Opensource.com

30 hidden gems in Python 3

Three cool features from each of the first ten versions of Python 3

ABOUT OPENSOURCE.COM

What is Opensource.com?

OPENSOURCE.COM publishes stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Submit a story idea: opensource.com/story

Email us: open@opensource.com

Supported by Red Hat

MOSHE ZADKA

MOSHE ZADKA has been involved in the Linux community since 1998, helping in Linux "installation parties". He has been programming Python since 1999, and has contributed to the core Python interpreter. Moshe has been a DevOps/SRE since before those terms existed, caring deeply about software reliability, build reproducibility and other such things. He has worked in companies as small as three people and as big as tens of thousands -- usually some place around where software meets system administration.

Follow me at @moshezadka

CHAPTERS

Introduction

THE RELEASE OF PYTHON 3, a backwards incompatible version of Python, was a news-making event. As Python rose in popularity, every version since has also been an event.

From `async` to the so-called “walrus operator” (the `:=` looks like walrus eyes and teeth), Pythonistas have been atwitter before, after, and during every single major release.

But what about the features that didn’t make the news?

In each one of those releases, there are hidden gems. Small improvements to the standard library. A little improved ergonomics in the interpreter. Maybe even a new operator, one that is not important to be on the front page.

Python 3 has been out since 2008, and it has had ten minor releases between 3.0 and 3.9. Each of those releases packed more features than most people know. Some of those are still little known.

The major challenge is not to find three cool things first released in a new version of Python. It’s not even to find three cool things that few people use. The challenge is how to pick just three from all the delightful options.

Enjoy these curated picks. Here are 30 features, three from each of the first ten versions of Python 3, that you might want to start using.

3 features that debuted in Python 3.0 you should use now

Explore some of the underutilized but still useful Python features.

THIS IS THE FIRST in a series of articles about features that first appeared in a version of Python 3.x. Python 3.0 was first released in 2008, and even though it has been out for a while, many of the features it introduced are underused and pretty cool. Here are three you should know about.

Keyword-only arguments

Python 3.0 first introduced the idea of **keyword-only** arguments. Before this, it was impossible to specify an API where some arguments could be passed in only via keywords. This is useful in functions with many arguments, some of which might be optional.

Consider a contrived example:

```
def show_arguments(base, extended=None, improved=None,
                   augmented=None):
    print("base is", base)
    if extended is not None:
        print("extended is", extended)
    if improved is not None:
        print("improved is", improved)
    if augmented is not None:
        print("augmented is", augmented)
```

When reading code that calls this function, it is sometimes hard to understand what is happening:

```
show_arguments("hello", "extra")
    base is hello
    extended is extra
show_arguments("hello", None, "extra")
    base is hello
    improved is extra
```

While it is possible to call this function with keyword arguments, it is not obvious that this is the best way. Instead, you can mark these arguments as keyword-only:

```
def show_arguments(base, *, extended=None, improved=None,
                   augmented=None):
    print("base is", base)
    if extended is not None:
        print("extended is", extended)
    if improved is not None:
        print("improved is", improved)
    if augmented is not None:
        print("augmented is", augmented)
```

Now, you can't pass in the extra arguments with positional arguments:

```
show_arguments("hello", "extra")
    ---------------------------------------------------------


    TypeError                    Traceback (most recent call last)


    <ipython-input-7-6000400c4441> in <module>
    ----> 1 show_arguments("hello", "extra")



    TypeError: show_arguments() takes 1 positional argument but 2
              were given
```

Valid calls to the function are much easier to predict:

```
show_arguments("hello", improved="extra")
    base is hello
    improved is extra
```

nonlocal

Sometimes, functional programming folks judge a language by how easy is it to write an accumulator. An accumulator is a function that, when called, returns the sum of all arguments sent to it so far.

The standard answer in Python before 3.0 was:

```
class _Accumulator:
    def __init__(self):
        self._so_far = 0
    def __call__(self, arg):
        self._so_far += arg
        return self._so_far

def make_accumulator():
    return _Accumulator()
```

While admittedly somewhat verbose, this does work:

```
acc = make_accumulator()
print("1", acc(1))
print("5", acc(5))
print("3", acc(3))
```

The output for this would be:

```
1 1
5 6
3 9
```

In Python 3.x, **nonlocal** can achieve the same behavior with significantly less code.

```
def make_accumulator():
    so_far = 0
    def accumulate(arg):
        nonlocal so_far
        so_far += arg
        return so_far
    return accumulate
```

While accumulators are contrived examples, the ability to use the nonlocal keyword to have inner functions with state is a powerful tool.

Extended destructuring

Imagine you have a CSV file where each row consists of several elements:

- The first element is a year
- The second element is a month
- The other elements are the total articles published that month, one entry for each day

Note that the last element is *total articles*, not *articles published* per day. For example, a row can begin with:

```
2021,1,5,8,10
```

This means that in January 2021, five articles were published on the first day. On the second day, three more articles were published, bringing the total to 8. On the third day, two more articles were published.

Months can have 28, 30, or 31 days. How hard is it to extract the month, day, and total articles?

In versions of Python before 3.0, you might write something like:

```
year, month, total = row[0], row[1], row[-1]
```

This is correct, but it obscures the format. With **extended destructuring**, the same can be expressed this way:

```
year, month, *rest, total = row
```

This means that if the format ever changes to prefix a description, you can change the code to:

```
_, year, month, *rest, total = row
```

Without needing to add 1 to each of the indices.

What's next?

Python 3.0 and its later versions have been out for more than 12 years, but some of its features are underutilized. In the next article in this series, I'll look at three more of them.

3 features released in Python 3.1 you should use in 2021

Explore some of the underutilized but still useful Python features.

THIS IS THE SECOND in a series of articles about features that first appeared in a version of Python 3.x. Python 3.1 was first released in 2009, and even though it has been out for a long time, many of the features it introduced are underused and pretty cool. Here are three of them.

Thousands formatting

When formatting large numbers, it is common to place commas every three digits to make the number more readable (e.g., 1,048,576 is easier to read than 1048576). Since Python 3.1, this can be done directly when using string formatting functions:

```
"2 to the 20th power is {:,d}".format(2**20)
'2 to the 20th power is 1,048,576'
```

The `,d` format specifier indicates that the number must be formatted with commas.

Counter class

The `collections.Counter` class, part of the standard library module `collections`, is a secret super-weapon in Python. It is often first encountered in simple solutions to interview questions in Python, but its value is not limited to that.

For example, find the five most common letters in the first eight lines of Humpty Dumpty's song [1]:

```
hd_song = """
In winter, when the fields are white,
I sing this song for your delight.
In Spring, when woods are getting green,
I'll try and tell you what I mean.

In Summer, when the days are long,
Perhaps you'll understand the song.

In Autumn, when the leaves are brown,
Take pen and ink, and write it down.
"""
import collections

collections.Counter(hd_song.lower().replace(' ', ''))
  .most_common(5)
[('e', 29), ('n', 27), ('i', 18), ('t', 18), ('r', 15)]
```

Executing packages

Python allows the -m flag to execute modules from the command line. Even some standard-library modules do something useful when they're executed; for example, `python -m cgi` is a CGI script that debugs the web server's CGI configuration.

However, until Python 3.1, it was impossible to execute *packages* like this. Starting with Python 3.1, `python -m package` will execute the `__main__` module in the package. This is a good place to put debug scripts or commands that are executed mostly with tools and do not need to be short.

Python 3.0 was released over 11 years ago, but some of the features that first showed up in this release are cool—and underused. Add them to your toolkit if you haven't already.

Links

[1] http://www2.open.ac.uk/openlearn/poetryprescription/humpty-dumptys-recitation.html

3 Python 3.2 features that are still relevant today

Explore some of the underutilized but still useful Python features.

THIS THE THIRD article in a series about features that first appeared in a version of Python 3.x. Some of those Python versions have been out for a while. For example, Python 3.2 was first released in 2011, yet some of the cool and useful features introduced in it are still underused. Here are three of them.

argparse subcommands

The `argparse` module first appeared in Python 3.2. There are many third-party modules for command-line parsing. But the built-in argparse module is more powerful than many give it credit for..

Documenting all the ins and outs of `argparse` would take its own article series. For a small taste, here is an example of how you can do subcommands with `argparse`.

Imagine a command with two subcommands: `negate`, which takes one argument, and `multiply` which takes two:

```
$ computebot negate 5
-5
$ computebot multiply 2 3
6
import argparse

parser = argparse.ArgumentParser()
subparsers = parser.add_subparsers()
```

The `add_subparsers()` methods creates an object that you can add subcommands to. The only trick to remember is that you need to add what subcommand was called through a `set_defaults()`:

```
negate  = subparsers.add_parser("negate")
negate.set_defaults(subcommand="negate")
negate.add_argument("number", type=float)
multiply  = subparsers.add_parser("multiply")
multiply.set_defaults(subcommand="multiply")
multiply.add_argument("number1", type=float)
multiply.add_argument("number2", type=float)
```

One of my favorite `argparse` features is that, because it separates parsing from running, testing the parsing logic is particularly pleasant.

```
parser.parse_args(["negate", "5"])
    Namespace(number=5.0, subcommand='negate')
parser.parse_args(["multiply", "2", "3"])
    Namespace(number1=2.0, number2=3.0, subcommand='multiply')
```

contextlib.contextmanager

Contexts are a powerful tool in Python. While many use them, writing a new context often seems like a dark art. With the `contextmanager` decorator, all you need is a one-shot generator.

Writing a context that prints out the time it took to do something is as simple as:

```
import contextlib, timeit

@contextlib.contextmanager
def timer():
    before = timeit.default_timer()
    try:
        yield
    finally:
        after = timeit.default_timer()
        print("took", after - before)
```

And you can use it with just:

```
import time

with timer():
    time.sleep(10.5)
    took 10.511025413870811
```

functools.lru_cache

Sometimes the caching results from a function in memory make sense. For example, imagine the classical problem: "How many ways can you make change for a dollar with quarters, dimes, nickels, and cents?"

The code for this can be deceptively simple:

```
def change_for_a_dollar():
    def change_for(amount, coins):
        if amount == 0:
            return 1
        if amount < 0 or len(coins) == 0:
            return 0
        some_coin = next(iter(coins))
        return (
            change_for(amount, coins - set([some_coin]))
            +
            change_for(amount - some_coin, coins)
        )
    return change_for(100, frozenset([25, 10, 5, 1]))
```

On my computer, this takes around 13ms:

```
with timer():
    change_for_a_dollar()
    took 0.013737603090703487
```

It turns out that when you calculate how many ways you can do something like making change from 50 cents, you use the same coins repeatedly. You can use `lru_cache` to avoid recalculating this over and over.

```
import functools

def change_for_a_dollar():
    @functools.lru_cache
    def change_for(amount, coins):
        if amount == 0:
            return 1
        if amount < 0 or len(coins) == 0:
            return 0
        some_coin = next(iter(coins))
        return (
            change_for(amount, coins - set([some_coin]))
            +
            change_for(amount - some_coin, coins)
        )
    return change_for(100, frozenset([25, 10, 5, 1]))
with timer():
    change_for_a_dollar()
    took 0.004180959425866604
```

A three-fold improvement for the cost of one line. Not bad.

Welcome to 2011

Although Python 3.2 was released 10 years ago, many of its features are still cool—and underused. Add them to your toolkit if you haven't already.

What Python 3.3 did to improve exception handling in your code

Explore exception handling and other underutilized but still useful Python features.

THIS IS THE FOURTH in a series of articles about features that first appeared in a version of Python 3.x. Python 3.3 was first released in 2012, and even though it has been out for a long time, many of the features it introduced are underused and pretty cool. Here are three of them.

yield from

The `yield` keyword made Python much more powerful. Predictably, everyone started using it to create a whole ecosystem of iterators. The itertools [1] module and the more-itertools [2] PyPI package are just two examples..

Sometimes, a new generator will want to use an existing generator. As a simple (if somewhat contrived) example, imagine you want to enumerate all pairs of natural numbers.

One way to do it is to generate all pairs in the order of `sum of pair, first item of pair`. Implementing this with `yield from` is natural.

The `yield` from `<x>` keyword is short for:

```
for item in x:
    yield item
import itertools

def pairs():
    for n in itertools.count():
        yield from ((i, n-i) for i in range(n+1))
list(itertools.islice(pairs(), 6))
    [(0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0)]
```

Implicit namespace packages

Imagine a fictional company called Parasol that makes a bunch of stuff. Much of its internal software is written in Python. While Parasol has open sourced some of its code, some of it is too proprietary or specialized for open source.

The company uses an internal DevPI [3] server to manage the internal packages. It does not make sense for every Python programmer at Parasol to find an unused name on PyPI, so all the internal packages are called `parasol.<business division>.<project>`. Observing best practices, the developers want the package names to reflect that naming system.

This is important! If the package `parasol.accounting.numeric_tricks` installs a top-level module called `numeric_tricks`, this means nobody who depends on this package will be able to use a PyPI package that is called `numeric_tricks`, no matter how nifty it is.

However, this leaves the developers with a dilemma: Which package owns the `parasol/__init__.py` file? The best solution, starting in Python 3.3, is to make `parasol`, and probably `parasol.accounting`, to be namespace packages [4], which don't have the `__init__.py` file.

Suppressing exception context

Sometimes, an exception in the middle of a recovery from an exception is a problem, and having the context to trace it is useful. However, sometimes it is not: the exception has been handled, and the new situation is a different error condition.

For example, imagine that after failing to look up a key in a dictionary, you want to fail with a `ValueError()` if it cannot be analyzed:

```
import time
```

```
def expensive_analysis(data):
    time.sleep(10)
    if data[0:1] == ">":
        return data[1:]
    return None
```

This function takes a long time, so when you use it, you want to cache the results:

```
cache = {}

def last_letter_analyzed(data):
    try:
        analyzed = cache[data]
    except KeyError:
        analyzed = expensive_analysis(data)
        if analyzed is None:
            raise ValueError("invalid data", data)
        cached[data] = analyzed
    return analyzed[-1]
```

Unfortunately, when there is a cache miss, the traceback looks ugly:

```
last_letter_analyzed("stuff")
    ---------------------------------------------------------------------------


    KeyError                          Traceback (most recent call last)

    <ipython-input-16-a525ae35267b> in last_letter_analyzed(data)
          4     try:
    ----> 5         analyzed = cache[data]
          6     except KeyError:


    KeyError: 'stuff'
```

During handling of the above exception, another exception occurs:

```
    ValueError                        Traceback (most recent call last)

    <ipython-input-17-40dab921f9a9> in <module>
    ----> 1 last_letter_analyzed("stuff")


    <ipython-input-16-a525ae35267b> in last_letter_analyzed(data)
          7         analyzed = expensive_analysis(data)
          8         if analyzed is None:
    ----> 9             raise ValueError("invalid data", data)
         10         cached[data] = analyzed
         11     return analyzed[-1]


    ValueError: ('invalid data', 'stuff')
```

If you use `raise ... from None`, you can get much more readable tracebacks:

```
def last_letter_analyzed(data):
    try:
        analyzed = cache[data]
    except KeyError:
        analyzed = expensive_analysis(data)
        if analyzed is None:
            raise ValueError("invalid data", data) from None
        cached[data] = analyzed
    return analyzed[-1]
last_letter_analyzed("stuff")
    ---------------------------------------------------------------------------


    ValueError                        Traceback (most recent call last)

    <ipython-input-21-40dab921f9a9> in <module>
    ----> 1 last_letter_analyzed("stuff")


    <ipython-input-20-5691e33edfbc> in last_letter_analyzed(data)
          5         analyzed = expensive_analysis(data)
          6         if analyzed is None:
    ----> 7             raise ValueError("invalid data", data)
                          from None
          8         cached[data] = analyzed
          9     return analyzed[-1]


    ValueError: ('invalid data', 'stuff')
```

Welcome to 2012

Although Python 3.3 was released almost a decade ago, many of its features are still cool—and underused. Add them to your toolkit if you haven't already.

Links

[1] https://docs.python.org/3/library/itertools.html
[2] https://more-itertools.readthedocs.io/en/stable/
[3] https://opensource.com/article/18/7/setting-devpi
[4] https://www.python.org/dev/peps/pep-0420/

Looking back at what Python 3.4 did for enum

Plus explore some of the underutilized but still useful Python features.

THIS IS THE FIFTH in a series of articles about features that first appeared in a version of Python 3.x. Python 3.4 was first released in 2014, and even though it has been out for a long time, many of the features it introduced are underused and pretty cool. Here are three of them.

enum

One of my favorite logic puzzles is the self-descriptive Hardest Logic Puzzle Ever [1]. Among other things, it talks about three gods who are called A, B, and C. Their identities are True, False, and Random, in some order. You can ask them questions, but they only answer in the god language, where "da" and "ja" mean "yes" and "no," but you do not know which is which.

If you decide to use Python to solve the puzzle, how would you represent the gods' names and identities and the words in the god language? The traditional answer has been to use strings. However, strings can be misspelled with disastrous consequences.

If, in a critical part of your solution, you compare to the string `jaa` instead of `ja`, you will have an incorrect solution. While the puzzle does not specify what the stakes are, that's probably best avoided.

The `enum` module gives you the ability to define these things in a debuggable yet safe manner:

```
import enum

@enum.unique
class Name(enum.Enum):
    A = enum.auto()
    B = enum.auto()
    C = enum.auto()

@enum.unique
class Identity(enum.Enum):
    RANDOM = enum.auto()
    TRUE = enum.auto()
    FALSE = enum.auto()


@enum.unique
class Language(enum.Enum):
    ja = enum.auto()
    da = enum.auto()
```

One advantage of enums is that in debugging logs or exceptions, the enum is rendered helpfully:

```
name = Name.A
identity = Identity.RANDOM
answer = Language.da
print("I suspect", name, "is", identity, "because they
      answered", answer)
    I suspect Name.A is Identity.RANDOM because they answered
Language.da
```

functools.singledispatch

While developing the "infrastructure" layer of a game, you want to deal with various game objects generically but still allow the objects to customize actions. To make the example easier to explain, assume it's a text-based game. When you use an object, most of the time, it will just print `You are using <x>`. But using a special sword might require a random roll, and it will fail otherwise.

When you acquire an object, it is usually added to the inventory. However, a particularly heavy rock will smash a random object; if that happens, the inventory will lose that object.

One way to approach this is to have methods `use` and `acquire` on objects. More and more of these methods will be added as the game's complexity increases, making game objects unwieldy to write.

Instead, `functools.singledispatch` allows you to add methods retroactively—in a safe and namespace-respecting manner.

You can define classes with no behavior:

```
class Torch:
    name="torch"

class Sword:
    name="sword"

class Rock:
    name="rock"
import functools

@functools.singledispatch
def use(x):
    print("You use", x.name)

@functools.singledispatch
def acquire(x, inventory):
    inventory.add(x)
```

For the torch, those generic implementations are enough:

```
inventory = set()

def deploy(thing):
    acquire(thing, inventory)
    use(thing)
    print("You have", [item.name for item in inventory])

deploy(Torch())
    You use torch
    You have ['torch']
```

However, the sword and the rock need some specialized functionality:

```
import random

@use.register(Sword)
def use_sword(sword):
    print("You try to use", sword.name)
    if random.random() < 0.9:
        print("You succeed")
    else:
        print("You fail")

deploy(sword)
    You try to use sword
    You succeed
    You have ['sword', 'torch']
import random

@acquire.register(Rock)
def acquire_rock(rock, inventory):
    to_remove = random.choice(list(inventory))
    inventory.remove(to_remove)
    inventory.add(rock)

deploy(Rock())
    You use rock
    You have ['sword', 'rock']
```

The rock might have crushed the torch, but your code is much easier to read.

pathlib

The interface to file paths in Python has been "smart-string manipulation" since the beginning of time. Now, with `pathlib`, Python has an object-oriented way to manipulate paths:

```
import pathlib
gitconfig = pathlib.Path.home() / ".gitconfig"
text = gitconfig.read_text().splitlines()
```

Admittedly, using / as an operator to generate path names is a little cutesy, but it ends up being nice in practice. Methods like `.read_text()` allow you to get text out of small files without needing to open and close file handles manually.

This lets you concentrate on the important stuff:

```
for line in text:
    if not line.strip().startswith("name"):
        continue
    print(line.split("=")[1])
     Moshe Zadka
```

Welcome to 2014

Python 3.4 was released about seven years ago, but some of the features that first showed up in this release are cool—and underused. Add them to your toolkit if you haven't already.

Links

[1] https://en.wikipedia.org/wiki/The_Hardest_Logic_Puzzle_Ever

Convenient matrices and other improvements Python 3.5 brought us

Explore some of the underutilized but still useful Python features.

THIS IS THE SIXTH in a series of articles about features that first appeared in a version of Python 3.x. Python 3.5 was first released in 2015, and even though it has been out for a long time, many of the features it introduced are underused and pretty cool. Here are three of them.

The @ operator

The `@` operator is unique in Python in that there are no objects in the standard library that implement it! It was added for use in mathematical packages that have matrices.

Matrices have two concepts of multiplication; point-wise multiplication is done with the `*` operator. But matrix composition (also considered multiplication) needed its own symbol. It is done using `@`.

For example, composing an "eighth-turn" matrix (rotating the axis by 45 degrees) with itself results in a quarter-turn matrix:

```
import numpy

hrt2 = 2**0.5 / 2
eighth_turn = numpy.array([
    [hrt2, hrt2],
    [-hrt2, hrt2]
])
eighth_turn @ eighth_turn
    array([[ 4.26642159e-17,  1.00000000e+00],
           [-1.00000000e+00, -4.26642159e-17]])
```

Floating-point numbers being imprecise, this is harder to see. It is easier to check by subtracting the quarter-turn matrix from the result, summing the squares, and taking the square root.

This is one advantage of the new operator: especially in complex formulas, the code looks more like the underlying math:

```
almost_zero = ((eighth_turn @ eighth_turn) - numpy.array([[0, 1],
                [-1, 0]]))**2
round(numpy.sum(almost_zero) ** 0.5, 10)
    0.0
```

Multiple keyword dictionaries in arguments

Python 3.5 made it possible to call functions with multiple keyword-argument dictionaries. This means multiple sources of defaults can "co-operate" with clearer code.

For example, here is a function with a ridiculous amount of keyword arguments:

```
def show_status(
    *,
    the_good=None,
    the_bad=None,
    the_ugly=None,
    fistful=None,
    dollars=None,
    more=None
):
    if the_good:
        print("Good", the_good)
    if the_bad:
        print("Bad", the_bad)
    if the_ugly:
        print("Ugly", the_ugly)
    if fistful:
        print("Fist", fistful)
    if dollars:
        print("Dollars", dollars)
    if more:
        print("More", more)
```

When you call this function in the application, some arguments are hardcoded:

```
defaults = dict(
    the_good="You dig",
    the_bad="I have to have respect",
    the_ugly="Shoot, don't talk",
)
```

More arguments are read from a configuration file:

```
import json

others = json.loads("""
{
"fistful": "Get three coffins ready",
"dollars": "Remember me?",
"more": "It's a small world"
}
""")
```

You can call the function from both sources together without having to construct an intermediate dictionary:

```
show_status(**defaults, **others)
    Good You dig
    Bad I have to have respect
    Ugly Shoot, don't talk
    Fist Get three coffins ready
    Dollars Remember me?
    More It's a small world
```

os.scandir

The `os.scandir` function is a new way to iterate through directories' contents. It returns a generator that yields rich data about each object. For example, here is a way to print a directory listing with a trailing / at the end of directories:

```
for entry in os.scandir(".git"):
    print(entry.name + ("/" if entry.is_dir() else ""))
    refs/
    HEAD
    logs/
    index
    branches/
    config
    objects/
    description
    COMMIT_EDITMSG
    info/
    hooks/
```

Welcome to 2015

Python 3.5 was released over six years ago, but some of the features that first showed up in this release are cool—and underused. Add them to your toolkit if you haven't already.

Are you using this magic method for filesystems from Python 3.6?

Explore os.fspath and two other underutilized but still useful Python features.

THIS IS THE SEVENTH in a series of articles about features that first appeared in a version of Python 3.x. Python 3.6 was first released in 2016, and even though it has been out for a while, many of the features it introduced are underused and pretty cool. Here are three of them.

Separated numeral constants

Quick, which is bigger, `10000000` or `200000`? Would you be able to answer correctly while scanning through code? Depending on local conventions, in prose writing, you would use 10,000,000 or 10.000.000 for the first number. The trouble is, Python uses commas and periods for other reasons.

Fortunately, since Python 3.6, you can use underscores to separate digits. This works both directly in code and when using the `int()` convertor from strings:

```
import math
math.log(10_000_000) / math.log(10)
    7.0
math.log(int("10_000_000")) / math.log(10)
    7.0
```

Tau is right

What's a 45-degree angle expressed in radians? One correct answer is π/4, but that's a little hard to remember. It's much easier to remember that a 45-degree angle is an eighth of a turn. As the Tau Manifesto [1] explains, 2π, called Τ, is a more natural constant.

In Python 3.6 and later, your math code can use the more intuitive constant:

```
print("Tan of an eighth turn should be 1, got",
      round(math.tan(math.tau/8), 2))
print("Cos of an sixth turn should be 1/2, got",
      round(math.cos(math.tau/6), 2))
print("Sin of a quarter turn should be 1, go",
      round(math.sin(math.tau/4), 2))
    Tan of an eighth turn should be 1, got 1.0
    Cos of an sixth turn should be 1/2, got 0.5
    Sin of a quarter turn should be 1, go 1.0
```

os.fspath

Starting in Python 3.6, there is a magic method that represents "convert to a filesystem path." When given an `str` or `bytes`, it returns the input.

For all types of objects, it looks for an `__fspath__` method and calls it. This allows passing around objects that are "filenames with metadata."

Normal functions like `open()` or `stat` will still be able to use them, as long as `__fspath__` returns the right thing.

For example, here is a function that writes some data into a file and then checks its size. It also logs the file name to standard output for tracing purposes:

```
def write_and_test(filename):
    print("writing into", filename)
    with open(filename, "w") as fpout:
        fpout.write("hello")
    print("size of", filename, "is", os.path.getsize(filename))
```

You can call it the way you would expect, with a string for a filename:

```
write_and_test("plain.txt")
    writing into plain.txt
    size of plain.txt is 5
```

However, it is possible to define a new class that adds information to the string representation of filenames. This allows the logging to be more detailed, without changing the original function:

```
class DocumentedFileName:
    def __init__(self, fname, why):
        self.fname = fname
        self.why = why
    def __fspath__(self):
        return self.fname
    def __repr__(self):
        return f"DocumentedFileName(fname={self.fname!r},
                 why={self.why!r})"
```

Running the function with a `DocumentedFileName` instance as input allows the `open` and `os.getsize` functions to keep working while enhancing the logs:

```
write_and_test(DocumentedFileName("documented.txt",
               "because it's fun"))
    writing into DocumentedFileName(fname='documented.txt',
                                    why="because it's fun")
    size of DocumentedFileName(fname='documented.txt',
                               why="because it's fun") is 5
```

Welcome to 2016

Python 3.6 was released about five years ago, but some of the features that first showed up in this release are cool—and underused. Add them to your toolkit if you haven't already.

Links

[1] https://tauday.com/tau-manifesto

Slice infinite generators with this Python 3.7 feature

Learn more about this and two other underutilized but still useful Python features.

THIS IS THE EIGHTH in a series of articles about features that first appeared in a version of Python 3.x. Python 3.7 [1] was first released in 2018, and even though it has been out for a few years, many of the features it introduced are underused and pretty cool. Here are three of them.

Postponed evaluation of annotations

In Python 3.7, as long as the right `__future__` flags are activated, annotations are not evaluated during runtime:

```
from __future__ import annotations

def another_brick(wall: List[Brick], brick: Brick) -> Education:
    pass
another_brick.__annotations__
    {'wall': 'List[Brick]', 'brick': 'Brick', 'return': 'Education'}
```

This allows recursive types (classes that refer to themselves) and other fun things. However, it means that if you want to do your own type analysis, you need to use ast explictly:

```
import ast
raw_type = another_brick.__annotations__['wall']
[parsed_type] = ast.parse(raw_type).body
subscript = parsed_type.value
f"{subscript.value.id}[{subscript.slice.id}]"
    'List[Brick]'
```

itertools.islice supports __index__

Sequence slices in Python have long accepted all kinds of *int-like objects* (objects that have `__index__()`) as valid slice parts. However, it wasn't until Python 3.7 that `itertools.islice`, the only way in core Python to slice infinite generators, gained this support.

For example, now it is possible to slice infinite generators by `numpy.short`-sized integers:

```
import numpy
short_1 = numpy.short(1)
short_3 = numpy.short(3)
short_1, type(short_1)
    (1, numpy.int16)
import itertools
list(itertools.islice(itertools.count(), short_1, short_3))
    [1, 2]
```

functools.singledispatch() annotation registration

If you thought singledispatch [2] couldn't get any cooler, you were wrong. Now it is possible to register based on annotations:

```
import attr
import math
from functools import singledispatch

@attr.s(auto_attribs=True, frozen=True)
class Circle:
    radius: float

@attr.s(auto_attribs=True, frozen=True)
class Square:
    side: float

@singledispatch
def get_area(shape):
    raise NotImplementedError("cannot calculate area for unknown
                               shape", shape)

@get_area.register
def _get_area_square(shape: Square):
    return shape.side ** 2

@get_area.register
def _get_area_circle(shape: Circle):
    return math.pi * (shape.radius ** 2)

get_area(Circle(1)), get_area(Square(1))
    (3.141592653589793, 1)
```

Welcome to 2017

Python 3.7 was released about four years ago, but some of the features that first showed up in this release are cool—and underused. Add them to your toolkit if you haven't already.

Links

[1] https://opensource.com/downloads/cheat-sheet-python-37-beginners
[2] https://opensource.com/article/19/5/python-singledispatch

Make your API better with this positional trick from Python 3.8

Explore positional-only parameters and two other underutilized but still useful Python features.

THIS IS THE NINTH in a series of articles about features that first appeared in a version of Python 3.x. Python 3.8 was first released in 2019, and two years later, many of its cool new features remain underused. Here are three of them.

importlib.metadata

Entry points [1] are used for various things in Python packages. The most familiar are console_scripts [2] entrypoints, but many plugin systems in Python use them.

Until Python 3.8, the best way to read entry points from Python was to use `pkg_resources`, a somewhat clunky module that is part of `setuptools`.

The new `importlib.metadata` is a built-in module that allows access to the same thing:

```
from importlib import metadata as importlib_metadata

distribution = importlib_metadata.distribution("numpy")
distribution.entry_points
    [EntryPoint(name='f2py', value='numpy.f2py.f2py2e:main',
      group='console_scripts'),
     EntryPoint(name='f2py3', value='numpy.f2py.f2py2e:main',
       group='console_scripts'),
     EntryPoint(name='f2py3.9', value='numpy.f2py.f2py2e:main',
       group='console_scripts')]
```

Entry points are not the only thing `importlib.metadata` permits access to. For debugging, reporting, or (in extreme circumstances) triggering compatibility modes, you can also check the version of dependencies—at runtime!

```
f"{distribution.metadata['name']}=={distribution.version}"
    'numpy==1.20.1'
```

Positional-only parameters

After the wild success of keywords-only arguments at communicating API authors' intentions, another gap was filled: positional-only arguments.

Especially for functions that allow arbitrary keywords (for example, to generate data structures), this means there are fewer constraints on allowed argument names:

```
def some_func(prefix, /, **kwargs):
    print(prefix, kwargs)
some_func("a_prefix", prefix="prefix keyword value")
    a_prefix {'prefix': 'prefix keyword value'}
```

Note that, confusingly, the value of the *variable* `prefix` is distinct from the value of `kwargs["prefix"]`. As in many places, take care to use this feature carefully.

Self-debugging expressions

The `print()` statement (and its equivalent in other languages) has been a favorite for quickly debugging output for over 50 years.

But we have made much progress in print statements like:

```
special_number = 5
print("special_number = %s" % special_number)
    special_number = 5
```

Yet self-documenting f-strings make it even easier to be clear:

```
print(f"{special_number=}")
    special_number=5
```

Adding an `=` to the end of an f-string interpolated section keeps the literal part while adding the value.

This is even more useful when more complicated expressions are inside the section:

```
values = {}
print(f"{values.get('something', 'default')=}")
    values.get('something', 'default')='default'
```

Welcome to 2019

Python 3.8 was released about two years ago, and some of its new features are cool—and underused. Add them to your toolkit if you haven't already.

Links

[1] https://packaging.python.org/specifications/entry-points/
[2] https://python-packaging.readthedocs.io/en/latest/command-line-scripts.html

How Python 3.9 fixed decorators and improved dictionaries

Explore some of the useful features of the recent version of Python.

THIS IS THE TENTH in a series of articles about features that first appeared in a version of Python 3.x. Some of these versions have been out for a while. Python 3.9 was first released in 2020 with cool new features that are still underused. Here are three of them.

Adding dictionaries

Say you have a dictionary with "defaults," and you want to update it with parameters. Before Python 3.9, the best option was to copy the defaults dictionary and then use the `.update()` method.

Python 3.9 introduced the union operator to dictionaries:

```
defaults = dict(who="someone", where="somewhere")
params = dict(where="our town", when="today")
defaults | params
    {'who': 'someone', 'where': 'our town', 'when': 'today'}
```

Note that the order matters. In this case, the where value from `params` overrides the default, as it should.

Removing prefixes

If you have done ad hoc text parsing or cleanup with Python, you will have written code like:

```
def process_pricing_line(line):
    if line.startswith("pricing:"):
        return line[len("pricing:"):]
    return line
process_pricing_line("pricing:20")
    '20'
```

This kind of code is prone to errors. For example, if the string is copied incorrectly to the next line, the price will become `0` instead of `20`, and it will happen silently.

Since Python 3.9, strings have a `.removeprefix()` method:

```
>>> "pricing:20".removeprefix("pricing:")
'20'
```

Arbitrary decorator expressions

Previously, the rules about which expressions are allowed in a decorator were underdocumented and hard to understand. For example, while:

```
@item.thing
def foo():
    pass
```

is valid, and:

```
@item.thing()
def foo():
    pass
```

is valid, the similar:

```
@item().thing
def foo():
    pass
```

produces a syntax error.

Starting in Python 3.9, any expression is valid as a decorator:

```
from unittest import mock

item = mock.MagicMock()

@item().thing
def foo():
    pass
print(item.return_value.thing.call_args[0][0])
    <function foo at 0x7f3733897040>
```

While keeping to simple expressions in the decorator line is still a good idea, it is now a human decision, rather than the Python parser's option.

Welcome to 2020

Python 3.9 was released about one year ago, but some of the features that first showed up in this release are cool—and underused. Add them to your toolkit if you haven't already.