix / best practice is to use `None` (a "sentinel value") as a default argument and handle this `None` case explicitly in the function's body.

Resources

https://docs.quantifiedcode.com/python-anti-patterns/correctness/mutable_default_value_as_argument.html
https://gist.github.com/pybites/749358550803b24018b15cf7dab04b21

179. Breaking code over multiple lines

Long lines of code are bad for opening files in split mode and for your code reviewers. Here are some techniques to break your code over multiple lines:

```
# break function args and dict (key, value) pairs
return render(request,
              'pomodoro/pomodoro.html',
              {'key': value, ...})

# break up ORM queries
completed_books_this_year = UserBook.objects.filter(
    user=user, status=COMPLETED, completed__year=goal.year
)

# multiline string
assert (f'<input class="js-favorite" title="favorite"'
        f' type="checkbox" bookid={snippet}') in html

# set compehensions
ids = {re.sub(r'.*id=', '', link.attrs['href'])
       for link in links
       if "/store/books/details/" in link.attrs['href']}

# function / generator expression
return sum(
    int(book.book.pages) if str(book.book.pages).isdigit() else 0
    for book in books if book.done_reading)

# break up conditionals
return (path in no_sidebar_pages
        or path.startswith('/tips/'))
```

Explanation

We already learned how to make multiline strings (Tip #5), but you should not stop there. Long lines can (and should) be broken over multiple lines by wrapping expressions in parentheses. You can manually do this checking `flake8` or you can automate this by using `black` (The Uncompromising Code Formatter).

Resources

https://pep8.org/#code-lay-out

180. Find and index

2 lesser known (used?) string methods to find the index of a substring:

```
>>> text = "I want to match premium and premium+ in this sentence"

>>> text.find('premium')
16
>>> text.rfind('premium')
28

>>> text.index('premium')
16
>>> text.rindex('premium')
28

# difference is how they deal with not founds
>>> text.find('nomatch')
-1
>>> text.index('nomatch')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Explanation

You can use `find` or `index` on a string to find the index of the substring you pass in. It returns the first (lowest) index `int` where the matching substring starts.

Their `rfind` and `rindex` counterparts return the last (highest) index, in other words, it starts looking from the right side of the string.

They work the same except for how they handle non matches: `find` returns `-1`, `index` raises a `ValueError` in that case.

Resources

https://docs.python.org/3/library/stdtypes.html#str.find
https://docs.python.org/3/library/stdtypes.html#str.index

181. Dictionary keys need to be immutable

You can only use immutable (hashable) types for `dict` keys:

```
>>> a = (1, 2, 3)  # tuple with immutable types
>>> b = (1, [], 2)  # spoiler: this tuple contains a mutable object
>>> d = dict()
>>> d[a] = 1
>>> d
{(1, 2, 3): 1}
>>> d[b] = 2
...
TypeError: unhashable type: 'list'

# another example
>>> a = set([1,2,3])
>>> d = {}
>>> d[a] = 1
...
TypeError: unhashable type: 'set'
>>> aa = frozenset(a)
>>> aa
frozenset({1, 2, 3})
>>> d[aa] = 1
>>>
```

Explanation

`list`s, `set`s, even `tuple`s with one or more mutable types inside of them cannot be used as dictionary keys which need to be hashable. For an immutable `set` you can use a `frozenset`.

Resources

http://radar.oreilly.com/2014/10/python-tuples-immutable-but-potentially-changing.html
https://docs.python.org/3/library/stdtypes.html#frozenset

182. How to tame greedy regexes

Be careful when you want to match a nested pattern, regexes are greedy:

```
>>> import re
>>> html = ("<div><p>Today a quick article on a nice caching module"
...         " when working with APIs.</p><p>Read more ...</p></div>")
>>> m = re.search('<p>.*</p>', html)
>>> m
<re.Match object; span=(5, 102), match='<p>Today a quick article on a nice
caching module>


# oops, matched both paragraphs
>>> m.group()
'<p>Today a quick article on a nice caching module when working with
APIs.</p><p>Read more ...</p>'


# adding a ? after the pattern to only match up until the first closing
paragraph tag
>>> m = re.search('<p>.*?</p>', html)
>>> m
# span matches 20 chars less
<re.Match object; span=(5, 82), match='<p>Today a quick article on a nice
caching module>


# again, group is great to refer to the matching substring
>>> m.group()
'<p>Today a quick article on a nice caching module when working with
APIs.</p>'
```

Explanation

We have seen this in Tip #54 but it bears repeating: regular expressions are greedy by default, that is they match as much text as possible!

To prevent this behavior, add the `?` after the qualifier (the regex meta character), which will do the match in non-greedy or minimal fashion; it will stop as soon as the matching condition has been satisfied.

By the way, here we use HTML tags because it makes for a good example, normally you'd use a library to parse HTML (e.g. `BeautifulSoup`, see Tip # 144),

Resources

https://pybit.es/mastering-regex.html
https://docs.python.org/3/library/re.html

183. How to use caching (memoization)

Caching matters! You can use "memoization" with the `lru_cache` decorator:

```
>>> from functools import lru_cache
>>> from timeit import timeit

>>> def nocache_fib(n):
...     if n < 2:
...         return n
...     return nocache_fib(n - 1) + nocache_fib(n - 2)
...

>>> @lru_cache
... def cached_fib(n):
...     if n < 2:
...         return n
...     return cached_fib(n - 1) + cached_fib(n - 2)
...

>>> timeit("nocache_fib(40)", "from __main__ import nocache_fib", number=1)
38.08430259900001
>>> timeit("cached_fib(40)", "from __main__ import cached_fib", number=1)
6.163200009723369e-05
```

Explanation

Caching is important and the time saving here is significant. `lru_cache` (LRU stands for "Least Recently Used") makes this pretty easy to implement too. You can give it an optional `maxsize` argument of recent calls it will save (by default this is 128). For timing we use `timeit` which we introduced in Tip #124.

Resources

https://docs.python.org/3/library/functools.html#functools.lru_cache
https://realpython.com/lru-cache-python/

184. Comparing two lists

Use `difflib.Differ` to compare two lists producing a nice diff output:

```
>>> import difflib
>>> julian_todos = ["1. Be awesome.", "2. Pybites.", "3. Enjoy a beer."]
>>> bob_todos = ["1. Be awesome!", "2. PyBites.", "3. Enjoy a beer."]
>>> for jtodo, btodo in zip(julian_todos, bob_todos):
...     if jtodo != btodo:
...         print(f"< {jtodo}\n> {btodo}")
...
< 1. Be awesome.
> 1. Be awesome!
< 2. Pybites.
> 2. PyBites.
# There must be a better way for this?! Yes! Enter difflib:
>>> for line in difflib.Differ().compare(julian_todos, bob_todos):
...     print(line)
...
- 1. Be awesome.
?              ^

+ 1. Be awesome!
?              ^

- 2. Pybites.
?      ^

+ 2. PyBites.
?      ^

  3. Enjoy a beer.
```

Explanation

The `difflib.Differ` class lets you compare sequences of lines of text, and produces human-readable diffs. No fun coding this up yourself, luckily the Standard Library has you covered!

Resources

https://pybit.es/comparing_lists.html
https://docs.python.org/3/library/difflib.html

185. Args and kwargs

There are 4 types of function arguments in Python:

```
>>> def send_email(from_, to='me', *cc, **messages):
...     print(f"{from_=} / {to=} / {cc=} / {messages=}")
...
# need at least from_ (using an underscore to not clash with a keyword)
>>> send_email()
...
TypeError: send_email() missing 1 required positional argument: 'from_'
# called with 1 arg, to defaults to 'me'
>>> send_email('info@pybit.es')
from_='info@pybit.es' / to='me' / cc=() / messages={}
# using *args
>>> send_email('info@pybit.es', '1@example.com', '2@example.com',
'3@example.com')
from_='info@pybit.es' / to='1@example.com' / cc=('2@example.com',
'3@example.com') / messages={}
# using *args and **kwargs
>>> send_email('info@pybit.es', '1@example.com', '2@example.com',
intro='hello', body='content', outro='kind regards')
from_='info@pybit.es' / to='1@example.com' / cc=('2@example.com',) /
messages={'intro': 'hello', 'body': 'content', 'outro': 'kind regards'}
# using * and ** unpacking
>>> cc = ('1@example.com', '2@example.com', '3@example.com')
>>> messages = {'intro': 'hello', 'body': 'content', 'outro': 'kind regards'}
>>> send_email('info@pybit.es', 'another@email.com', *cc, **messages)
from_='info@pybit.es' / to='another@email.com' / cc=('1@example.com',
'2@example.com', '3@example.com') / messages={'intro': 'hello', 'body':
'content', 'outro': 'kind regards'}
```

Explanation

1. Positional arguments are mandatory and have no default values.
2. Keyword arguments are not mandatory and have default values.
3. The arbitrary argument list (often called `args`) lets you pass in an extensible number of positional arguments.
4. The arbitrary keyword argument dictionary (often called `kwargs`) lets you pass in an undetermined series of named arguments.

Use 3. and 4. with caution though because it can obfuscate the signature of your function!

Resources

https://docs.python-guide.org/writing/style/#function-arguments

186. How to log an exception

`logging.exception` adds the exception info to the logging message:

```
>>> import logging
>>> FORMAT = '%(asctime)s %(levelname)s %(module)s %(funcName)s %(message)s'
>>> logging.basicConfig(filename='example.log', encoding='utf-8',
level=logging.INFO, format=FORMAT)
>>> logging.debug('not logged')
>>> logging.info('info is logged')
>>> def func(num1, num2):
...     try:
...         return num1/num2
...     except ZeroDivisionError:
...         logging.exception(f"{num1=}/{num2=} -> cannot divide by 0")
...
>>> func(1, 2)
0.5
>>> func(2, 0)
>>> ^D
$ cat example.log
2020-12-14 23:02:01,030 INFO <stdin> <module> info is logged
2020-12-14 23:03:41,510 ERROR <stdin> func num1=2/num2=0 -> cannot divide by 0
Traceback (most recent call last):
  File "<stdin>", line 3, in func
ZeroDivisionError: division by zero
```

Explanation

The `logging` module is quite easy to setup and use. Here we log various messages to an `example.log`.

Notice how `logging.exception` logs the `Traceback` alongside the message we gave it (note it should only be called from an exception handler):

Check out the "Logging HOWTO" linked below how to work with different loggers, handlers, filters, and formatters.

Resources

https://docs.python.org/3/library/logging.html
https://docs.python.org/3/howto/logging.html

187. Progress spinner

Make a terminal progress spinner with `itertools` and flushing standard output:

```python
import itertools
import sys
import time


def spinner(seconds):
    """Show an animated spinner while we sleep."""
    symbols = itertools.cycle('-\|/')
    tend = time.time() + seconds
    while time.time() < tend:
        # '\r' is carriage return: return cursor to the start of the line.
        sys.stdout.write('\rPlease wait... ' + next(symbols))  # no newline
        sys.stdout.flush()
        time.sleep(0.1)
    print()  # newline


if __name__ == "__main__":
    spinner(3)
```

Explanation

We found this recipe in the Kung Fu Itertools talk (EuroPython 2016). It's another cool use case of `itertools.cycle`, iterating ad infinitum over the various symbols that make up the spinner.

The important thing here is how it renders the characters "in place":
1. the `\r` in `write` returns the cursor to the start of the line, and
2. `sys.stdout.flush` will write everything in the buffer to the terminal.

Copy this code in your REPL and see this nice animation, all Standard Library. Credit and thanks to Víctor Terrón.

Resources

https://github.com/vterron/EuroPython-2016/blob/master/kung-fu-itertools.ipynb
https://pybit.es/itertools-examples.html

188. Countdown timer with music (PyPI)

Here is how to set a timer and play a song upon completion:

```python
import sys
import time

from playsound import playsound  # on Mac this requires PyObjC


def countdown(seconds: int) -> None:
    while seconds:
        mins, secs = divmod(seconds, 60)
        print(f'{mins:02}:{secs:02}', end="\r")
        time.sleep(1)
        seconds -= 1
    print("00:00", end="\r")
    playsound('alarm.mp4')


if __name__ == '__main__':
    if len(sys.argv) < 2 or not sys.argv[1].isdigit():
        script = sys.argv[0]
        print(f"Usage: {script} <minutes_till_alarm:int>")
        sys.exit()

    seconds = int(sys.argv[1]) * 60
    countdown(seconds)
```

Explanation

We use the `countdown` function to count down from a number of seconds to 0. `divmod` is a useful builtin that takes two (non complex) numbers as arguments and returns a pair of numbers consisting of their quotient and remainder when using integer division.

The `playsound` module is external so you'd have to `pip install` it. You can just feed it an mp3 / mp4 and it will play it, that simple! Further notice `sys.argv` for very simple command line parsing (for anything beyond this, use `argparse` or `click`). And lastly, `isdigit` is useful to see if a string can be converted to a digit.

Resources

https://www.geeksforgeeks.org/how-to-create-a-countdown-timer-using-python/
https://pypi.org/project/playsound/

189. Parse blog feeds (PyPI)

Need to parse RSS feeds, use `feedparser`:

```
>>> import feedparser
>>> from operator import itemgetter

>>> blog_feed = feedparser.parse('https://pybit.es/feeds/all.rss.xml')
>>> entry = blog_feed.entries[100]
>>> entry.title
'PyBites Twitter Digest - Issue 27, 2018'
>>> entry.author
'PyBites'
>>> entry.link
'https://pybit.es/twitter_digest_201827.html'
>>> for entry in blog_feed.entries[:5]:
...     itemgetter('link', 'author', 'published')(entry)
...
('https://pybit.es/get-python-source.html', 'Bob', 'Mon, 14 Dec 2020 19:05:00
+0100')
('https://pybit.es/guest-create-aws-lambda-layers.html', 'Michael Aydinbas',
'Mon, 05 Oct 2020 14:22:00 +0200')
('https://pybit.es/guest-clean-text-data.html', 'David Colton', 'Wed, 30 Sep
2020 20:34:00 +0200')
('https://pybit.es/code-reviewing.html', 'Bob', 'Thu, 24 Sep 2020 18:43:00
+0200')
('https://pybit.es/opensource-package-pypi.html', 'Bob', 'Mon, 31 Aug 2020
12:05:00 +0200')
```

Explanation

We love `feedparser`, it makes parsing an RSS feed a breeze. Here we parse our feed, taking a blog entry and retrieving its data by attribute which is a really nice feature (check out the dunder methods used in `feedparser.FeedParserDict` ...)

Then we reuse `itemgetter` (Tip #65) to retrieve multiple attributes at once. Few lines of code, a lot accomplished, thanks to this amazing library.

Resources

https://pypi.org/project/feedparser/
https://pybit.es/guest-pybites-blog-tag-analysis-plotly.html

Exercise

Bite 220. Analysing @pythonbytes RSS feed

190. __slots__ can save you memory

If you are going to create a lot of instances of a class, you can reduce memory usage with `__slots__`:

```
>>> class WithoutSlots:
...     def __init__(self, x, y, z):
...         self.x = x
...         self.y = y
...         self.z = z
...
>>> class WithSlots:
...     __slots__ = 'x', 'y', 'z'
...     def __init__(self, x, y, z):
...         self.x = x
...         self.y = y
...         self.z = z
...
>>> from pympler.asizeof import asizeof  # pip install
>>> w1 = WithoutSlots(1, 2, 3)
>>> w2 = WithSlots(4, 5, 6)
>>> asizeof(w1)
416
>>> asizeof(w2)
152
>>> w1.a = 1
>>> w2.a = 1
...
AttributeError: 'WithSlots' object has no attribute 'a'
```

Explanation

As per the exhaustive Stack Overflow answer linked below, the goal of using `__slots__` is to save space in objects. Instead of having a dynamic dict that allows adding attributes to objects at any time, there is a static structure which does not allow additions after creation. This saves the overhead of one dict for every object that uses slots

That's why in the example `w1.a = 1` is allowed but `w2.a = 1` is not. For that limitation you get quite a significant reduction in memory use in return though.

Resources

https://blog.usejournal.com/a-quick-dive-into-pythons-slots-72cdc2d334e
https://stackoverflow.com/a/28059785

191. Assert statements

Be careful with asserts (unless you're using `pytest`), they can be turned off!

```
$ more script.py
from datetime import date

pybites_founded = date(2016, 12, 19)

days_alive = (date.today() - pybites_founded).days
years, remainder = divmod(days_alive, 365)

assert remainder == 0, 'Not a PyBites birthday'

print(f"PyBites is {years} years old")

# running this the 17th of Dec, 2 days before our birthday
$ python script.py
Traceback (most recent call last):
...
AssertionError: Not a PyBites birthday

# -O ignored asserts
$ python -O script.py
PyBites is 3 years old
```

Explanation

Here we assert that `script.py` gets run exactly on PyBites' birthday, the 19th of December. If that's not the case the `assert` (debugging) statement is not `True` which will cause an `AssertionError` to be raised. You can specify a custom message after the comma.

Be wary though that if we run Python with `-O` it removes `assert` (and the equivalent `__debug__`) statements from your code (you can also set the `PYTHONOPTIMIZE` environment variable to a non-empty string). Calling Python with `-OO` will also discard docstrings.

Resources

https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement
https://docs.python.org/3/using/cmdline.html#cmdoption-o

192. Heap queues in Python

The `heapq` module provides Python's implementation of the heap / priority queue algorithm. It's useful to get the top / bottom N items:

```
>>> import heapq
>>> import random
>>> from operator import itemgetter

>>> numbers = random.sample(range(50), 10)
>>> numbers
[39, 10, 17, 37, 4, 13, 42, 6, 30, 27]
>>> heapq.nlargest(3, numbers)
[42, 39, 37]
>>> heapq.nsmallest(3, numbers)
[4, 6, 10]
>>> heapq.heapify(numbers)
>>> numbers
[4, 6, 13, 30, 10, 17, 42, 37, 39, 27]

>>> n1 = [10, 17, 37, 39]
>>> n2 = [4, 6, 13, 27, 30, 42]
>>> list(heapq.merge(n1, n2))
[4, 6, 10, 13, 17, 27, 30, 37, 39, 42]

>>> user_scores = {'bob': 3, 'julian': 7, 'tim': 10, 'sara': 2}
>>> heapq.nsmallest(2, user_scores)  # by lowest name first char
['bob', 'julian']
>>> heapq.nsmallest(2, user_scores.items(),key=itemgetter(1))  # lowest score
[('sara', 2), ('bob', 3)]
# >>> sorted(user_scores.items(), key=itemgetter(1))[:2]
```

Explanation

In this example we get some random numbers first. Right off the bat we get the top / bottom N numbers using `nlargest` and `nsmallest`. Note that, like `sorted`, you can use the `key` keyword argument to sort by other attributes. You can also sort the list with `heapify`, and even `merge` multiple sorted inputs into a single sorted output.

Resources

https://docs.python.org/3/library/heapq.html

193. Most popular email domains (PyPI)

List five in your head, then look at the code ...

```
>>> import bs4
>>> import requests

>>> url = ("https://email-verify.my-addr.com/list-of-most-"
...        "popular-email-domains.php")
>>> resp = requests.get(url)
>>> soup = bs4.BeautifulSoup(resp.content, 'html.parser')

>>> div = soup.find("div", attrs={"class": "middle_info_noborder"})
>>> trs = div.find_all('tr')

>>> for tr in trs[:5]:
...     tr.find_all('td')[2:]
...
[<td>gmail.com</td>, <td>17.74% </td>]
[<td>yahoo.com</td>, <td>17.34% </td>]
[<td>hotmail.com</td>, <td>15.53% </td>]
[<td>aol.com</td>, <td>3.2% </td>]
[<td>hotmail.co.uk</td>, <td>1.27% </td>]

>>> tr1 = trs[0]
>>> tr1.find_all('td')[2:]
[<td>gmail.com</td>, <td>17.74% </td>]
>>> [t.text.strip() for t in tr1.find_all('td')[2:]]
['gmail.com', '17.74%']
```

Explanation

Combining a lot of elements from previous tips here:
- First we use `requests` (Tip #39) to get the html of mentioned `url`.
- Then we use `BeautifulSoup` (Tip #144) to navigate the DOM elements more easily locating a `div` with class name `middle_info_noborder`, then get all `tr` (table row) elements.
- Lastly we loop through the first 5 extracting the 3rd and 4th column of the table = the domain and % used.
- As the returned elements are `bs4.element.Tags` you can access the `text` attribute on them which we do in a list comprehension (Tip #18) in the last step.

Resources

https://beautiful-soup-4.readthedocs.io/en/latest/

194. Debugging a hanging test (PyPI)

The `pytest-timeout` plugin is useful to break a hanging test:

```
# script.py
from time import sleep

def call_api():
    sleep(60)  # faking a timeout
    return dict(
        status=200,
        response=[1, 2, 3])

# test_script.py
from script import call_api

def test_call_api():
    resp = call_api()
    assert resp['status'] == 200
    assert resp['response'] == [1, 2, 3]

$ pytest test_script.py --timeout=3
# output truncated
plugins: timeout-1.4.2
...
== FAILURES ==
__ test_call_api __
...
    def call_api():
>       sleep(60)  # faking a timeout
E       Failed: Timeout >3.0s
...
== 1 failed in 3.05s ==
```

Explanation

`pytest-timeout` lets you call `pytest` with `--timeout=SECONDS`. When aborting a test it will show a stack dump of all threads running at the time, very useful when you have a hanging test.

Resources

https://pybit.es/pytest-timeout.html

195. Use sqlite3 in Python

Python comes with `sqlite3`, a DB-API 2.0 interface for SQLite databases:

```
>>> from contextlib import contextmanager
>>> import sqlite3

>>> @contextmanager
... def connect_db():
...     try:
...         conn = sqlite3.connect('movies.db')
...         conn.row_factory = sqlite3.Row  # fetchall returns dicts
...         yield conn.cursor()
...     finally:
...         conn.commit()
...         conn.close()
...
>>> movies = [('Inception', 2010, 'Christopher Nolan', 8.8),
...           ('The Shawshank Redemption', 1994, 'Frank Darabont', 9.2),
...           ('Heat', 1995, 'Michael Mann', 8.2)]

>>> with connect_db() as cursor:
...     cursor.execute("CREATE TABLE movies (title, year, director, rating)")
...     cursor.executemany('INSERT INTO movies VALUES (?,?,?,?)', movies)
...     movies = cursor.execute("SELECT * FROM movies WHERE year > 2000;")
...     first = movies.fetchall()[0]
...     dict(first)
...
# omitting "<sqlite3.Cursor object at ... " standard output
{'title': 'Inception', 'year': 2010, 'director': 'Christopher Nolan',
'rating': 8.8}
```

Explanation

Here we make a DB connector context manager which we then use in a `with` statement. We create a `movies` table and insert 3 movies. Then we retrieve the one that is newer than 2000. Thanks to `conn.row_factory = sqlite3.Row` the matching row comes back as a `dict` instead of a `tuple`. Note that to prevent SQL injection, you should always use `?` placeholders (not `%s`!)

Resources

https://docs.python.org/3/library/sqlite3.html

196. Left and right justify strings

With f-strings you can use `<`, `>` and `^` to left / right / center justify strings:

```
>>> import random
>>> names = 'Julian Bob Martin Rodolfo'.split()
>>> scores = random.sample(range(1, 11), len(names))
>>> scores
[8, 3, 9, 2]
>>> for name, score in zip(names, scores):
...     print(f"{name:<20} {score}")
...
Julian               8
Bob                  3
Martin               9
Rodolfo              2
>>> for name, score in zip(names, scores):
...     print(f"{name:^20} | {score:<5}")
...
       Julian         | 8
        Bob           | 3
       Martin         | 9
      Rodolfo         | 2
>>> for name, score in zip(names, scores):
...     print(f"{name:>20} | {score:<05}")
...
              Julian | 80000
                 Bob | 30000
              Martin | 90000
             Rodolfo | 20000
```

Explanation

Here we use `random.sample` (Tip #13) again to get some random scores for the 4 `names`. Then we loop over the `names` and `scores` with `zip` (see Tips #3, #73, #78, #104, #137 and #184) and look at various ways to adjust the `name` and `score` strings. We can even prepend the `scores` with 0s.

Resources

https://www.python.org/dev/peps/pep-0498/#format-specifiers
https://docs.python.org/3/library/string.html#format-specification-mini-language

197. Use re.VERBOSE to explain your regex

You can use `re.VERBOSE` to make your regular expression patterns more readable:

```
>>> import re
>>> bio = """
... name: PyBites
... origin: Worldwide
... born: 2016-12-19
... """
>>> pat = re.compile(r"""
...      \nname:\s          # a newline, then a literal string and a space
...      (?P<name>.*)\n     # capture name, then a newline
...      origin:\s          # literal string followed by a space
...      (?P<origin>.*)\n   # capture origin, then a newline
...      born:\s            # literal string followed by a space
...      (?P<born>.*)\n     # capture born date, then a newline
... """, re.VERBOSE)
>>>
>>> m = pat.match(bio)
>>> m
<re.Match object; span=(0, 50), match='\nname: PyBites\norigin:
Worldwide\nborn: 2016-12>
>>> m.groupdict()
{'name': 'PyBites', 'origin': 'Worldwide', 'born': '2016-12-19'}
>>> m.group('name'), m.group('origin'), m.group('born')
('PyBites', 'Worldwide', '2016-12-19')
```

Explanation

Here we break up a regular expression pattern over multiple lines thanks to `re.VERBOSE` which ignores whitespace and comments. It's verbose but it can be helpful to somebody that is still new to regexes.

This tip also illustrates how to capture strings from matches using `(?P.*)`. If the pattern matches, the regular expression engine puts the captured strings in the match object (`m` here) which you can retrieve with `groupdict()` or `group('variable')`.

Resources

https://pybit.es/mastering-regex.html

198. Split by a more complex pattern

You can use `re.split` to split by a regular expression pattern:

```
>>> import re
>>> lines = """
... Inception,2010;Christopher Nolan|8.8
... The Shawshank Redemption;1994|Frank Darabont?9.2
... Heat^1995[Michael Mann]8.2
... """

>>> for line in lines.strip().splitlines():
...     re.split(r'[^\w\s\.]', line)
...
['Inception', '2010', 'Christopher Nolan', '8.8']
['The Shawshank Redemption', '1994', 'Frank Darabont', '9.2']
['Heat', '1995', 'Michael Mann', '8.2']

# or to not repeat the regex pattern inside the loop:
>>> pat = re.compile(r'[^\w\s\.]')
>>> for line in lines.strip().splitlines():
...     re.split(pat, line)
...
['Inception', '2010', 'Christopher Nolan', '8.8']
['The Shawshank Redemption', '1994', 'Frank Darabont', '9.2']
['Heat', '1995', 'Michael Mann', '8.2']
```

Explanation

The advantage of `re.split` is that you can define a very robust pattern, here that is anything that is not (`^`) an alphanumeric character (`\w`) nor a space (`\s`) nor a literal dot (`.`). This example might be a bit contrived but it does show that we can turn messy data into something usable.

Lastly, if you are going to repeat a regex inside the loop it's probably better to define it once upfront (although Python's regular expression engine already has optimizations for this scenario).

Resources

https://docs.python.org/3/library/re.html#re.split
https://docs.python.org/3/howto/regex.html

199. Make an md5 hash of a string

Here is how we can make a `md5` hash from an email:

```
>>> import hashlib
>>> email = 'bob@pybit.es'
>>> hashlib.md5(email)
...
TypeError: Unicode-objects must be encoded before hashing
>>> hashlib.md5(email.encode('utf-8'))
<md5 _hashlib.HASH object @ 0x7fddf88e75f0>
>>> dir(hashlib.md5(email.encode('utf-8')))
[..., 'block_size', 'copy', 'digest', 'digest_size', 'hexdigest', 'name',
'update']
>>> hashlib.md5(email.encode('utf-8')).digest()
b"[\x135mFz\xf8\x861P<'\xa3\xd0\xe0\xcf"
>>> hashlib.md5(email.encode('utf-8')).hexdigest()
'5b13356d467af88631503c27a3d0e0cf'
```

Explanation

Gravatar (Globally Recognized Avatar) uses this technique for their avatar image links.

Note that we need to `encode()` the email string first. `digest()` returns a bytes object, and `hexdigest()` returns a hexadecimal string object, twice as long as the object returned by `digest()`.

Also note that we use `dir()` again (Tip #25) to see what methods we can call on the object.

Resources

https://docs.python.org/3/library/hashlib.html
https://en.gravatar.com/site/implement/images/python/

200. Decoding binary data

Here we use `b64decode` (`base64` module) to decode base64-encoded csv data:

```
>>> import json
>>> import base64
>>> import requests

>>> url = "https://bites-data.s3.us-east-2.amazonaws.com/sales.json"
>>> response = requests.get(url)
>>> data = json.loads(response.text)

>>> data['content']
'bW9udGgsc2FsZXMNCjIwMTMtMDEtMDEsMTQyMzY...'

>>> content = base64.b64decode(data["content"])  # returns bytes
>>> for line in content.decode('utf-8').splitlines():
...     print(line)
...
month,sales
2013-01-01,14236.9
2013-02-01,4519.89
2013-03-01,55691.01
2013-04-01,28295.35
2013-05-01,23648.29
...
```

Explanation

`base64.b64decode` decodes a `Base64` encoded bytes-like object / ASCII string and returns the decoded bytes.

Here we download some (fake) JSON sales data using `requests` (Tip #39) and load it in with `json.loads` (Tip #84). The content is encoded so we decode it using `base64.b64decode`. As it returns `bytes` we need to convert it to a string for which we use `decode('utf-8')` (Tip #126). Then we split by newline / `\n` (Tip #62) and print the data.

Resources

https://docs.python.org/3/library/base64.html#base64.b64decode

201. Dictionary unpacking

Here is how you can unpack a `dict` inside `format`:

```
# example from libgravatar
>>> base_url = "{protocol}://{domain}/avatar/{hash}{extension}{params}"

>>> params_dict = {
...     'protocol': 'https',
...     'domain': 'secure.gravatar.com',
...     'hash': '5b13356d467af88631503c27a3d0e0cf',
...     'extension': '.jpg',
...     'params': '?somevar=1'
... }

>>> base_url.format(**params_dict)
'https://secure.gravatar.com/avatar/5b13356d467af88631503c27a3d0e0cf.jpg?somev
ar=1'
```

Explanation

In Tips #6 and #34 we looked at tuple unpacking. Did you know you can also unpack a `dict`?

We touched upon it in Tip #185 when we talked about `kwargs`, but this is a bit more explicit example.

Another use case is the old way of merging `dict`s: `{**dict1, **dict2}` (as discussed in Tip #114 the Python >= 3.9 way is to use a union operator here: `dict1 | dict2`).

Resources

https://www.python.org/dev/peps/pep-0448/
https://github.com/pabluk/libgravatar/blob/71904d7e7bfe0e23c97e5c1cc7590a92583c5cb5/libgravatar/__init__.py#L143

202. Add more info to your tests (PyPI)

Use `pytest.param` to apply marks or set test IDs to individual parametrized test:

```
# test_objects.py
import pytest
from objects import score_objects  # gist linked below

@pytest.mark.parametrize("arg, expected", [
    pytest.param(['none', 1, 'nonsense'], 0, id="nothing_matches"),
    pytest.param(['random'], 3, id="one_module"),
    pytest.param(['raise', 'random'], 5, id="one_keyword_one_module"),
])
def test_score_objects(arg, expected):
    assert score_objects(arg) == expected

# 1. more test function info in verbose mode
$ pytest -v
...
test_objects.py::test_score_objects[nothing_matches] PASSED
# 2. upon failure = shows id
$ pytest
...
FAILED test_objects.py::test_score_objects[one_keyword_one_module] -
AssertionError: assert 5 == 4
# 3. helps filtering tests
$ pytest -k nothing
...
1 passed, 2 deselected
```

Explanation

We love `@pytest.mark.parametrize`, but you might lose some detail you had when using individual test functions. One way to solve this is to use `pytest.param` setting marks, or in this case, add a string to the `id` keyword which shows up alongside the test function name and allows you to filter on using `pytest`'s `-k` switch.

Resources

https://docs.pytest.org/en/stable/example/parametrize.html
https://gist.github.com/pybites/c964c34e46c84f5e9c9e335b098b3aaa

203. The handy modulo operator

The modulo operator can be handy to figure out which numbers are even or do an operation every N items:

```
>>> names = ['Phyllis', 'Amelina', 'Delaney', 'Kinsley', 'Carena',
...          'Imogene', 'Tyler', 'Olag', 'Gilberta', 'Rufe']
>>>
>>> def print_names_to_columns(names: list[str], cols: int = 2) -> None:
...     for i, name in enumerate(names, 1):  # you can start at 1!
...         print(f"| {name:<10}", end="")
...         if i % cols == 0 or i == len(names):
...             print()
...
>>> print_names_to_columns(names)
| Phyllis   | Amelina
| Delaney   | Kinsley
| Carena    | Imogene
| Tyler     | Olag
| Gilberta  | Rufe
>>> print_names_to_columns(names, 4)
| Phyllis   | Amelina   | Delaney   | Kinsley
| Carena    | Imogene   | Tyler     | Olag
| Gilberta  | Rufe
```

Explanation

Here we create a function called `print_names_to_columns` which takes a `list` of names and number of columns (`cols`, default = `2`) to print them to. `{name:<10}` makes the columns 10 characters wide and aligns the text to the left (see Tip #196).

We use the modulo operator to add a newline `\n` after each Nth column (= `i % cols == 0` check). We added the `or i == len(names)` to this condition to make sure we always finish the output with a newline.

Resources

https://docs.python.org/3/reference/expressions.html#binary-arithmetic-operations

204. Printing to standard error

You can print to standard error output by passing `file=sys.stderr` to the `print` function:

```
# script_1.py
print("printing...")

$ python script_1.py 1>out.txt 2>err.txt

$ wc -l out.txt err.txt
    1 out.txt
    0 err.txt
    1 total

# script_2.py
import sys
print("printing...", file=sys.stderr)

$ python script_2.py 1>out.txt 2>err.txt

$ wc -l out.txt err.txt
    0 out.txt
    1 err.txt
    1 total
```

Explanation

In your (Unix) shell you can access the "stdout" (standard output) and "stderr" (standard error) output streams by the "file descriptors" `1` and `2` respectively.

Here we use this technique to redirect each kind of output to a file. By default `print` outputs to standard output, but if we give it `file=sys.stderr`, it will print to standard error instead.

Resources

https://docs.python.org/3/library/functions.html#print
https://stackoverflow.com/a/818284

205. Add content to an existing file

Use `open` with `a` (append mode) if you want to append to an existing file (not truncating its existing content):

```
>>> with open('file', 'w') as f:
...     f.write("some content\n")
...
13
>>> with open('file', 'r') as f:
...     f.readlines()
...
['some content\n']
>>> with open('file', 'w') as f:
...     f.write("some other content\n")
...
19
>>> with open('file', 'r') as f:
...     f.readlines()
...
['some other content\n']  # oops
>>> with open('file', 'a') as f:
...     f.write("appending new content\n")
...
22
>>> with open('file', 'r') as f:
...     f.readlines()
...
['some other content\n', 'appending new content\n']
```

Explanation

If you open a file in `w` mode, you will not only open it for writing, you will also truncate it first!

Sometimes you want this, other times you rather want to append to existing content. For that goal use `open`'s `a` mode ("open for writing, appending to the end of the file if it exists") instead.

If you just want to be warned (by means of a `FileExistsError` exception) use `open` with `x` (see Tip #35).

Resources

https://docs.python.org/3/library/functions.html#open

206. Extract dictionary keys and values

Here is a concise way of unpacking keys and values of a dictionary into two lists:

```
>>> user_scores = {'bob': '11', 'julian': '22', 'tim': '33', 'sara': '44'}
>>> user_scores.keys()
dict_keys(['bob', 'julian', 'tim', 'sara'])
>>> user_scores.values()
dict_values(['11', '22', '33', '44'])

# in one go using tuple unpacking:
>>> x_axis, y_axis = zip(*user_scores.items())
>>> x_axis
('bob', 'julian', 'tim', 'sara')
>>> y_axis
('11', '22', '33', '44')

# example with a list of 3 item tuples:
>>> x, y, z = zip(*[(i, k, v) for i, (k, v) in
...                  enumerate(user_scores.items(), start=1)])
>>> x
(1, 2, 3, 4)
>>> y
('bob', 'julian', 'tim', 'sara')
>>> z
('11', '22', '33', '44')
```

Explanation

Remember Tip #3 where we made a `dict` from two `list`s? This tip is the inverse.

First we extract the keys and values with `dict`'s `keys()` and `values()` respectively. Then we do it in one go using `zip` (which works because `user_scores.items()` returns a `list` of `tuple`s).

Data vizualization libraries use this technique often to extract x and y axis values for plotting.

Resources

https://stackoverflow.com/a/6612821

207. How to test logging

You can test logging with `pytest`'s `caplog` fixture:

```
# script.py
import logging

def func():
    logging.debug("a debug message to ignore")
    logging.info("an info message")
    try:
        1 / 0
    except ZeroDivisionError:
        logging.exception("cannot divide by 0")

# test_script.py
import logging

from script import func

def test_func(caplog):
    caplog.set_level(logging.INFO)
    func()
    record1, record2 = caplog.records
    assert record1.levelname == "INFO"  # no debug
    assert record1.message == "an info message"
    assert record2.message == "cannot divide by 0"
    assert record2.exc_info[0] is ZeroDivisionError
```

Explanation

Here we made a function called `func` that logs 3 messages: `DEBUG`, `INFO` and `ERROR` (an exception uses that `levelname` as we learned in Tip #76).

In the test we use the `caplog` fixture to grab those logging messages, a really useful technique! Each record in `caplog.records` is a `LogRecord` object. Using `dir` on it (see Tip #25) you can see it has a lot of useful attributes (e.g. `created`, `exc_info`, `levelname`, `message`, `lineno`, etc.)

Resources

https://docs.pytest.org/en/latest/logging.html#caplog-fixture

208. Get annotations

You can get the type annotations of a function via its `__annotations__` attribute:

```
>>> def hello(name: str) -> str:
...     return f"Hello {name}"
...
>>> hello.__annotations__
{'name': <class 'str'>, 'return': <class 'str'>}

# no return
>>> def hello(name: str):
...     pass
...
>>> hello.__annotations__
{'name': <class 'str'>}

# another example
>>> from typing import List  # Python < 3.9
>>> def print_names_to_columns(names: List[str], cols: int = 2) -> None:
...     """A function to print names to N columns"""
...
>>> print_names_to_columns.__annotations__
{'names': typing.List[str], 'cols': <class 'int'>, 'return': None}
```

Explanation

Here we define a couple of functions with different arguments and return values ("function signatures"), including one without an explicit return.

We can reference the special `__annotations__` attribute on the function, which is a `dict` holding all type annotations of the function (as per Tip #38: functions are "first-class objects")

As we've seen in Tip #117, starting Python 3.9 you can use `list` directly for typing, so you can drop the `from typing import List` import.

Resources

https://www.python.org/dev/peps/pep-3107/#accessing-function-annotations

209. Check similarity between blog tags

You can use `SequenceMatcher` (`difflib` module) to analyze similarity between strings:

```
>>> from itertools import combinations
>>> from difflib import SequenceMatcher

>>> tags = 'python pythonista developer development'.split()

>>> for pair in combinations(tags, 2):
...     similarity = SequenceMatcher(None, *pair).ratio()
...     print(pair, similarity)
...
('python', 'pythonista') 0.75
('python', 'developer') 0.13333333333333333
('python', 'development') 0.23529411764705882
('pythonista', 'developer') 0.10526315789473684
('pythonista', 'development') 0.19047619047619047
('developer', 'development') 0.8
```

Explanation

Here we use `itertools.combinations` to get all possible pairs between the 4 tags (it works like this: `combinations('ABCD', 2) --> AB AC AD BC BD CD`).

We pass each pair into the `SequenceMatcher` class calling its `ratio` method which returns the similarity score (float) in the range (0, 1). Here we start to see the functionality of a little recommendation engine ...

Resources

https://docs.python.org/3/library/itertools.html#itertools.combinations
https://docs.python.org/3/library/difflib.html#difflib.SequenceMatcher

210. ASCII strings

Here is how to determine if a string is ASCII only:

```
>>> s, t = "python", "très bien"

# use ord
>>> def is_ascii(word):
...     return all(ord(char) < 128 for char in word)
...
>>> is_ascii(s), is_ascii(t)
(True, False)

# another way
>>> t.encode()
b'tr\xc3\xa8s bien'
>>> t.encode('ascii', errors='ignore')
b'trs bien'
>>> len(t.encode('ascii', errors='ignore')) < len(t)
True

# now you can use isascii though (>= Python 3.7)
>>> s.isascii()
True
>>> s.isascii(), t.isascii()
(True, False)
```

Explanation

Here we define two strings and see if they contain only ASCII characters. First we define a helper that uses `ord` which returns an integer representation of each (Unicode) character.

Secondly we use `encode` ignoring any (Unicode) characters that are not encodable using the `errors='ignore'` flag.

However starting Python 3.7 there is an even easier way: `str`s now have the `isascii` method for this purpose.

Resources

https://docs.python.org/3/library/stdtypes.html#str.isascii
https://stackoverflow.com/a/196392

211. Use selenium for integration testing (PyPI)

Here we use Selenium to test logging in to our app:

```
# test module
import pytest
from selenium import webdriver
from selenium.webdriver.common.keys import Keys


@pytest.fixture
def browser():
    driver = webdriver.Chrome()
    yield driver
    driver.quit()


def test_login(browser):
    browser.get("http://localhost:8000/login/")
    src = browser.page_source
    assert 'PyBites Login' in src
    browser.find_element_by_name('username').send_keys('bob')
    browser.find_element_by_name('password').send_keys(
        'handl3bar' + Keys.RETURN)
    src = browser.page_source
    assert 'Logout' in src
    assert 'Login' not in src
```

Explanation

First we define a `pytest.fixture` (Tip #130) that returns a driver object (browser) and automatically closes it.

The test function uses this fixture to go to the login page of the web app we have running. It asserts we landed on the logged out view first. Then we login and assert that we are in the logged in view.

This is just for example's sake. We probably would turn this login sequence into another fixture to reuse it across other test functions.

Resources

https://selenium-python.readthedocs.io/
https://pybit.es/pytest-fixtures.html

212. Convert a dict to json

You can use `json.dumps` to convert a `dict` into JSON:

```
>>> payload = {
...   "description": "the description for this gist",
...   "public": True,
...   "files": {
...     "file1.txt": {
...       "content": "String file contents"
...     }
...   }
... }
>>> type(payload)
<class 'dict'>
>>> import json
>>> payload_json = json.dumps(payload, indent=4)
>>> type(payload_json)
<class 'str'>
>>> print(payload_json)
{
    "description": "the description for this gist",
    "public": true,
    "files": {
        "file1.txt": {
            "content": "String file contents"
        }
    }
}
```

Explanation

This is handy when you need to pass in JSON when calling an API endpoint like we'll see in the next tip.

Note that when a dictionary is converted into JSON, all the keys of the dictionary are coerced to strings. So converting it back to a `dict` you might not end up with the same initial data.

Resources

https://docs.python.org/3/library/json.html#json.dumps

213. Create a gist using requests (PyPI)

Automatically add a gist to your Github account using `requests` and the Github API:

```
import json

from decouple import config  # pip install python-decouple
import requests

TOKEN = config('GITHUB_TOKEN')

def create_gist(filename, description, code, public=True):
    payload = json.dumps({
        "description": description,
        "public": public,
        "files": {
            filename: {"content": code}
        }
    })
    gists_url = "https://api.github.com/gists"
    headers = {'Authorization': f'token {TOKEN}'}
    resp = requests.post(gists_url, data=payload, headers=headers)
    return resp.json()['html_url']
```

Explanation

First create a personal access token to your Github account. Then you can use the function above like this: `create_gist('file2.txt', 'my automated gist', '#!env python\nprint("Hello world!")')`. It will return the URL of the gist that was created: `https://gist.github.com//`.

We saw in the previous tip how you can use `json.dumps` to turn a `dict` into JSON. We use `python-decouple` (Tip #151) again to load in the secret Github token from an `.env` file (or overriding environment variable).

We use `requests.post` against the Github gists API endpoint, including the payload JSON and a header that includes the API access token.

Resources

https://docs.github.com/en/free-pro-team@latest/rest/reference/gists#create-a-gist
https://requests.readthedocs.io/en/master/user/quickstart/#more-complicated-post-requests

214. Walk a directory tree

You can use `os.walk` to get all the files in a directory recursively:

```python
from collections import Counter
import os
from pathlib import Path

def count_extensions(origin='.', top=10):
    cnt = Counter()
    for root, dirs, files in os.walk(origin):
        for file in files:
            cnt[Path(file).suffix] += 1
    return cnt.most_common(top)

ret = count_extensions()
print(ret)

# [('.py', 1659), ('.pyc', 711), ('.html', 337), ('', 109), ('.txt', 35),
# ('.json', 16), ('.csv', 14), ('.sample', 11), ('.exe', 10), ('.png', 9)]
```

Explanation

`os.walk` lets us walk a directory tree returning `tuple`s of `root, dirs, files`.

Here we only want the filenames to see which file extensions are most common, for which we use `Counter` (Tip #14) again.

The more 'modern way' of getting a file extension is to use `pathlib.Path`'s `suffix` method.

Resources

https://docs.python.org/3/library/os.html#os.walk
https://docs.python.org/3/library/pathlib.html#pathlib.PurePath.suffix

215. Randomly shuffle a sequence

Here we use `random.shuffle` to reorder a deck of cards in place:

```
>>> from collections import namedtuple
>>> from pprint import pprint as pp
>>> import random

>>> Card = namedtuple('Card', ['value', 'suit'])
>>> suits = ['hearts', 'diamonds', 'spades', 'clubs']
>>> cards = [Card(value, suit) for value in range(1, 14) for suit in suits]
>>> pp(cards[:3])
[Card(value=1, suit='hearts'),
 Card(value=1, suit='diamonds'),
 Card(value=1, suit='spades')]
>>> random.shuffle(cards)

# shuffled in place
>>> pp(cards[:3])
[Card(value=13, suit='clubs'),
 Card(value=1, suit='spades'),
 Card(value=2, suit='spades')]
```

Explanation

First we create a deck of cards using `namedtuple` (Tip #143). The list comprehension (Tip #18) creates them in order.

Then we pass the cards `list` to `random.shuffle` which reorders them in place. We use `pprint` (Tip #145) again for pretty printing a slice of the cards `list`.

Resources

https://docs.python.org/3/library/random.html#random.shuffle
https://stackoverflow.com/a/41971084

216. Dataclasses and order

`dataclasses` are not orderable by default, unless you pass in `order=True` when constructing them:

```
>>> from dataclasses import dataclass
>>> from enum import IntEnum

>>> class BiteLevel(IntEnum):  # supports comparing integers
...     BEGINNER = 2
...     INTERMEDIATE = 3
...     ADVANCED = 4
...
>>> @dataclass
... class Bite:
...     number: int
...     title: str
...     level: int = BiteLevel.BEGINNER
...
>>> bites = [
...     Bite(11, "Enrich a class", BiteLevel.ADVANCED),
...     Bite(21, "Query a nested DS", BiteLevel.BEGINNER),
...     Bite(25, "No promo twice", BiteLevel.INTERMEDIATE),
... ]
>>> sorted(bites)
...
TypeError: '<' not supported between instances of 'Bite' and 'Bite'

>>> @dataclass(order=True)  # now comparison works out of the box
```

Explanation

If you add `order=True` to the `dataclass` decorator (`False` by default), `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` methods will be generated. These compare the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type.

Or, we can sort `bites` using `sorted(bites, key=operator.attrgetter('level'))`. Here, we again need `IntEnum` (see the docs below and Tip #162).

Resources

https://docs.python.org/3/library/dataclasses.html
https://docs.python.org/3/library/enum.html#intenum

217. Create an iterator

Here is how to make an iterator in Python:

```
>>> class Halving:
...     def __init__(self, start):
...         self.number = start
...     def __iter__(self):
...         return self
...     def __next__(self):
...         self.number /= 2
...         if self.number < 10:
...             raise StopIteration
...         return self.number
...
>>> ha = Halving(50)
>>> next(ha)
25.0
>>> next(ha)
12.5
>>> next(ha)
...
StopIteration
>>> for number in Halving(50):
...     print(number)
...
25.0
12.5
```

Explanation

Python's iterator protocol dictates that an object is "iterable" if it implements the `__iter__()` and `__next__()` methods. Here we have the constructor receive a starting number and halve it upon each iteration. Until it gets below `10`, then we raise `StopIteration` which effectively stops the iteration. Interestingly using the iterator inside a `for` loop, it catches this exception for you nice and cleanly.

Resources

https://docs.python.org/3/library/stdtypes.html#iterator-types
https://treyhunner.com/2018/06/how-to-make-an-iterator-in-python/

218. How to implement slicing

Here is to add support for slicing to your objects:

```
>>> class A:
...     def __init__(self, items):
...         self._items = items
...
>>> a = A([1,2,3])
>>> a[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'A' object is not subscriptable
>>> class A:
...     def __init__(self, items):
...         self._items = items
...     def __getitem__(self, index):
...         return self._items[index]
...
>>> a = A([1,2,3])
>>> a[0]
1
>>> a[-1]
3
>>> a[1:]
[2, 3]
```

Explanation

In order to support indexing / slicing on your objects implement the `__getitem__()` special method. This works because under the hood `container[index]` gets evaluated to `container.__getitem__(index)`.

Resources

https://stackoverflow.com/a/43627489
https://docs.python.org/3/reference/datamodel.html#special-method-names

219. Max and min builtins

The `min` and `max` builtin functions can save you some looping:

```
>>> import random
>>> numbers = random.sample(range(1, 100), k=10)
>>> numbers
[32, 68, 47, 11, 36, 22, 31, 67, 27, 2]
>>> min(numbers)
2
>>> max(numbers)
68
>>> min(numbers, key=str)
11
>>> min([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: min() arg is an empty sequence
>>> min([], default=1)
1
```

Explanation

No need to loop over `numbers` (we met `random.sample` in Tip #13) keeping track of the minimum / maximum value, `min` and `max` builtins calculate this in one go.

You can give any callable to the optional `key` argument, so if we give `min`'s `key` argument `str`, then `11` becomes the minimum instead of `2`.

There is also an optional default argument which is returned in case the iterable passed in is empty. Instead of writing `max(items) if items else 1`, you can write `max(items, default=1)` which is more concise.

Resources

https://docs.python.org/3/library/functions.html#min
https://docs.python.org/3/library/functions.html#max

220. Go to a position in a file

You can use `seek` on a file object to go to a certain position in the file:

```
>>> content = "line 1\nline 2\nline 3\nline 4\n"
>>> with open("file1.txt", "w") as f:
...     f.write(content.strip())
...
27
>>> with open("file1.txt", "r") as f:
...     f.readline()
...     f.tell()
...     f.seek(0)
...     f.tell()
...     f.readline()
...     f.seek(15)
...     f.read()
...
'line 1\n'
7
0
0
'line 1\n'
15
'ine 3\nline 4'
```

Explanation

First we write 4 lines to a file. Then we open the same file for reading and read the first line.

We then use `tell()` on the file object to see we are at position 7 of the current "stream".

Then we go to position 0 using `seek(0)` (compare this to rewinding an old cassette tape) and use `tell` to confirm we are at the start again, and read the same line. Lastly we go back to byte 15, and `read()` the remaining file which yields `ine 3\nline 4`.

Resources

https://docs.python.org/3/tutorial/inputoutput.html#methods-of-file-objects

221. Exporting Django model data (PyPI)

Here is how to export Django model data to a JSON file and load it in:

```
(venv) $ python manage.py dumpdata books.Book > books.json


# REPL
>>> import json
>>> from pprint import pprint as pp
>>> with open('books.json') as f:
...     books = json.loads(f.read())
...
>>> type(books)
<class 'list'>
>>> first_book = books[0]
>>> pp(first_book)
{'fields': {'added': '2020-11-27T15:29:23.924Z',
            'authors': 'Gary Keller',
            'cover': 'https://images-na.ssl-images-
amazon.com/images/I/31bLXJwHXlL._SX344_BO1,204,203,200_.jpg',
            'description': 'The One Thing explains the success habit to ...',
            'link': 'https://www.amazon.com/ONE-Thing-Surprisingly-
Extraordinary-Results/dp/1885167776',
            'page_count': 240,
            'publisher': 'John Murray Press',
            'title': 'The One Thing'},
 'model': 'books.book',
 'pk': 1}
```

Explanation

First we use Django's `manage.py` with its `dumpdata` switch to dump the data of one of our app's models.

In Unix we can use `>` to redirect standard output to a file. Next we enter in the REPL and load this file in using `json.loads` (seen before in Tips #84 and #200) and pretty print the first book in the resulting `list` using `pprint` (see Tip#145).

To do the inverse, populating a model with data stored from a JSON file, you can use `manage.py`'s `loaddata` switch.

Resources

https://docs.djangoproject.com/en/3.1/ref/django-admin/#django-admin-dumpdata

222. Data validation with pydantic (PyPI)

`pydantic` is an amazing library for data validation and settings management using Python type annotations:

```
>>> from datetime import datetime
>>> from typing import Optional
>>> from pydantic import BaseModel, EmailStr  # pip install pydantic[email]
>>> class User(BaseModel):
...     id: int
...     username: str
...     email: EmailStr
...     added: Optional[datetime] = datetime.now()
...
>>> User(id=1, username='pybob', email='bob@pybit.es')
User(id=1, username='pybob', email='bob@pybit.es',
added=datetime.datetime(2020, 12, 30, 9, 42, 6, 570021))
>>> User(id=2, username='pybob')
Traceback (most recent call last):
...
pydantic.error_wrappers.ValidationError: 1 validation error for User
email
  field required (type=value_error.missing)
>>> User(id=2, username='pybob', email='pybit.es')
...
pydantic.error_wrappers.ValidationError: 1 validation error for User
email
  value is not a valid email address (type=value_error.email)
>>> User(id=2, username='pybob', email='bob@pybit.es')
User(id=2, username='pybob', email='bob@pybit.es',
added=datetime.datetime(2020, 12, 30, 9, 42, 6, 570021))
```

Explanation

First we define a `User` model inheriting from `BaseModel` defining attributes with type hints (`EmailStr` requires the `email-validator` extension). If `added` is not provided it will default to `datetime.now()`. Then the fun starts: every time we try to make a `User` without respecting defined requirements `pydantic` throws a `ValidationError`. Take note of this library, it's used in important (modern) Python projects like FastAPI.

Resources

https://pydantic-docs.helpmanual.io/

223. Python statistics

Do you need statistics in Python? Since version 3.4 there is a dedicated Standard Library module for this:

```
>>> import statistics
>>> data = [13, 17, 23, 15, 21, 6, 18, 37, 25, 23, 47, 19, 26, 24, 17, 24, 83,
8, 15, 29]
>>> statistics.__all__
['NormalDist', 'StatisticsError', 'fmean', 'geometric_mean', 'harmonic_mean',
'mean', 'median', 'median_grouped', 'median_high', 'median_low', 'mode',
'multimode', 'pstdev', 'pvariance', 'quantiles', 'stdev', 'variance']
>>> sum(data) / len(data)
24.5
>>> statistics.mean(data)
24.5
>>> statistics.median(data)
22.0
>>> statistics.mode(data)
17
>>> statistics.stdev(data)
16.602155852399665
>>> statistics.variance(data)
275.63157894736844
>>> statistics.quantiles(data)
[15.5, 22.0, 25.75]
```

Explanation

We define some numbers in a `list` called `data`. We check what functions the module makes available to us using `__all__` (Tip #77) and call various of them.

Looking at the source of `statistics` you see some interesting patterns we discussed before: use of doctests (Tip #156), raising custom exceptions, e.g. `StatisticsError` (Tip #174) and use of `assert`s (Tip #191).

Resources

https://docs.python.org/3/library/statistics.html

224. Disassembling bytecode

The dis module supports the analysis of CPython bytecode by disassembling it:

```
>>> def hello(name=None):
...     name = 'Stranger' if name is None else name
...     return f"Hello {name}"
...
>>> import dis
>>> dis.dis(hello)
  2           0 LOAD_FAST                0 (name)
              2 LOAD_CONST               0 (None)
              4 IS_OP                    0
              6 POP_JUMP_IF_FALSE       12
              8 LOAD_CONST               1 ('Stranger')
             10 JUMP_FORWARD             2 (to 14)
        >>   12 LOAD_FAST                0 (name)
        >>   14 STORE_FAST               0 (name)

  3          16 LOAD_CONST               2 ('Hello ')
             18 LOAD_FAST                0 (name)
             20 FORMAT_VALUE             0
             22 BUILD_STRING             2
             24 RETURN_VALUE
>>> list(hello.__code__.co_code)
[124, 0, 100, 0, 117, 0, 114, 12, 100, 1, 110, ...]
>>> dis.opname[124]
'LOAD_FAST'
```

Explanation

We wrote a little function called `hello` passing it to `dis.dis` which explains its bytecode. What do the numbers mean? As per the StackOverflow below, the numbers on the far left are the line numbers in the source code from which this byte code was compiled. The numbers in the column on the left are the offset of the instruction within the bytecode, and the numbers on the right are the opargs. Another great resource is the "A Python Interpreter Written in Python" chapter in the 500 Lines or Less AOSA book.

Resources

https://www.aosabook.org/en/500L/a-python-interpreter-written-in-python.html
https://stackoverflow.com/a/12673195

225. Pipdeptree (PyPI)

Here is a nice tool to easily see what dependencies depend on each other:

```
(venv) $ pip show requests
...
Requires: chardet, urllib3, certifi, idna
Required-by: tinys3, social-auth-core, requests-oauthlib, PyGithub

# pipdeptree gives more info:
(venv) $ pipdeptree -p requests
requests==2.25.1
  - certifi [required: >=2017.4.17, installed: 2020.6.20]
  - chardet [required: >=3.0.2,<5, installed: 3.0.4]
...

# list the tree in reverse = what other project dependencies are using
requests?
(venv) $ pipdeptree -r -p requests
requests==2.25.1
  - PyGithub==1.54.1 [requires: requests>=2.14.0]
  - requests-oauthlib==1.3.0 [requires: requests>=2.0.0]
...

# it even warns you about conflicting dependencies:
(venv) $ pipdeptree -p Django
Warning!!! Possibly conflicting dependencies found:
* Django==3.1.5
 - sqlparse [required: >=0.2.2, installed: 0.1.0]
```

Explanation

Although `pip show` shows that `requests` (Tip #39) requires `chardet`, `urllib3`, `certifi` and `idna`, and that it's required for `tinys3`, `social-auth-core`, `requests-oauthlib`, and `PyGithub`, it does not show version info.

Enter `pipdeptree` (PyPI), using it with `-p PACKAGES` it shows the dependencies of given package(s) in a nice tree format including which specific version is required vs. installed for each one. Very useful! You can also use `poetry` (linked below) to make sure all dependencies align.

Resources

https://stackoverflow.com/a/30450999
https://github.com/python-poetry/poetry

226. Rounding numbers

Different ways of rounding in Python:

```
>>> numbers = [0.5, 1.5, 2.5]
>>> for num in numbers:
...     num, round(num)
...
(0.5, 0)
(1.5, 2)
(2.5, 2)  # rounds "half even" aka "bankers rounding"
>>> import math
>>> for num in numbers:
...     num, math.floor(num), math.ceil(num)
...
(0.5, 0, 1)
(1.5, 1, 2)
(2.5, 2, 3)
>>> from decimal import Decimal, getcontext, ROUND_FLOOR
>>> getcontext().rounding = ROUND_FLOOR  # default ROUND_HALF_EVEN
>>> for num in numbers:
...     num, Decimal(num).quantize(Decimal("1"))
...
(0.5, Decimal('0'))
(1.5, Decimal('1'))
(2.5, Decimal('2'))
```

Explanation

Interestingly `round` uses bankers rounding, that is to the nearest even integer. This is also the the default of the `decimal` module (`ROUND_HALF_EVEN`), but it can be tweaked by setting `decimal.getcontext().rounding`. You can also use the `math`'s `ceil` and `floor` module to round up or down.

Resources

https://docs.python.org/3/library/functions.html#round
https://docs.python.org/3/library/decimal.html

227. What day of the year is it?

How much of the year did we progress? Use `timetuple`:

```
>>> from datetime import date
>>> date.today()
datetime.date(2021, 1, 11)
>>> date.today().timetuple()
time.struct_time(tm_year=2021, tm_mon=1, tm_mday=11, tm_hour=0,
                 tm_min=0, tm_sec=0, tm_wday=0, tm_yday=11,
                 tm_isdst=-1)
>>> date.today().timetuple().tm_yday
11
>>> day_in_july = date(2021, 7, 14)
>>> day_in_july.timetuple()
time.struct_time(tm_year=2021, tm_mon=7, tm_mday=14, tm_hour=0,
                 tm_min=0, tm_sec=0, tm_wday=2, tm_yday=195,
                 tm_isdst=-1)
>>> day_in_july.timetuple().tm_yday
195
# another way:
>>> int(day_in_july.strftime('%j'))
195
```

Explanation

The `datetime.timetuple()` converts your `datetime` (`date`) object to a `time.struct_time` which has some interesting attributes, one being the `tm_yday` property, which shows the day of the year. Another way is using `strftime` with `%j` but it returns a `str` so you need to convert it to an `int` to get the same result.

Resources

https://docs.python.org/3/library/datetime.html#datetime.datetime.timetuple
https://stackoverflow.com/a/623312

228. Detect accented characters

You can use the Unicode Database (`unicodedata`) module for this purpose:

```
>>> from unicodedata import decomposition
>>> for c in 'Cataluña':
...     c, decomposition(c)
...
('C', '')  # = no character decomposition mapping
('a', '')
('t', '')
('a', '')
('l', '')
('u', '')
('ñ', '006E 0303')  # bingo, this is an accented letter
('a', '')
>>> text = ("La capital de Cataluña, es la ciudad más visitada "
...         "de España y la segunda más poblada.")
# use enumerate to get the indices or positions of the matches
>>> positions = [i for i, c in enumerate(text) if decomposition(c)]
>>> for pos in positions:
...     pos, text[pos]
...
(20, 'ñ')
(38, 'á')
(57, 'ñ')
(74, 'á')
```

Explanation

If a character is accented `unicodedata.decomposition` returns its character decomposition mapping. We use this technique here to match the Spanish accented `ñ` and `á` characters. And we use `enumerate` again (see Tip #7) to match the positions in the text string. Oh and this is all Standard Library by the way :)

Resources

https://docs.python.org/3/library/unicodedata.html

229. Testing floating point numbers

Sometimes you need a bit of tolerance in your tests, for example when dealing with `float`s:

```
$ more script.py
def sum_numbers(*numbers):
    return sum(numbers)

$ more test.py
from script import sum_numbers

def test_sum_numbers_ints():
    assert sum_numbers(1, 2, 3) == 6

def test_sum_numbers_floats():
    assert sum_numbers(0.1, 0.2) == 0.3  # uh-oh

$ pytest test.py
...
E       assert 0.30000000000000004 == 0.3
E        +  where 0.30000000000000004 = sum_numbers(0.1, 0.2)
...
1 failed, 1 passed in 0.06s

$ more test.py
from pytest import approx
...
def test_sum_numbers_floats():
    assert sum_numbers(0.1, 0.2) == approx(0.3)  # this passes
```

Explanation

`pytest`'s `approx` asserts that two numbers (or two sets of numbers) are equal to each other within some tolerance. Here we see a good example of `float`'s inherent imprecision and the trouble it may cause in testing. But no worries, `approx` asserts that `0.30000000000000004` equals `0.3`.

Resources

https://docs.pytest.org/en/latest/reference.html#pytest-approx
https://cs50.stackexchange.com/a/15645

230. How to copy / remove a directory

Use `shutil` for some handy high-level file operations like copying a directory:

```
>>> import os
>>> from pathlib import Path
>>> import shutil
>>> homework = Path("homework")
>>> os.makedirs(homework)
>>> for i in range(1, 4):
...     open(homework / f"file{i}", 'w').close()
...
>>> os.listdir(homework)
['file3', 'file2', 'file1']
>>> homework2 = Path("homework2")
>>> shutil.copytree(homework, homework2)
PosixPath('homework2')
>>> os.listdir(homework2)
['file3', 'file2', 'file1']
>>> shutil.rmtree(homework2)  # be very careful with this command, it doesn't
ask you first!
>>> homework2.is_dir()
False
# some other fun ones (all use os under the hood)
>>> shutil.which("python3")
'/Library/Frameworks/Python.framework/Versions/3.9/bin/python3'
>>> shutil.disk_usage("/")  # os.statvfs
usage(total=1027680514048, used=15045521408, free=324004749312)
>>> shutil.get_terminal_size()  # os.get_terminal_size
os.terminal_size(columns=114, lines=37)
```

Explanation

First we create a `homework` directory and write 3 empty files into it (see the Stack Overflow for this recipe). We confirm with `os.listdir` that the files are created, then use `shutil.copytree` to copy the `homework` directory to `homework2`. We use `os.listdir` again to confirm this worked. Then we remove the `homework2` directory and confirm it's no longer there. Lastly, we show some other cool `shutil` functions.

Resources

https://docs.python.org/3/library/shutil.html
https://stackoverflow.com/a/12654798

231. Setting attributes

You can use the `setattr` built-in function to assign attributes to class instances / objects:

```
>>> s = "name: PyBites\norigin: Worldwide\nborn: 2016-12-19"
>>> class MyClass:
...     pass
...
>>> mc1 = MyClass()
>>> for line in s.splitlines():
...     key, value = line.split(": ")
...     mc1.key = value
...
# oops!
>>> [attr for attr in mc1.__dir__() if not attr.startswith('__')]
['key']
# the proper way:
>>> mc2 = MyClass()
>>> for line in s.splitlines():
...     key, value = line.split(": ")
...     setattr(mc2, key, value)
...
>>> [attr for attr in mc2.__dir__() if not attr.startswith('__')]
['name', 'origin', 'born']
>>> mc2.name, mc2.origin, mc2.born
('PyBites', 'Worldwide', '2016-12-19')
```

Explanation

Here we define a string `s` with some "key, value" pairs, separated by newlines (`\n`). Next we define a class called `A` and instantiate it into `a`.

We split the `s` string by newline and try to set each "key, value" pair dynamically but it fails. Hence we make another instance of (dummy) class B, now using `setattr` which works now.

Resources

https://docs.python.org/3/library/functions.html#setattr

232. Parse an XML Feed (PyPI)

Here we parse our XML blog feed identifying most commonly used categories:

```
>>> from collections import Counter
>>> import xml.etree.ElementTree as ET
>>> import requests
>>> pybites_rss_feed = "https://pybit.es/feeds/all.rss.xml"
>>> resp = requests.get(pybites_rss_feed)
>>> tree = ET.fromstring(resp.content)
>>> categories = (e.text for e in tree.findall("./channel/item/category"))
>>> most_comon_categories = Counter(categories).most_common(10)
>>> print(*most_comon_categories, sep='\n')
('learning', 102)
('news', 98)
('twitter', 96)
('python', 88)
('tips', 74)
('codechallenges', 72)
('Flask', 68)
('pybites', 67)
('Django', 54)
('data science', 38)
```

Explanation

First we use `requests` to load in our feed, passing the content of its response into `xml.etree.ElementTree`'s `fromstring` function.

Then we create a generator expression of article categories (picked up in our platform's forum - thanks @jcass77). Generator expressions are a high performance, memory efficient generalization of list comprehensions, see the PEP that introduced them below.

Lastly we feed this `categories` generator into `collections.Counter` (see Tip #14) retrieving the most common ones and using a single print to print out the results (see Tip #34).

Resources

https://docs.python.org/3/library/xml.etree.elementtree.html
https://www.python.org/dev/peps/pep-0289/

233. Set up recurring datetimes (PyPI)

When to host our Tuesday / Thursday coaching calls?

```
>>> from dateutil.rrule import rrule, WEEKLY, SU, TU, TH
>>> from dateutil.parser import parse
>>> from pprint import pprint as pp
>>> dates = rrule(WEEKLY, count=10, wkst=SU, byweekday=(TU,TH),
...               dtstart=parse("20210118T160000"))
>>> pp(list(dates))
[datetime.datetime(2021, 1, 19, 16, 0),
 datetime.datetime(2021, 1, 21, 16, 0),
 datetime.datetime(2021, 1, 26, 16, 0),
 datetime.datetime(2021, 1, 28, 16, 0),
 datetime.datetime(2021, 2, 2, 16, 0),
 datetime.datetime(2021, 2, 4, 16, 0),
 datetime.datetime(2021, 2, 9, 16, 0),
 datetime.datetime(2021, 2, 11, 16, 0),
 datetime.datetime(2021, 2, 16, 16, 0),
 datetime.datetime(2021, 2, 18, 16, 0)]
```

Explanation

Here we use it to get our weekly Tuesday and Thursday coaching calls for the coming 5 weeks (ok, we assume the same 4PM time for both to keep it simple).

We use `dateutil.parser.parse` again (see Tip #85) to convert a string to a `datetime` object, here to specify a start offset (= `dtstart` argument).

This is really cool. If you think about how you set up recurring calendar appointments, you'll soon realize how much coding this `dateutil` module can save you!

Resources

https://dateutil.readthedocs.io/en/stable/rrule.html
https://dateutil.readthedocs.io/en/stable/index.html

234. How to swapcase all vowels in a text

You can use `str.maketrans` and `str.translate` to apply a mapping of replacements:

```
>>> vowels = 'aeiou'
>>> text = "It's Friday evening, which means X-FILES night"
>>> table = {c: c.swapcase() for c in vowels + vowels.upper()}
>>> table
{'a': 'A', 'e': 'E', 'i': 'I', 'o': 'O', 'u': 'U', 'A': 'a', 'E': 'e',
 'I': 'i', 'O': 'o', 'U': 'u'}

>>> translation = text.maketrans(table)
# keys are outputs of ord() = integer representation of (Unicode) characters
# so ord('A') -> 65, ord('E') -> 69, etc.
>>> translation
{97: 'A', 101: 'E', 105: 'I', 111: 'O', 117: 'U', 65: 'a', 69: 'e',
 73: 'i', 79: 'o', 85: 'u'}

# apply the translation mapping, effectively "swapcasing" all vowels
>>> text.translate(translation)
"it's FrIdAy EvEnIng, whIch mEAns X-FiLeS nIght"

# an alternative: using join + a generator expression
>>> "".join(table.get(c, c) for c in text)
"it's FrIdAy EvEnIng, whIch mEAns X-FiLeS nIght"
```

Explanation

Here we define a string called `text` and make a `table` mapping (`dict`) of all vowels with their swapcase'd version, both upper and lower case. The `swapcase()` string method is convenient for this.

Then we make a `translation` (`dict`) using the `maketrans` method that converts characters to their integer representations. This is needed for the `translate` method we pass this mapping to next.

An alternative is using the `table` dictionary directly, looping over the characters one by one. As this is a `dict`, you can use its `.get()` method (see Tip #91). Next we piece all characters back together using `join`, which can receive a generator expression. As opposed to a list comprehension (Tip #18) generator expressions apply "lazy loading", that is they load in the values one by one which makes them more memory efficient.

Resources

https://docs.python.org/3/library/stdtypes.html#str.translate
https://docs.python.org/3/library/stdtypes.html#str.maketrans

235. How to validate an IP address

You can use the `ipaddress` or `socket` module to validate IP addresses:

```
>>> import ipaddress
>>> import socket
>>> ip_addresses = ("192.168.0.2", "127.0.0.1", "192.168.1.257", "1.0x2.3.4",
...                 "2001:0db8:85a3:0000:0000:8a2e:0370:7334")
>>> def valid_ip(ip: str) -> bool:
...     try:
...         ipaddress.ip_address(ip)
...         return True
...     except ValueError:
...         return False
...
>>> def valid_ip4_addr(ip: str) -> bool:
...     try:
...         socket.inet_pton(socket.AF_INET, ip)
...         return True
...     except socket.error:
...         return False
...
>>> for ip in ip_addresses:
...     print(f"{ip:<40} | {str(valid_ip(ip)):<5} | {valid_ip4_addr(ip)}")
...
192.168.0.2                              | True  | True
127.0.0.1                                | True  | True
192.168.1.257                            | False | False
1.0x2.3.4                                | False | False
2001:0db8:85a3:0000:0000:8a2e:0370:7334  | True  | False
```

Explanation

Don't go with a regular expression here, leverage the Standard Library! Using the `ipaddress` module is the most straightforward. You can pass it both IPv4 and IPv6 addresses. We also show you how to do it with the `socket` module in case you want to look at a particular address family, for example `AF_INET` (v4 IP addresses). Lastly, the string justifying we learned in Tip #196 comes in handy again to print a table of IPs and if they are valid using both functions.

Resources

https://docs.python.org/3/library/ipaddress.html#ipaddress.ip_address
https://docs.python.org/3/library/socket.html#socket.inet_pton

236. Requests exception handling (PyPI)

The `Response` class of the `requests` module has a convenient method (abstraction) to raise an `HTTPError` exception:

```
>>> import requests
# example from requests docs
>>> bad_r = requests.get('https://httpbin.org/status/404')
>>> bad_r.status_code
404
>>> bad_r.raise_for_status()
...
requests.exceptions.HTTPError: 404 Client Error

# good response
>>> good_r = requests.get('https://httpbin.org/status/200')
>>> good_r.raise_for_status()
>>>
# requests source > models.py
def raise_for_status(self):
    """Raises :class:`HTTPError`, if one occurred."""
    http_error_msg = ''
    ...
    if 400 <= self.status_code < 500:
        http_error_msg = u'%s Client Error: %s for url: %s' %
(self.status_code, reason, self.url)

    elif 500 <= self.status_code < 600:
        http_error_msg = u'%s Server Error: %s for url: %s' %
(self.status_code, reason, self.url)

    if http_error_msg:
        raise HTTPError(http_error_msg, response=self)
```

Explanation

If you want a `requests` response to raise an exception for a bad (non-200) status code, use `response.raise_for_status()`.

Resources

https://requests.readthedocs.io/en/master/user/quickstart/#response-status-codes
https://github.com/psf/requests/blob/master/requests/models.py

237. Organizing pytest fixtures (PyPI)

You can use a `conftest.py` module to add fixtures you want to reuse across modules in the same package:

```
# https://github.com/PyBites-Open-Source/pysource
# conftest.py
...
...
SOURCE_CODE = dict(
    choice=choice, match=match,
    getsource=getsource,
    this=this, capwords=capwords
)

@pytest.fixture(scope='module')
def source_code():
    return SOURCE_CODE

# this fixture can now be used in another test modules
# test_pysource.py
...
@pytest.mark.parametrize("func", [
    choice, match, ...
])
def test_print_source(func, capfd, source_code):
    print_source(func)
    ...
    assert actual == expected
```

Explanation

You can have per-directory fixture scopes by placing fixture functions in a `conftest.py` file. This avoids possible setup code repetition and makes your test modules leaner.

Resources

https://docs.pytest.org/en/stable/example/simple.html#package-directory-level-fixtures-setups
https://github.com/PyBites-Open-Source/pysource/tree/main/tests

238. PYTHONSTARTUP environment variable

You can set `PYTHONSTARTUP` to a Python script file containing commands that will run upon entering the REPL:

```
$ cat ~/bin/pystartup.py
import os
from pprint import pprint as pp
import sys

sys.path.append(os.path.join(os.path.expanduser("~"), 'bin'))

$ export PYTHONSTARTUP=~/bin/pystartup.py

# this script should be importable now
$ ls ~/bin/c*py
/Users/bobbelderbos/bin/chaining.py

$ python3
>>> pp(dir('a string'))  # pp is available
['__add__',
 '__class__',
 '__contains__',
...
>>> sys.path[-1]
'/Users/bobbelderbos/bin'
>>> import chaining  # works as importable from ~/bin
```

Explanation

Here we create a script called `pystartup.py` where we import `pprint` and add `~/bin` to `sys.path` (which contains a list of string paths where Python will look for modules).

Next we set the `PYTHONSTARTUP` environment variable in the shell to this file. Then we enter into the Python REPL and see that `pp` is available and we can import a script located in `~/bin`.

Great tip we discovered on Eric Chou's Network Automation Nerds blog (article linked below).

Resources

https://docs.python.org/3/using/cmdline.html#envvar-PYTHONSTARTUP
https://networkautomation.ninja/p/blog/customize-python-shell-pythonstartup/

239. Parsing URLs into parts

You can use the `urllib.parse` module to break apart URLs and put them back together:

```
>>> from urllib.parse import urlparse, urlunparse, ParseResult
>>> url = "https://tim.blog/2019/03/21/learn-to-code/"

>>> parts = urlparse(url)
>>> parts
ParseResult(scheme='https', netloc='tim.blog',
            path='/2019/03/21/learn-to-code/', params='', query='',
            fragment='')
>>> parts[1]
'tim.blog'
>>> parts.netloc
'tim.blog'

# converting parts into URL and manipulating it:
>>> urlunparse(parts)
'https://tim.blog/2019/03/21/learn-to-code/'
>>> parts.netloc = "pybit.es"
...
AttributeError: can't set attribute
>>> urlunparse(parts._replace(netloc='fourhourworkweek.com',
query='share=twitter'))
'https://fourhourworkweek.com/2019/03/21/learn-to-code/?share=twitter'
```

Explanation

Using `urlparse()` on a URL gives you a `ParseResult` object which is a 6-item named tuple (see Tip #143) which makes it easy to reference parts of the URL.

The inverse is `urlunparse()` which puts a parsed URL (`ParseResult` object) back together again. As this object is immutable and being a `namedtuple`, you can update it using its `_replace()` method.

Resources

https://docs.python.org/3/library/urllib.parse.html

240. Suppressing exceptions

You can suppress exceptions with the `contextlib.suppress` context manager:

```
>>> import os
>>> from contextlib import suppress
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> with suppress(ZeroDivisionError):
...     1/0
...
# from docs
>>> os.remove('someotherfile.tmp')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'someotherfile.tmp'
>>> with suppress(FileNotFoundError):
...     os.remove('someotherfile.tmp')
...
```

Explanation

This context manager suppresses any of the specified exceptions if they occur in the body of a with statement.

This can be useful if you expect certain error conditions to happen which you don't need to handle (where `try..except` is for cases that you do need to handle).

Of course use this with care, because as the Zen of Python (`>>> import this`) says: "Explicit is better than implicit".

Explanation

https://docs.python.org/3/library/contextlib.html#contextlib.suppress

241. Create an entry point to your package

Adding a `__main__.py` file to your package you can call it with `python -m`:

```
# given this simple package:
$ cat my_package/file_1.py
def add_two_numbers(a, b):
    return a + b

# we cannot run it as a package:
$ python -m my_package
... No module named my_package.__main__; 'my_package' is a package and cannot
be directly executed

# adding a __main__.py you can add an entry point to your package:
$ cat my_package/__main__.py
from . import file_1

def foo():
    print(file_1.add_two_numbers(3, 4))

if __name__ == '__main__':
    foo()

# now you can run the package like this:
$ python -m my_package
7
```

Explanation

Similarly to the `if __name__ == "__main__":` entry point for a script (see Tip #46), you can create an entry point to your package by adding a `__main__.py` module to it, making it callable using: `python -m my_package`.

Another way is to add the `entry_points` keyword argument to `setuptools.setup()` in `setup.py` (or `[tool.poetry.scripts]` in `pyproject.toml` if you use `poetry`).

Explanation

https://docs.python.org/3/using/cmdline.html#cmdoption-m
https://python-packaging.readthedocs.io/en/latest/command-line-scripts.html

242. Run a function every 5 minutes (PyPI)

Here is how you can execute a function for a set interval:

```
>>> from datetime import datetime
>>> import time
>>> import sched
>>> s = sched.scheduler(time.time, time.sleep)
>>> def hydrate(sc):
...     print(datetime.now(), "Sip water!")
...     s.enter(300, 1, hydrate, (sc,))
...
>>> s.enter(300, 1, hydrate, (s,))
Event(time=1611405298.4987588, priority=1, action=...
>>> s.run()
2021-01-23 13:34:58.499065 Sip water!
2021-01-23 13:39:58.500204 Sip water!
...
# a nicer way:
>>> import schedule  # PyPI package
>>> def hydrate():
...     print(datetime.now(), "Sip water!")
...
>>> schedule.every(5).minutes.do(hydrate)
Every 5 minutes do hydrate() (... next run: 2021-01-23 14:39:29)
>>> while True:
...     schedule.run_pending()
...     time.sleep(1)
...
2021-01-23 14:39:30.087519 Sip water!
2021-01-23 14:44:30.714231 Sip water!
...
```

Explanation

You can use the `sched` module (Standard Library) for this but it's quite confusing. The `schedule` package uses method chaining (Tip #103) and therefore has a much nicer interface.

Resources

https://schedule.readthedocs.io/en/stable/
https://stackoverflow.com/a/474543

243. Get a random item from a range

The `random` module has a method to choose a random item from `range(start, stop[, step])`:

```
>>> from random import choice, randrange
>>> list(range(10, 101, 10))
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
>>> choice(range(10, 101, 10))
10
>>> choice(range(10, 101, 10))
50
>>> randrange(10, 101, 10)
60
>>> randrange(10, 101, 10)
10
```

Explanation

Sure, `choice(range(10, 101, 10))` does the trick, but `randrange(10, 101, 10)` is more readable and concise.

Resources

https://docs.python.org/3/library/random.html#random.randrange
https://youtu.be/UANN2Eu6ZnM?t=900

244. Function, method or builtin?

Here is how we can inspect if an object is a function, method or builtin:

```
>>> import inspect
>>> import random
>>> class A:
...     def my_method(self):
...         pass
...
>>> a = A()
>>> def func():
...     pass
...
>>> for obj in random.sample, inspect.isbuiltin, any, a.my_method, func, A:
...     if inspect.isfunction(obj):
...         print(f"{obj.__name__} is a function.")
...     elif inspect.ismethod(obj):
...         print(f"{obj.__name__} is a method.")
...     elif inspect.isbuiltin(obj):
...         print(f"{obj.__name__} is a builtin.")
...     else:
...         print(f"I don't know what {obj.__name__} is")
...
sample is a method.
isbuiltin is a function.
any is a builtin.
my_method is a method.
func is a function.
I don't know what A is
```

Explanation

Here we use some convenient functions of the `inspect` module to see if an object is one of the three. This should cover your bases. Another way is to accomplish this is using the `types` module (checking if an object is an instance of `types.MethodType` or `types.FunctionType`) and the `builtins` module (see Tip #167).

Resources

https://docs.python.org/3/library/inspect.html

245. Partial argparse parsing

You can use `argparse.parse_known_args` to accept (and not exit) when extra arguments are present:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(description='Calculate your BMI.')
>>> parser.add_argument("-w", "--weight", type=int, help='Your weight in kg')
_StoreAction(option_strings=['-w', '--weight'], ...
>>> parser.add_argument("-l", "--length", type=int, help='Your length in cm')
_StoreAction(option_strings=['-l', '--length'], ...
>>> parser.parse_args(["-w", '80', "--length", '186'])
Namespace(weight=80, length=186)

# cannot take other args:
>>> try:
...     parser.parse_args(["-w", '80', "--length", '186', "--years", "23"])
... except SystemExit:  # prevent REPL exit
...     pass
...
usage: [-h] [-w WEIGHT] [-l LENGTH]
: error: unrecognized arguments: --years 23

# this way we can:
>>> known, unknown = parser.parse_known_args(["-w", '80', "--length", '186',
"--years", "23"])
>>> known
Namespace(weight=80, length=186)
>>> unknown
['--years', '23']
```

Explanation

The `parse_known_args()` method works like `parse_args()` except that it does not produce an error when extra arguments are present. Instead, it returns a two item tuple containing the populated namespace and the list of remaining argument strings.

Resources

https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser.parse_known_args

246. Timeit decorator

Here we use a decorator to time the duration of a function execution:

```
>>> from functools import wraps
>>> from time import time, sleep

>>> def timing(f):
...     """A simple timer decorator"""
...     @wraps(f)
...     def wrapper(*args, **kwargs):
...         start = time()
...         result = f(*args, **kwargs)
...         end = time()
...         print(f'Elapsed time {f.__name__}: {end - start}')
...         return result
...     return wrapper
...

>>> @timing
... def func():
...     sleep(2)
...     print('Done')
...
>>> func()
Done
Elapsed time func: 2.0039281845092773
```

Explanation

Here we create a timer decorator measuring the start and end timings of a function's execution, printing the duration. As per Tip #97 always use `wraps` on your decorators to preserve your function's meta data (e.g. docstring). We use this (slightly modified) decorator for Bite #62 - "Data structures matter - speed up your Python code".

Resources

https://pybit.es/decorators-by-example.html

247. Make a retry decorator (with optional argument)

Here we make a `retry` decorator that tries to call a function N times before giving up:

```
>>> from functools import wraps, partial
>>> import requests
>>> def retry(func=None, *, times=3):
...     if func is None:
...         return partial(retry, times=times)
...     @wraps(func)
...     def wrapper(*args, **kwargs):
...         attempt = 0
...         while attempt < times:
...             try:
...                 return func(*args, **kwargs)
...             except Exception as exc:
...                 attempt += 1
...                 print(f"Exception {func}: {exc} (attempt: {attempt})")
...         return func(*args, **kwargs)
...     return wrapper
>>> @retry  # or: @retry(times=<int>)
... def get(url):
...     resp = requests.get(url)
...     resp.raise_for_status()
>>> get('https://httpbin.org/status/200')
>>> get('https://httpbin.org/status/404')
Exception <function get at 0x7fb4592dc280>: 404 Client Error: NOT FOUND ...
Exception <function get at 0x7fb4592dc280>: 404 Client Error: NOT FOUND ...
Exception <function get at 0x7fb4592dc280>: 404 Client Error: NOT FOUND ...
...
requests.exceptions.HTTPError: 404 Client Error: NOT FOUND ...
```

Explanation

Here we try to call an endpoint using the `requests` module (Tip #39). If it raises an exception (using `raise_for_status()`, see Tip #236), it tries again, up till `times` attempts. We use `partial` (Tip #75) to have the decorator accept an optional argument. This is quite mind-blowing code which took us various attempts. Luckily we stumbled upon this recipe in Python Cookbook 3rd ed (see all attempts in the article below).

Resources

https://pybit.es/decorator-optional-argument.html

248. Temporarily use a different Decimal context

The `decimal` module has a function called `localcontext()` that returns a context manager to override `decimal`'s current context:

```
>>> from decimal import Decimal, localcontext, ROUND_UP
>>> def calculate_revenue(units_sold, price):
...     return units_sold * Decimal(price)
...
>>> calculate_revenue(47, 6.99)
Decimal('328.5300000000000100186525742')  # default 28 places

# use a different precision within the with statement:
>>> with localcontext() as ctx:
...     ctx.prec = 5
...     calculate_revenue(47, 6.99)
...
Decimal('328.53')

# use different precision and rounding
>>> with localcontext() as ctx:
...     ctx.prec = 5
...     ctx.rounding = ROUND_UP
...     calculate_revenue(47, 6.99)
...
Decimal('328.54')

# outside the with block context is restored:
>>> calculate_revenue(47, 6.99)
Decimal('328.5300000000000100186525742')
```

Explanation

Context managers are great for temporarily overriding behavior (see also Tip #150). The `localcontext()` function (`decimal` module) returns a context manager with a copy of the supplied context. This means you can override the precision (`prec`), `rounding` and other things (see: `decimal.getcontext()` for all attributes) inside the with statement.

Resources

https://docs.python.org/3/library/decimal.html#decimal.localcontext

249. Slice objects

If you need to use a slice more than once you can define and reuse it as a `slice` object:

```
>>> numbers = list(range(1, 11))
>>> numbers
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numbers2 = [num*10 for num in numbers]
>>> numbers2
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# define and reuse:
>>> first_N = slice(0, 2)
>>> numbers[first_N]
[1, 2]
>>> numbers2[first_N]
[10, 20]

# wait, I wanted the first three, no problem, update once:
>>> first_N = slice(0, 3)
>>> numbers[first_N]
[1, 2, 3]
>>> numbers2[first_N]
[10, 20, 30]

# use None to go till the end
>>> last_three = slice(-3, 0)
>>> numbers[last_three]
[]
>>> last_three = slice(-3, None)
>>> numbers[last_three]
[8, 9, 10]
```

Explanation

You can store a `slice` object (`slice(start, stop[, step])`) into a variable for reuse. This might be more readable and could prevent you from updating the same slice multiple times. Note that for for "the last N" you need to use `None` (not `0`) for the `stop` boundary.

Resources

https://docs.python.org/3/library/functions.html#slice

250. Loop through a sequence twice in one go (PyPI)

Use `zip` and `max` to get the maximum uptime in days from a server log:

```
>>> from dateutil.parser import parse
>>> logs = """
... Wed Apr 10 22:39
... Wed Mar 27 16:24
... Wed Mar 27 15:01
... Sun Mar  3 14:51
... Sun Feb 17 11:36
... Thu Jan 17 21:54
... Mon Jan 14 09:25
... """.strip()

>>> log_lines = reversed(logs.splitlines())
>>> reboots = [parse(line.strip()) for line in log_lines]

>>> uptimes = []
>>> for dt1, dt2 in zip(reboots, reboots[1:]):
...     uptime_days = (dt2 - dt1).days
...     uptimes.append(uptime_days)
...
>>> max(uptimes)
30
```

Explanation

First we use `reversed` (Tip #23) to reverse the reboots `list` from oldest to newest. In the same list comprehension (Tip #18) we use `dateutil.parser.parse` (Tip #85) to convert the date strings to `datetime` objects.

Then the interesting part: we use the `zip` builtin (introduced in Tip #3) to loop through the list of reboot `datetimes` twice. By doing `zip(reboots, reboots[1:])` we are effectively comparing each reboot with its previous one, which is useful to calculate the time difference ("uptime") between them.

We store the uptimes in a `list` called `uptimes` and lastly we use the `max` builtin (Tip #219) to get the maximum which is `30` days (`Sun Feb 17 11:36` - `Thu Jan 17 21:54` = `datetime.timedelta(days=30, seconds=49320)`).

Resources

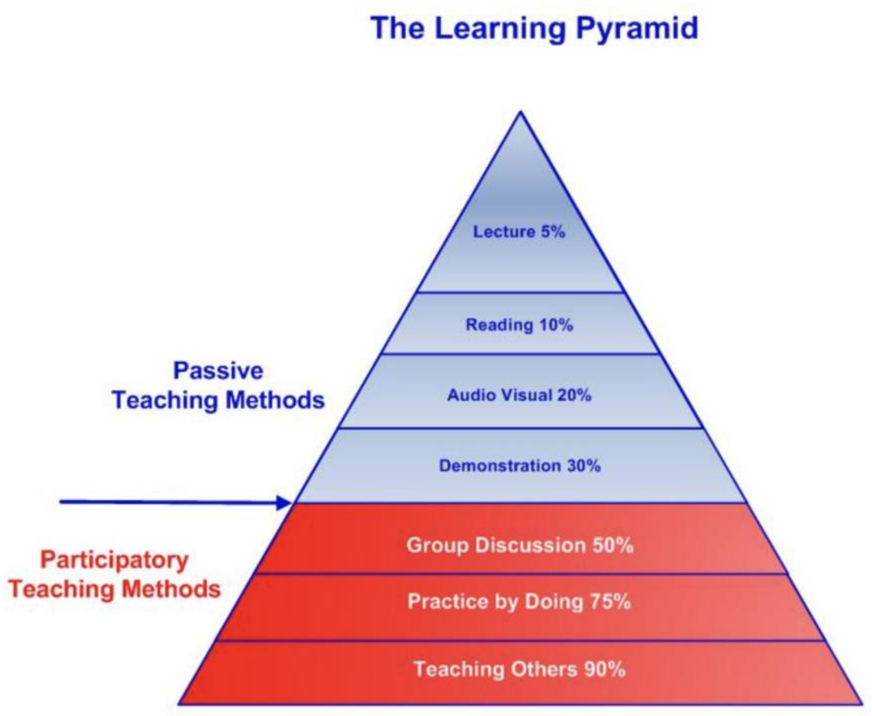
https://docs.python.org/3/library/functions.html#zip

Conclusion

For a non-linear book it's hard to write a conclusion. If you're reading this far, we hope you've picked up a dozen or so techniques that you'll apply in your daily coding.

Remember, reading and consuming is one thing, it's *taking action* that makes a great programmer. Take what you've learned here and *apply it*.

Don't wait too long, the "science" is clear: learning without doing won't get you far:



(source: ThePeakPerformanceCenter.com)

You retain almost 3 times more by practicing what you've learned. So, start applying what you've learned as soon as you stop reading this book.

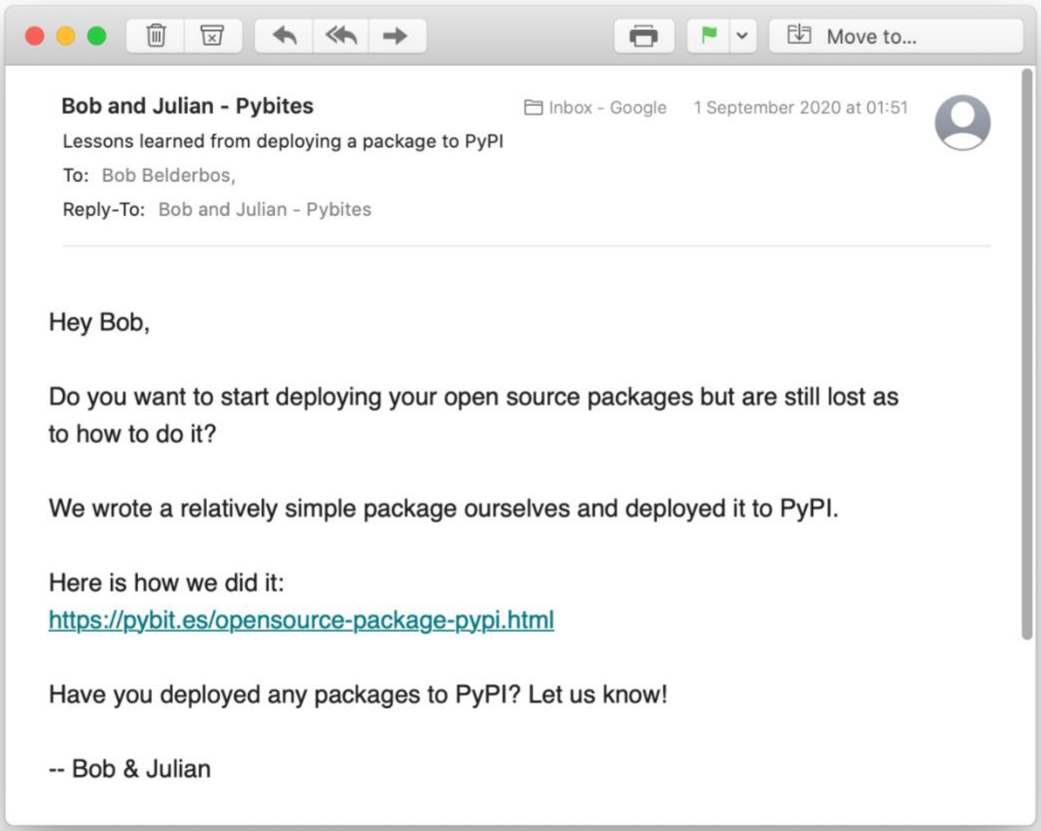
If you're looking for some inspiration, head back to the tips that include links to Python Exercises on our platform and get coding **now**.

Thank you.

-- Bob and Julian

PyBites Friends List

Get 2-3 weekly valuable emails with valuable Python, career and mindset content:
https://pybit.es/friends



Bob and Julian - Pybites

Inbox - Google 1 September 2020 at 01:51

Lessons learned from deploying a package to PyPI

To: Bob Belderbos,

Reply-To: Bob and Julian - Pybites

Hey Bob,

Do you want to start deploying your open source packages but are still lost as to how to do it?

We wrote a relatively simple package ourselves and deployed it to PyPI.

Here is how we did it:
https://pybit.es/opensource-package-pypi.html

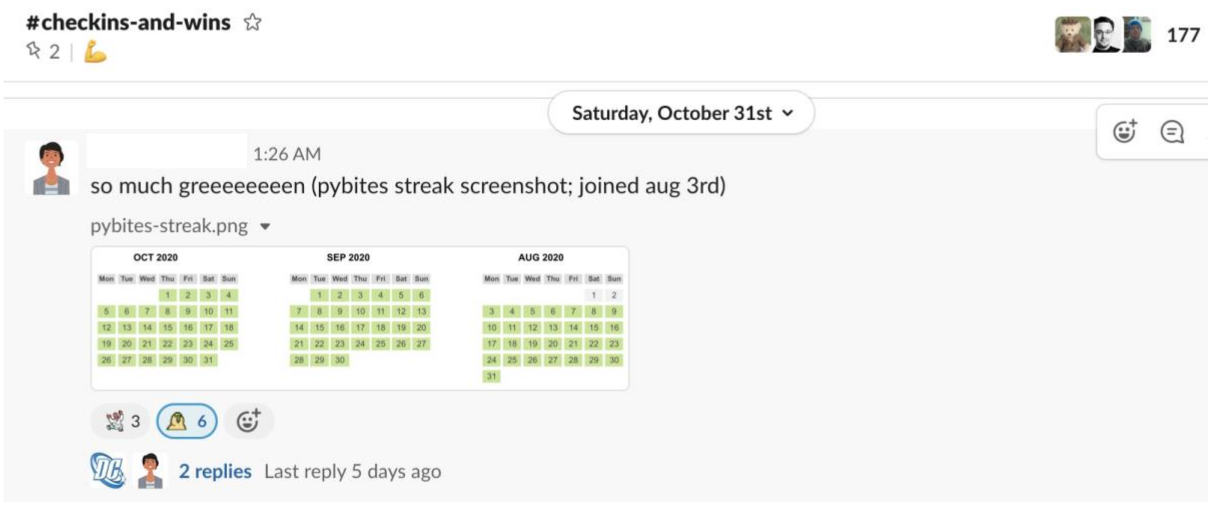
Have you deployed any packages to PyPI? Let us know!

-- Bob & Julian

PyBites Community

We have a thriving community of over two thousand Pythonistas eager to share and learn together.

It’s a beautiful place where extraordinary things happen, and leaders are born. There is a very positive vibe in the group true to the wider and well-known Python community spirit.



#checkins-and-wins

2 | 💪 177

Saturday, October 31st

[redacted] 1:26 AM

so much greeeeeeeen (pybites streak screenshot; joined aug 3rd)

pybites-streak.png

3 6

2 replies Last reply 5 days ago

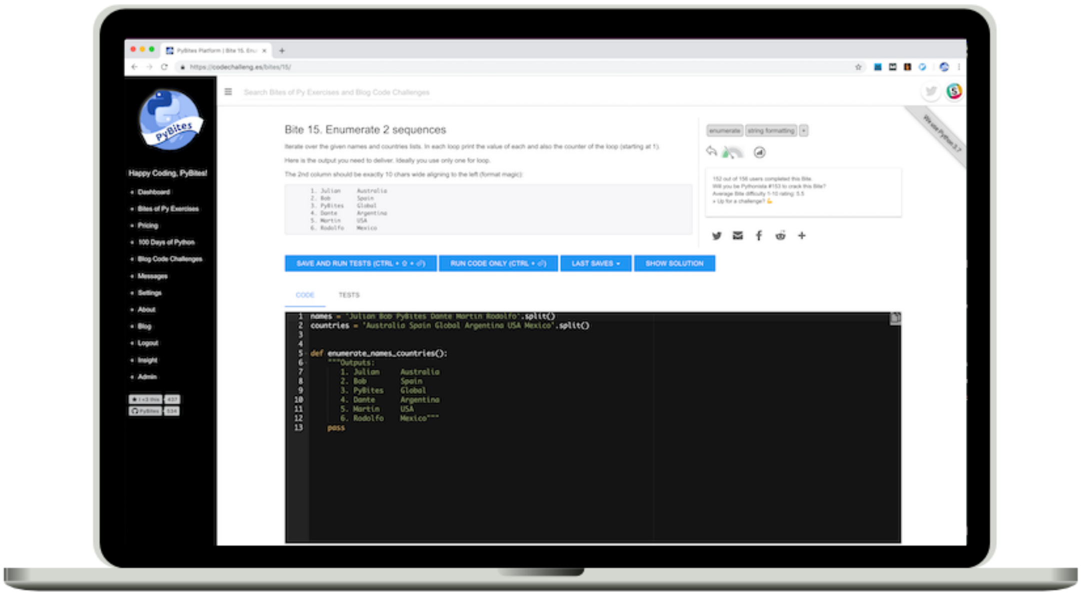
The PyBites Slack Community is now like a second family to me. I now know some of the people on this community better then work colleagues who I've shared an office with for 15 years.
- David C

You don’t have to go at it alone. Join today, it’s totally free: https://pybit.es/community

PyBites Platform

Talking about learning by doing, the best way to do this is [on our coding platform]().

There you will find 300+ real world exercises which people have used to upskill Python, land developer jobs and gain confidence as programmers.



Bite 15. Enumerate 2 sequences

Iterate over the given names and countries lists. In each loop print the value of each and also the counter of the loop (starting at 1).

Here is the output you need to deliver. Ideally you use only one for loop.

The 2nd column should be exactly 10 chars wide aligning to the left (format magic):

```
1. Julian    Australia
2. Bob       Spain
3. PyBites   Global
4. Dante     Argentina
5. Martin    USA
6. Rodolfo   Mexico
```

152 out of 156 users completed this Bite.
Will you be Pythonista #153 to crack this Bite?
Average Bite difficulty 1-10 rating: 5.5
» Up for a challenge? 💪

```python
names = 'Julian Bob PyBites Dante Martin Rodolfo'.split()
countries = 'Australia Spain Global Argentina USA Mexico'.split()


def enumerate_names_countries():
    """Outputs:
       1. Julian     Australia
       2. Bob        Spain
       3. PyBites    Global
       4. Dante      Argentina
       5. Martin     USA
       6. Rodolfo    Mexico"""
    pass
```

I spent about 2 months using sites like Udemy and Codeacademy and while they are good, I've learned more with the challenges here in 3 days than I have in the last 2 months of watching videos and doing very basic exercises. The challenges aren't easy, but they do force you to code, fail, Google, read docs, Stack Overflow, code more, learn and finally solve the problem. PyBites has been immensely helpful.
- Robert M

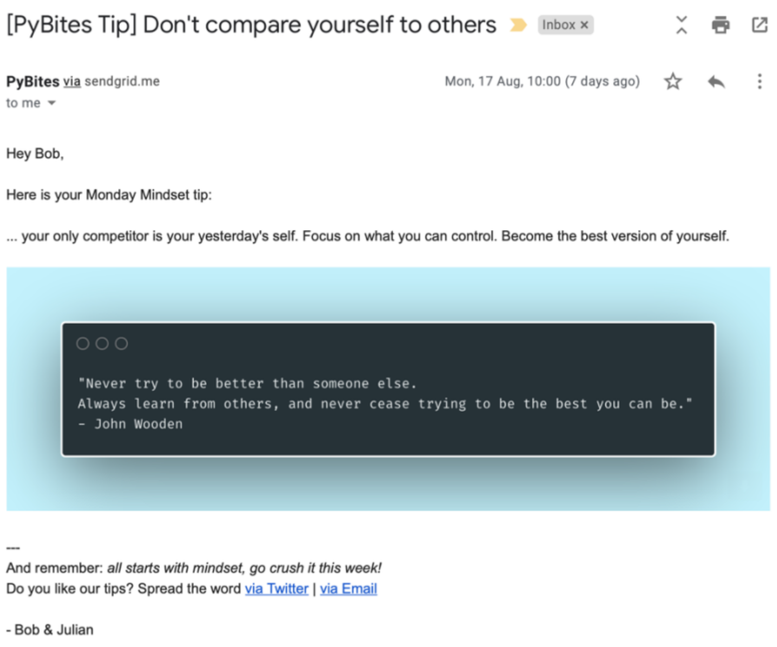
START YOUR FREE 14 DAY TRIAL

Even more tips

Your level of success will seldom exceed your level of personal development, because success is something you attract by the person you become. – Jim Rohn

While we're best known for our Python Tips, did you know we also send out weekly Career and Mindset tips? [Subscribe here](#).

Remember, coding is just one piece of the puzzle. Having the right Mindset is everything.



[PyBites Tip] Don't compare yourself to others

PyBites via sendgrid.me
to me

Mon, 17 Aug, 10:00 (7 days ago)

Hey Bob,

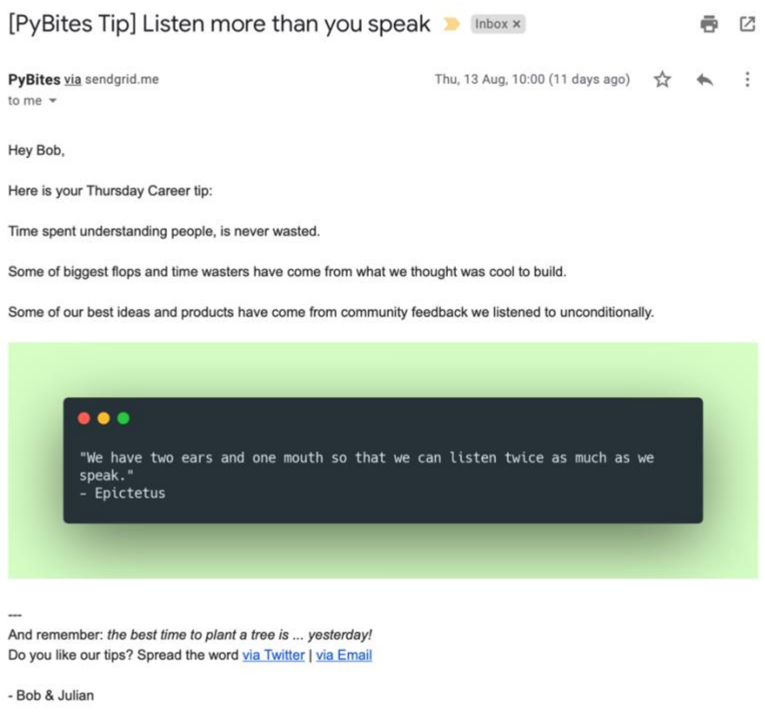
Here is your Monday Mindset tip:

... your only competitor is your yesterday's self. Focus on what you can control. Become the best version of yourself.

```
"Never try to be better than someone else.
Always learn from others, and never cease trying to be the best you can be."
- John Wooden
```

And remember: *all starts with mindset, go crush it this week!*
Do you like our tips? Spread the word via Twitter | via Email

- Bob & Julian



[PyBites Tip] Listen more than you speak

PyBites via sendgrid.me
to me

Thu, 13 Aug, 10:00 (11 days ago)

Hey Bob,

Here is your Thursday Career tip:

Time spent understanding people, is never wasted.

Some of biggest flops and time wasters have come from what we thought was cool to build.

Some of our best ideas and products have come from community feedback we listened to unconditionally.

```
"We have two ears and one mouth so that we can listen twice as much as we
speak."
- Epictetus
```

And remember: *the best time to plant a tree is ... yesterday!*
Do you like our tips? Spread the word via Twitter | via Email

- Bob & Julian

PyBites Workshop

Are you currently using Python and looking for more fulfillment in your job?

Are you looking for a higher paying job where you'll have the flexibility to work anywhere?

Do you feel overwhelmed by the amount of things you have to know as a developer?

This is more common than you might realize. We've been there and have learned a thing or two being successful software developers ourselves.

We can help you get there.

Check out our free PyBites Developer Workshop and learn:

- The common tech industry pitfalls that can seriously hold you back.
- What imposter syndrome is and how it can cripple you from getting out there, sharing your code and contributing to open source.
- How Python is only a relatively small part of the whole picture and that the other career accelerators are less obvious than you might think...

You can watch it here.

Module Index

Here are all the modules used in our tips:

Module	Std Lib / External	Tip number
abc	Standard Lib	96, 177
argparse	Standard Lib	44, 52, 67, 245
ast	Standard Lib	84
asyncio	Standard Lib	152
base64	Standard Lib	200
bisect	Standard Lib	51
boto3	External (PyPI)	172
bs4	External (PyPI)	144, 193
builtins	Standard Lib	167, 168
calendar	Standard Lib	24, 42
collections	Standard Lib	14, 16, 44, 80, 86, 127, 143, 147, 170, 214, 215, 232
concurrent	Standard Lib	123, 125
contextlib	Standard Lib	45, 195, 240
copy	Standard Lib	27
csv	Standard Lib	165, 166
dataclasses	Standard Lib	146, 216
datetime	Standard Lib	42, 50, 85, 95, 99, 100, 101, 108, 116, 143, 159, 160, 170, 191. 222, 227, 242
decimal	Standard Lib	226, 248
difflib	Standard Lib	106, 184, 209
dis	Standard Lib	224
distutils	Standard Lib	47
dj-database-url	External (PyPI)	151
django	External (PyPI)	63, 88, 99, 113, 153, 221
django-extensions	External (PyPI)	98
doctest	Standard Lib	156
emoji	External (PyPI)	135
enum	Standard Lib	162, 216
faker	External (PyPI)	153
feedparser	External (PyPI)	189
flake8	External (PyPI)	164
freezegun	External (PyPI)	95
functools	Standard Lib	75, 97, 107, 183, 246, 247
getpass	Standard Lib	129
gettext	Standard Lib	110
hashlib	Standard Lib	199
heapq	Standard Lib	192
howdoi	External (PyPI)	67
imageio	External (PyPI)	60
importlib	Standard Lib	154
inspect	Standard Lib	96, 129, 154, 244
io	Standard Lib	61
ipaddress	Standard Lib	41, 235
ipykernel	External (PyPI)	71
ipython	External (PyPI)	98
itertools	Standard Lib	12, 28, 29, 30, 43, 92, 94, 137, 138, 145, 149, 187, 209
json	Standard Lib	59, 84, 200, 212, 213, 221

keyword	Standard Lib	167, 169, 170
logging	Standard Lib	76, 174, 186, 207
markdown	External (PyPI)	64
math	Standard Lib	226
operator	Standard Lib	65, 87, 94, 189, 192
os	Standard Lib	44, 83, 123, 125, 129, 157, 172, 214, 230, 238, 240
pandas	External (PyPI)	61
pathlib	Standard Lib	58, 68, 163, 214, 230
pickle	Standard Lib	170
pipdeptree	External (PyPI)	225
platform	Standard Lib	49
playsound	External (PyPI)	188
pprint	Standard Lib	42, 80, 108, 129, 135, 145, 147, 168, 169, 215, 221, 233, 238
pydantic	External (PyPI)	222
Pympler	External (PyPI)	190
pyperclip	External (PyPI)	83
pyppeteer	External (PyPI)	152
pytest	External (PyPI)	81, 90, 130, 140, 202, 207, 211, 229, 237
python-dateutil	External (PyPI)	61, 85, 160, 233, 250
python-decouple	External (PyPI)	151, 213
random	Standard Lib	13, 28, 69, 86, 132, 192, 196, 215, 219, 243, 244
re	Standard Lib	54, 74, 83, 109, 131, 133, 152, 155, 182, 197, 198
requests	External (PyPI)	36, 39, 54, 121, 123, 125, 144, 193, 200, 213, 232, 236, 247
requests_cache	External (PyPI)	36
sched	Standard Lib	242
schedule	External (PyPI)	242
selenium	External (PyPI)	211
setuptools	Standard Lib	119
shlex	Standard Lib	133
shutil	Standard Lib	230
socket	Standard Lib	235
sqlite3	Standard Lib	195
statistics	Standard Lib	223
stdlib_list	External (PyPI)	141
string	Standard Lib	37, 69, 72, 128, 158, 171
subprocess	Standard Lib	173, 174
sys	Standard Lib	33, 119, 152, 187, 188, 204, 238
tempfile	Standard Lib	163
textblob	External (PyPI)	15
textwrap	Standard Lib	26, 48
time	Standard Lib	187, 188, 194, 242, 246
timeit	Standard Lib	86, 124, 183
tqdm	External (PyPI)	125
typing	Standard Lib	117, 208, 222
unicodedata	Standard Lib	228
unittest	Standard Lib	82, 89
urllib	Standard Lib	39, 121, 157, 239
uuid	Standard Lib	21
xml	Standard Lib	232
xmltodict	External (PyPI)	142