14 Advanced Python Features

2025-04-14 | tech | #python | RSS

Python is one of the most widely adopted programming languages in the world. Yet, because of it's ease and simplicity to just "get *something* working", it's also one of the most underappreciated.

If you search for `Top 10 Advanced Python Tricks` on Google or any other search engine, you'll find tons of blogs or LinkedIn articles going over trivial (but still useful) things like `generators` or `tuples`.

However, as someone who's written Python for the past 12 years, I've come across a lot of really interesting, underrated, unique, or (as some might say) "un-pythonic" tricks to *really* level up what Python can do.

That's why I decided to compile the top 14 of said features alongside examples and additional resources if you want to dive deeper into any of them.

> These tips & tricks were originally featured as part of a 14-day series on X/Twitter between March 1st and March 14th (pi-day, hence why there are 14 topics in the article).
>
> All X/Twitter links will also be accompanied with a Nitter mirror. Nitter is a privacy-abiding open source Twitter frontend. Learn more about the project here.

Table of Contents

1. Typing Overloads

> **Original X/Twitter Thread | Nitter Mirror**

`@overload` is a decorator from Python's `typing` module that lets you define multiple signatures for the same function. Each overload tells the type checker exactly what types to expect when specific parameters are passed in.

> Note that Python's `@overload` works differently than some other languages. In Python, `@overload` tells the type checker to *expect different signatures* for the same function. It doesn't actually define multiple versions of the function itself, unlike Java or C++.

For example, the code below dictates that *only* `list[str]` can be returned if `mode=split`, and *only* `str` can be returned if `mode=upper`. (The `Literal` type also forces mode to be either one of `split` or `upper`)

```python
from typing import Literal, overload

@overload
def transform(data: str, mode: Literal["split"]) -> list[str]:
    ...

@overload
def transform(data: str, mode: Literal["upper"]) -> str:
    ...

def transform(data: str, mode: Literal["split", "upper"]) -> list[str] | str:
    if mode == "split":
        return data.split()
    else:
        return data.upper()

split_words = transform("hello world", "split")  # Return type is list[str]
split_words[0]  # Type checker is happy

upper_words = transform("hello world", "upper")  # Return type is str
upper_words.lower()  # Type checker is happy

upper_words.append("!")  # Cannot access attribute "append" for "str"
```

Overloads can do more than just change return type based on arguments! In another example, we use typing overloads to ensure that either one of `id` OR `username` are passed in, but ***never both***.

```python
@overload
def get_user(id: int = ..., username: None = None) -> User:
    ...

@overload
def get_user(id: None = None, username: str = ...) -> User:
    ...

def get_user(id: int | None = None, username: str | None = None) -> User:
    ...

get_user(id=1)  # Works!
get_user(username="John")  # Works!
get_user(id=1, username="John")  # No overloads for "get_user" match the provided argume
```

> The `...` is a special value often used in overloads to indicate that a parameter is optional, but still requires a value.

✨ **Quick bonus trick:** As you probably saw, Python also has support for **String Literals**. These help assert that only specific string values can be passed to a parameter, giving you even more type safety. Think of them like a lightweight form of Enums!

```
def set_color(color: Literal["red", "blue", "green"]) -> None:
    ...

set_color("red")
set_color("blue")
set_color("green")
set_color("fuchsia")  # Argument of type "Literal['fuchsia']" cannot be assigned to para
```

copy

Additional Resources

- Python Type Hints: How to use `@overload`
- PEP 3124 - Overloading, Generic Functions, Interfaces, and Adaptation
- Python Docs - Overloads
- PEP 586 - Literal Types

2. Keyword-only and Positional-only Arguments

> Original X/Twitter Thread | Nitter Mirror

By default, both required parameters and optional parameters can be assigned with both positional and keyword syntax. However, what if you *don't* want that to happen? **Keyword-only and Positional-only args** let you control that.

```
def foo(a, b, /, c, d, *, e, f):
    #         ^        ^
    # Ever seen these before?
    ...
```

copy

`*` **(asterisk)** marks keyword-only parameters. Arguments *after* `*` *must* be passed as keyword arguments.

```
#   KW+POS | KW ONLY
#       vv | vv
def foo(a, *, b):
    ...

# == ALLOWED ==
foo(a=1, b=2)  # All keyword
foo(1, b=2)  # Half positional, half keyword

# == NOT ALLOWED ==
foo(1, 2)  # Cannot use positional for keyword-only parameter
#      ^
```

copy

`/` **(forward slash)** marks positional-only parameters. Arguments *before* `/` *must* be passed positionally and cannot be used as keyword arguments.

```
# POS ONLY | KW POS
#       vv | vv
def bar(a, /, b):
    ...
```

copy

```
# == ALLOWED ==
bar(1, 2)  # All positional
bar(1, b=2)  # Half positional, half keyword

# == NOT ALLOWED ==
bar(a=1, b=2)  # Cannot use keyword for positional-only parameter
#   ^
```

Keyword-only and Positional-only arguments are especially helpful for API developers to enforce how their arguments may be used and passed in.

Additional Resources

- Using Positional-Only And Keyword-Only Arguments in Python
- Stack Overflow: Why use positional-only parameters in Python 3.8+?
- PEP 3102 – Keyword-Only Arguments
- PEP 570 – Python Positional-Only Parameters

3. Future Annotations

> **Original X/Twitter Thread** | **Nitter Mirror**

A quick history lesson into Python's typing:

> This is less of a "Python Feature" and more of a history lesson into Python's type system, and what `from __future__ import annotations` does if you ever encounter it in production code.

Python's typing system started off as a hack. Function annotation syntax was first introduced with PEP 3107 back in Python 3.0 as purely an extra way to decorate functions with no actual type-checking functionality.

Proper specifications for type annotations were later added in Python 3.5 through PEP 484, but they were designed to be evaluated at bound / definition time. This worked great for simple cases, but it increasingly caused headaches with one type of problem: **forward references**.

This meant that forward references (using a type before it gets defined) required falling back to string literals, making the code less elegant and more error-prone.

copy

```
# This won't work
class Foo:
    def action(self) -> Foo:
        # The `-> Foo` return annotation is evaluated immediately during definition,
        # but the class `Foo` is not yet fully defined at that point,
        # causing a NameError during type checking.
        ...
```

copy

```
# This is the workaround -> Using string types
class Bar:
    def action(self) -> "Bar":
        # Workaround with string literals, but ugly and error-prone
        ...
```

Introduced as a PEP (Python Enhancement Proposal), PEP 563: Postponed Evaluation of Annotations aimed to fix this by changing when type annotations were evaluated. Instead of evaluating annotations at definition time, PEP 563 "*string-ifies*" types behind the scenes and postpones evaluation until they're actually needed, typically during static analysis. This allows for cleaner forward references without explicitly defining string literals and reduces the runtime overhead of type annotations.

```
from __future__ import annotations

class Foo:
    def bar(self) -> Foo:  # Works now!
        ...
```

So what was the problem?

For type checkers, this change is largely transparent. But because PEP 563 implements this by essentially treating all types as strings behind the scenes, anything that relies on accessing return types at runtime (i.e., ORMs, serialization libraries, validators, dependency injectors, etc.) will have compatibility issues with the new setup.

That's why even after ten years after the initial proposal, modern Python (3.13 as of writing this) still relies on the same hacked-together type system introduced in Python 3.5.

```
# ===== Regular Python Typing =====
def foobar() -> int:
    return 1

ret_type = foobar.__annotations__.get("return")
ret_type
# Returns: <class 'int'>
new_int = ret_type()
```

```
# ===== With Postponed Evaluation =====
from __future__ import annotations

def foobar() -> int:
    return 1

ret_type = foobar.__annotations__.get("return")
ret_type
# "int" (str)
new_int = ret_type()  # TypeError: 'str' object is not callable
```

Recently, PEP 649 proposes a new method to handle Python function and class annotations through deferred, or "lazy," evaluation. Instead of evaluating annotations at the time of function or class definition, as is traditionally done, this approach delays their computation until they are actually accessed.

This is achieved by compiling the annotation expressions into a separate function, stored in a special `__annotate__` attribute. When the `__annotations__` attribute is accessed for the first time, this function is invoked to compute and cache the annotations, making them readily available for subsequent accesses.

```
# Example code from the PEP 649 proposal

class function:
    # __annotations__ on a function object is already a
    # "data descriptor" in Python, we're just changing
```

```
    # what it does
    @property
    def __annotations__(self):
        return self.__annotate__()

# ...

def annotate_foo():
    return {'x': int, 'y': MyType, 'return': float}

def foo(x = 3, y = "abc"):
    ...

foo.__annotate__ = annotate_foo

class MyType:
   ...

foo_y_annotation = foo.__annotations__['y']
```

This deferred evaluation strategy addresses issues like forward references and circular dependencies, as annotations are only evaluated when needed. Moreover, it enhances performance by avoiding the immediate computation of annotations that might not be used, and maintains full semantic information, supporting introspection and runtime type-checking tools.

✨ **Bonus Fact:** Since Python 3.11, Python now supports a "Self" type (PEP 673) that allows for proper typing of methods that return instances of their own class, solving this particular example of self-referential return types.

```
from typing import Self

class Foo:
    def bar(self) -> Self:
        ...
```

copy

Additional Resources

- A History Of Annotations
- Python, Type Hints, and Future Annotations
- __future__ — Future Statement Definitions
- PEP 484 – Type Hints
- PEP 563 – Postponed Evaluation of Annotations
- PEP 649 – Deferred Evaluation Of Annotations Using Descriptors
- PEP 749 – Implementing PEP 649

4. Generics

> Original X/Twitter Thread | Nitter Mirror

Did you know that Python has **Generics**? In fact, since Python 3.12, a newer, sleeker, and sexier syntax for Generics was introduced.

```
class KVStore[K: str | int, V]:
    def __init__(self) -> None:
        self.store: dict[K, V] = {}

    def get(self, key: K) -> V:
```

copy

```
        return self.store[key]

    def set(self, key: K, value: V) → None:
        self.store[key] = value

kv = KVStore[str, int]()
kv.set("one", 1)
kv.set("two", 2)
kv.set("three", 3)
```

Python 3.5 initially introduced Generics through the `TypeVar` syntax. However, PEP 695 for Python 3.12 revamped type annotations with native syntax for generics, type aliases, and more.

```
# OLD SYNTAX - Python 3.5 to 3.11
from typing import Generic, TypeVar

UnBounded = TypeVar("UnBounded")
Bounded = TypeVar("Bounded", bound=int)
Constrained = TypeVar("Constrained", int, float)

class Foo(Generic[UnBounded, Bounded, Constrained]):
    def __init__(self, x: UnBounded, y: Bounded, z: Constrained) → None:
        self.x = x
        self.y = y
        self.z = z
```

```
# NEW SYNTAX - Python 3.12+
class Foo[UnBounded, Bounded: int, Constrained: int | float]:
    def __init__(self, x: UnBounded, y: Bounded, z: Constrained) → None:
        self.x = x
        self.y = y
        self.z = z
```

This change also introduces an even more powerful version of **variadic generics**. Meaning you can have an arbitrary number of type parameters for complex data structures and operations.

```
class Tuple[*Ts]:
    def __init__(self, *args: *Ts) → None:
        self.values = args

# Works with any number of types!
pair = Tuple[str, int]("hello", 42)
triple = Tuple[str, int, bool]("world", 100, True)
```

Finally, as part of the 3.12 typing changes, Python also introduced a new concise syntax for type aliases!

```
# OLD SYNTAX - Python 3.5 to 3.9
from typing import NewType
Vector = NewType("Vector", list[float])
```

```
# OLD-ish SYNTAX - Python 3.10 to 3.11
from typing import TypeAlias
Vector: TypeAlias = list[float]
```

```
# NEW SYNTAX - Python 3.12+
```

```
type Vector = list[float]
```

> It's important to note that `NewType` and `TypeAlias` (including the new `type` syntax) are slightly different! NewType creates a subtype that is not equivalent to its base type, whereas TypeAlias assigns an alternative name for an existing type.

```
from typing import TypeAlias

type NewInt = int
AliasInt: TypeAlias = int

NewInt == int  # False, not equivalent
AliasInt == int  # True, equivalent
```

Additional Resources

- Blog on Python 3.12 Generics
- Python 3.12 Preview: Static Typing Improvements
- Python Docs - Generics
- PEP 695 - Type Parameter Syntax

5. Protocols

> Original X/Twitter Thread | Nitter Mirror

One of Python's major features (and also major complaints) is its support for **Duck Typing**. There's a saying that goes:

> **"If it walks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."**

However, that raises the question: *How do you **type** duck typing?*

```
class Duck:
    def quack(self): print('Quack!')

class Person:
    def quack(self): print("I'm quacking!")

class Dog:
    def bark(self): print('Woof!')

def run_quack(obj):
    obj.quack()

run_quack(Duck())  # Works!
run_quack(Person())  # Works!
run_quack(Dog())  # Fails with AttributeError
```

That's where **Protocols** come in. Protocols (also known as **Structural Subtyping**) are typing classes in Python defining the structure or behavior that classes can follow **without** the use of interfaces or inheritance.

```
from typing import Protocol
```

```python
class Quackable(Protocol):
    def quack(self) → None:
        ...  # The ellipsis indicates this is just a method signature

class Duck:
    def quack(self): print('Quack!')

class Dog:
    def bark(self): print('Woof!')

def run_quack(obj: Quackable):
    obj.quack()

run_quack(Duck())  # Works!
run_quack(Dog())  # Fails during TYPE CHECKING (and runtime)
```

In essence, Protocols check what your object ***can*** do, not what it ***is***. They simply state that as long as an object implements certain methods or behaviors, it qualifies, regardless of its actual type or inheritance.

✨ **Additional quick tip:** Add the `@runtime_checkable` decorator if you want `isinstance()` checks to work alongside your Protocols!

```python
@runtime_checkable
class Drawable(Protocol):
    def draw(self) → None:
        ...
```

Additional Resources

- Python Protocols: Leveraging Structural Subtyping
- MyPy: Protocols and structural subtyping
- Python Docs - Protocols
- PEP 544 - Protocols: Structural subtyping

6. Context Managers

> Original X/Twitter Thread | Nitter Mirror

Context Managers are objects that define the methods: `__enter__()` and `__exit__()`. The `__enter__()` method runs when you enter the `with` block, and the `__exit__()` method runs when you leave it (even if an exception occurs).

`Contextlib` simplifies this process by wrapping all that boilerplate code in a single easy-to-use decorator.

```python
# OLD SYNTAX - Traditional OOP-style context manager
class retry:
    def __enter__(self):
        print("Entering Context")

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Exiting Context")
```

```python
# NEW SYNTAX - New contextlib-based context manager
import contextlib
```

```python
@contextlib.contextmanager
def retry():
    print("Entering Context")
    yield
    print("Exiting Context")
```

To create your own, write a function with the `@contextlib.contextmanager` decorator. Add setup code before `yield`, cleanup code after it. Any variables on yield will be passed in as additional context. That's it.

The `yield` statement instructs the context manager to pause your function and lets content within the `with` block run.

```python
import contextlib

@contextlib.contextmanager
def context():
    # Setup code here
    setup()
    yield (...)  # Any variables you want to be passed to the with block
    # Teardown code here
    takedown()
```

The ***real power*** with `contextlib` comes from its exception handling and state management. For example, the same logic with this following function:

```python
@contextlib.contextmanager
def handle_file(path: str):
    try:
        with open(path, 'w') as f:
            f.write('Hello, world!')
            yield f
            f.write('Goodbye, world!')
    except Exception as e:
        raise CustomFileOpenError() from e
```

Using raw `__enter__` and `__exit__` methods would look like this:

```python
class handle_file:
    def __init__(self, path: str):
        self.path = path
        self.file = None

    def __enter__(self):
        try:
            self.file = open(self.path, 'w')
            self.file.write('Hello, world!')
            return self.file
        except Exception as e:
            raise CustomFileOpenError() from e

    def __exit__(self, exc_type, exc_val, exc_tb):
        try:
            # Only write "Goodbye, world!" if no exception occurred during the yield
            if exc_type is None:
                self.file.write('Goodbye, world!')
        finally:
            if self.file:
                self.file.close()
        # Not suppressing any exceptions
        return None
```

Both are functionally identical, but the `@contextmanager` decorator lets us write more readable and maintainable code by handling all the boilerplate for us.

(No need to deal with `exec_type`, `exec_val`, `exec_tb`, etc!)

> ✨ **Bonus Tip:**
>
> `raise ... from ...` is also a neat Python feature that allows you to chain exceptions together!
>
> For example,
>
> ```
> try:
>     raise ValueError()
> except Exception as e:
>     raise NameError() from e
> ```
>
> returns an exception like:
>
> ```
> Traceback (most recent call last):
>   File "...", line ..., in ...
> ValueError
>
> The above exception was the direct cause of the following exception:
>
> Traceback (most recent call last):
>   File "...", line ..., in ...
>   NameError
> ```

Additional Resources

- Context Managers and Python's with Statement
- Python Tips: Context Manager
- Python Docs: `contextlib` — Utilities for with-statement contexts
- PEP 3134 – Exception Chaining and Embedded Tracebacks

7. Structural Pattern Matching

> Original X/Twitter Thread | Nitter Mirror

Introduced in Python 3.10, **Structural Pattern Matching** gives Python developers a powerful alternative to traditional conditional logic. At its most basic, the syntax looks like this:

```
match value:
    case pattern1:
        # code if value matches pattern1
    case pattern2:
        # code if value matches pattern2
    case _:
        # wildcard case (default)
```

The real power comes with **destructuring**! Match patterns break down complex data structures and extract values in a single step.

```
# Destructuring and matching tuples
```

```python
match point:
    case (0, 0):
        return "Origin"
    case (0, y):
        return f"Y-axis at {y}"
    case (x, 0):
        return f"X-axis at {x}"
    case (x, y):
        return f"Point at ({x}, {y})"
```

copy

```python
# Using OR pattern (|) to match multiple patterns
match day:
    case ("Monday"
          | "Tuesday"
          | "Wednesday"
          | "Thursday"
          | "Friday"):
        return "Weekday"
    case "Saturday" | "Sunday":
        return "Weekend"
```

copy

```python
# Guard clauses with inline 'if' statements
match temperature:
    case temp if temp < 0:
        return "Freezing"
    case temp if temp < 20:
        return "Cold"
    case temp if temp < 30:
        return "Warm"
    case _:
        return "Hot"
```

copy

```python
# Capture entire collections using asterisk (*)
match numbers:
    case [f]:
        return f"First: {f}"
    case [f, l]:
        return f"First: {f}, Last: {l}"
    case [f, *m, l]:
        return f"First: {f}, Middle: {', '.join(m)}, Last: {l}"
    case []:
        return "Empty list"
```

You can also combine match-case with other Python features like **walrus operators** to create even more powerful patterns.

copy

```python
# Check if a packet is valid or not
packet: list[int] = [0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07]

match packet:
    case [c1, c2, *data, footer] if (  # Deconstruct packet into header, data, and foote
        (checksum := c1 + c2) == sum(data) and  # Check that the checksum is correct
        len(data) == footer  # Check that the data length is correct
    ):
        print(f"Packet received: {data} (Checksum: {checksum})")
    case [c1, c2, *data]:  # Failure case where structure is correct but checksum is wro
        print(f"Packet received: {data} (Checksum Failed)")
    case [_, *__]:  # Failure case where packet is too short
        print("Invalid packet length")
    case []:  # Failure case where packet is empty
        print("Empty packet")
    case _:  # Failure case where packet is invalid
```

```
        print("Invalid packet")
```

> **NOTE:** This is not the cleanest example, but instead more of a demo to show how you can use match-case with other Python features to create more powerful patterns.

Additional Resources

- Structural Pattern Matching in Python
- Structural pattern matching in Python 3.10
- Good StackOverflow Thread
- Python Docs: The match statement
- PEP 634 – Structural Pattern Matching: Specification
- PEP 636 – Structural Pattern Matching: Tutorial

8. Python Slots

> Original X/Twitter Thread | Nitter Mirror

Slots are a way to potentially speed up the creation and access of any Python class.

TLDR: They define a fixed set of attributes for classes, optimizing and speeding up accesses during runtime.

```
class Person:
    __slots__ = ('name', 'age')

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Under the hood, Python classes store instance attributes in an internal dictionary called `__dict__`, meaning a hash table lookup is required each time you want to access a value.

In contrast, `__slots__` uses an array-like structure where attributes can be looked up in true O(1) time, bringing a minor overall speed bump to Python.

```
# Without __slots__
class FooBar:
    def __init__(self):
        self.a = 1
        self.b = 2
        self.c = 3

f = FooBar()
print(f.__dict__)  # {'a': 1, 'b': 2, 'c': 3}
```

```
# With __slots__
class FooBar:
    __slots__ = ('a', 'b', 'c')

    def __init__(self):
```

```
        self.a = 1
        self.b = 2
        self.c = 3

f = FooBar()
print(f.__dict__)  # AttributeError
print(f.__slots__)  # ('a', 'b', 'c')
```

There is still debate about whether `__slots__` is worth using, as it complicates class definitions with very marginal or no performance benefits at all. However, it is a useful tool to have in your arsenal if you ever need it.

Additional Resources

- Using Slots - Python Wiki
- Don't forget about `__slots__` in Python!
- StackOverflow - Usage of `__slots__`

9. Python Nitpicks

> Original X/Twitter Thread | Nitter Mirror

> This is not a Python "feature" or "tip" per se, but instead a handful of quick syntax tips to really clean up your Python codebase.
>
> As someone who's seen a lot of Python code.

9.1 For-else statements

If you ever need to check if a for loop completes without a break, **for-else statements** are a great way to accomplish this **without** using a temporary variable.

copy

```
# ===== Don't write this =====
found_server = False  # Keep track of whether we found a server
for server in servers:
    if server.check_availability():
        logger.info(f"Found server: {server}")  # Adding some side-effects here
        primary_server = server
        found_server = True  # Set the flag to True
        break
if not found_server:
    logger.info("No server found. Using backup server.")
    # Use the backup server if no server was found
    primary_server = backup_server

# Continue execution with whatever server we found
deploy_application(primary_server)
```

copy

```
# ===== Write this instead =====
for server in servers:
    if server.check_availability():
        logger.info(f"Found server: {server}")  # Adding some side-effects here
        primary_server = server
        break
else:
    # Use the backup server if no server was found
    logger.info("No server found. Using backup server.")
```

```
    primary_server = backup_server

# Continue execution with whatever server we found
deploy_application(primary_server)
```

9.2 Walrus Operator

If you need to define and evaluate a variable all in one expression, the **Walrus Operator** (new in Python 3.8 with PEP 572) is a quick way to accomplish just that.

> Walrus operators are really useful for using a value right after checking if it is `not None`!

copy

```
# ===== Don't write this =====
response = pattern.search(line)
if response:
    print('You pressed:', response)
else:
    print('You pressed nothing')
```

copy

```
# ===== Write this instead =====
if response := pattern.search(line):
    print('You pressed:', response)
else:
    print('You pressed nothing')
```

9.3 Short Circuit Evaluation

Short-circuit Evaluation is a shortcut for getting the "next available" or "next truthy" value in a list of expressions. It turns out you can simply chain `or` statements!

copy

```
# ===== Don't write this =====
username, full_name, first_name = get_user_info()

if username is not None:
    display_name = username
elif full_name is not None:
    display_name = full_name
elif first_name is not None:
    display_name = first_name
else:
    display_name = "Anonymous"
```

copy

```
# ===== Write this instead =====
username, full_name, first_name = get_user_info()

display_name = username or full_name or first_name or "Anonymous"
```

> **NOTE:** As some readers have pointed out, there are some hidden footguns with evaluating variables as booleans!
>
> Empty strings, zeros, and empty lists are all considered "falsy" values, which means you need to be extra careful when checking explicitly for `None` or `0`.

```
# Zero is considered falsy, so it gets skipped even when it's a valid value
user_data = {"name": "Alice", "age": 0}
age = user_data.get("age") or "Unknown"
print(f"User age: {age}")  # "User age: Unknown"
```

9.4 Operator Chaining

Finally, Python lets you **chain comparison operators** together to shorten up integer range comparisons, making them more readable than the equivalent boolean expressions.

```
# ===== Don't write this =====
if 0 < x and x < 10:
    print("x is between 0 and 10")
```

```
# ===== Write this instead =====
if 0 < x < 10:  # Instead of if 0 < x and x < 10
    print("x is between 0 and 10")
```

Additional Resources

- `for/else` - Python Tips
- The Walrus Operator: Python's Assignment Expressions
- PEP 572 – Assignment Expressions
- Using the "or" Boolean Operator in Python
- Chaining Comparison Operators

10. Advanced f-string String Formatting

> **Original X/Twitter Thread** | **Nitter Mirror**

Python's **f-strings** are no secret by now. Introduced in Python 3.6 with PEP 498, they are a better, cleaner, faster, and safer method of interpolating variables, objects, and expressions into strings.

But did you know there is more to f-strings than just inserting variables? There exists a hidden formatting syntax called the **Format Mini-Language** that allows you to have much greater control over string formatting.

```
print(f"{' [ Run Status ] ':=^50}")
print(f"[{time:%H:%M:%S}] Training Run {run_id=} status: {progress:.1%}")
print(f"Summary: {total_samples:,} samples processed")
print(f"Accuracy: {accuracy:.4f} | Loss: {loss:#.3g}")
print(f"Memory: {memory / 1e9:+.2f} GB")
```

Output:

```
========== [ Run Status ] ==========
[11:16:37] Training Run run_id=42 status: 87.4%
Summary: 12,345,678 samples processed
Accuracy: 0.9876 | Loss: 0.0123
Memory: +2.75 GB
```

You can do things like enable debug expressions, apply number formatting (similar to `str.format`), add string padding, format datetime objects, and more! All within f-string format specifiers.

Regular f-strings

```
print(f"Hello {item}!")
```

```
Hello World!
```

Debug Expressions

```
print(f"{name=}, {age=}")
```

```
name='Claude', age=3
```

Number Formatting

```
print(f"Pi: {pi:.2f}")
print(f"Avogadro: {avogadro:.2e}")
print(f"Big Number: {big_num:,}")
print(f"Hex: {num:#0x}")
print(f"Number: {num:09}")
```

```
Pi: 3.14
Avogadro: 6.02e+23
Big Number: 1,000,000
Hex: 0x1a4
Number: 000000420
```

String Padding

```
print(f"Left: |{word:<10}|")
print(f"Right: |{word:>10}|")
print(f"Center: |{word:^10}|")
print(f"Center *: |{word:*^10}|")
```

```
Left: |Python    |
Right: |    Python|
Center: |  Python  |
Center *: |**Python**|
```

Date Formatting

```
print(f"Date: {now:%Y-%m-%d}")
print(f"Time: {now:%H:%M:%S}")
```

```
Date: 2025-03-10
Time: 14:30:59
```

Percentage Formatting

```
print(f"Progress: {progress:.1%}")
```

```
Progress: 75.0%
```

Additional Resources

- Python's F-String for String Interpolation and Formatting
- Python's Format Mini-Language for Tidy Strings
- Python Docs - Input and Output
- PEP 498 - Literal String Interpolation

11. Cache / lru_cache

> Original X/Twitter Thread | Nitter Mirror

You can use the built-in `@cache` decorator to dramatically speed up recursive functions and expensive calculations! (which superseded `@lru_cache` in Python 3.9!)

```
from functools import cache

@cache
def fib(n):
    return n if n < 2 else fib(n-1) + fib(n-2)
```

Since Python 3.2, `@lru_cache` was introduced as part of the `functools` module for quick & clean function memoization. Starting with Python 3.9, `@cache` was added for the same effect with less code. `lru_cache` still exists if you want explicit control of the cache size.

```
FIB_CACHE = {}

# With Manual Caching :(
def fib(n):
    if n in FIB_CACHE:
        return FIB_CACHE[n]
    if n <= 2:
        return 1
    FIB_CACHE[n] = fib(n - 1) + fib(n - 2)
    return FIB_CACHE[n]
```

```
from functools import lru_cache

# Same code with lru_cache :)
@lru_cache(maxsize=None)
def fib(n):
    return n if n < 2 else fib(n-1) + fib(n-2)
```

```
from functools import cache

# Same code with new Python 3.9's cache :D
@cache
```

```
def fib(n):
    return n if n < 2 else fib(n-1) + fib(n-2)
```

Additional Resources

- Python Cache: Two Simple Methods
- (outdated) Caching in Python Using the LRU Cache Strategy
- Python Docs - `@functools.cache`
- Python Docs - `@functools.lru_cache`

12. Python Futures

> **Original X/Twitter Thread** | **Nitter Mirror**

Did you know that Python has native `Promise`-like concurrency control?

```
from concurrent.futures import Future

# Manually create a Future Object
future = Future()

# Set its result whenever you want
future.set_result("Hello from the future!")

# Get the result
print(future.result())  # "Hello from the future!"
```

Python's **`concurrent.futures`** module gives you direct control over async operations, just like JS Promises. For example, they let you attach callbacks that run when the result is ready (just like JS's `.then()`).

```
from concurrent.futures import Future

future = Future()

# Add callbacks BEFORE or AFTER completion!
future.add_done_callback(lambda f: print(f"Got: {f.result()}"))

future.set_result("Async result")
# Prints: "Got: Async result"

future.add_done_callback(lambda f: print(f"After: {f.result()}"))
# Prints: "After: Async result"
```

Python Futures also come with primitives to handle exceptions, set timeouts, or stop tasks completely.

```
from concurrent.futures import Future
import time, threading

# Create and manage a future manually
future = Future()

# Background task function
def background_task():
    time.sleep(2)
    future.set_result("Done!")
```

```
thread = threading.Thread(target=background_task)
thread.daemon = True
thread.start()

# Try all control operations
print(f"Cancelled: {future.cancel()}")  # Likely False if started

try:
    # Wait at most 0.5 seconds
    result = future.result(timeout=0.5)
except TimeoutError:
    print("Timed out!")

# Create failed future
err_future = Future()
err_future.set_exception(ValueError("Failed"))
print(f"Has error: {bool(err_future.exception())}")
```

Just like modern JS, the `asyncio` module has its own Future that works seamlessly with Python's `async/await` syntax:

```
import asyncio

async def main():
    future = asyncio.Future()

    # Set result after delay
    asyncio.create_task(set_after_delay(future))

    # Await just like a JS Promise!
    result = await future
    print(result)  # "Worth the wait!"

async def set_after_delay(future):
    await asyncio.sleep(1)
    future.set_result("Worth the wait!")

asyncio.run(main())
```

Finally, for CPU or I/O bound tasks, Python's `ThreadPoolExecutor` can automatically create and manage futures for you.

```
from concurrent.futures import ThreadPoolExecutor
import time

def slow_task():
    time.sleep(1)
    return "Done!"

with ThreadPoolExecutor() as executor:
    # Returns a Future immediately
    future = executor.submit(slow_task)

    # Do other work while waiting...
    print("Working...")

    # Get result when needed
    print(future.result())
```

Additional Resources

- Introduction to `concurrent.futures` in Python
- Adventures in Python with `concurrent.futures`

- Python Docs - Futures
- Python Docs - `concurrent.futures`

13. Proxy Properties

> Original X/Twitter Thread | Nitter Mirror

Did you know you can make class attributes act as **BOTH** methods **AND** properties?!? *This isn't a built-in feature of Python*, but instead a demonstration of what you can do with clever use of Python's dunder (magic) methods and descriptors.

> (Note that this is very much an example implementation and should not be used in production)

```python
from typing import Callable, Generic, TypeVar, ParamSpec, Self

P = ParamSpec("P")
R = TypeVar("R")
T = TypeVar("T")

class ProxyProperty(Generic[P, R]):
    func: Callable[P, R]
    instance: object

    def __init__(self, func: Callable[P, R]) -> None:
        self.func = func

    def __get__(self, instance: object, _=None) -> Self:
        self.instance = instance
        return self

    def __call__(self, *args: P.args, **kwargs: P.kwargs) -> R:
        return self.func(self.instance, *args, **kwargs)

    def __repr__(self) -> str:
        return self.func(self.instance)

def proxy_property(func: Callable[P, R]) -> ProxyProperty[P, R]:
    return ProxyProperty(func)

class Container:
    @proxy_property
    def value(self, val: int = 5) -> str:
        return f"The value is: {val}"

# Example usage
c = Container()
print(c.value)      # Returns: The value is: 5
print(c.value(7))   # Returns: The value is: 7
```

How does this work under the hood? It comes down to Python's **Descriptor Protocol**:

1. The `__get__` method transforms the `ProxyProperty` object into a **descriptor**.
2. When you access `c.value`, Python calls `__get__` which returns `self` (the descriptor instance).
3. The `__repr__` method handles property access (returning default values).
4. The `__call__` method handles method calls with parameters.

This creates a dual-purpose attribute that can be both read directly AND called like a function!

The benefit of this class is that it allows you to create intuitive APIs where a property might need configuration, or properties that should have sensible defaults but still allow for customization.

> If you want to look at a proper production-ready implementation of **proxy properties**, check out Codegen's implementation of `ProxyProperty` here: codegen/src/codegen/sdk/_proxy.py

Additional Resources

- Python Descriptors: An Introduction
- Demystifying Python's Descriptor Protocol
- Descriptor Guide - Python Wiki
- Proxies and Wrappers

14. Metaclasses

Finally, introducing one of Python's most powerful yet mysterious features: **Metaclasses**

```python
class MyMetaclass(type):
    def __new__(cls, name, bases, namespace):
        # Magic happens here
        return super().__new__(cls, name, bases, namespace)

class MyClass(metaclass=MyMetaclass):
    pass

obj = MyClass()
```

Classes in Python aren't just blueprints for objects. They're objects too! And every object needs a class that created it. So what creates class objects? **Metaclasses**.

By default, Python uses the `type` metaclass to create all classes. For example, these two are equivalent to each other:

```python
# Create a MyClass object
class MyClass:
    ...
obj = MyClass()

# Also creates a MyClass object
obj2 = type("MyClass", (), {})
```

To break down what those arguments mean, here is an example that creates a class with an attribute `x` and a method `say_hi`, that also subclasses off `object`.

```python
# type(
#     name,
#     bases,
#     attributes
```

```
# )
CustomClass = type(
    'CustomClass',
    (object,),
    {'x': 5, 'say_hi': lambda self: 'Hello!'}
)

obj = CustomClass()
print(obj.x)  # 5
print(obj.say_hi())  # Hello!
```

In essence, Metaclasses let you customize and modify these arguments during class creation. For example, here is a metaclass that doubles every integer attribute for a class:

```
class DoubleAttrMeta(type):
    def __new__(cls, name, bases, namespace):
        new_namespace = {}
        for key, val in namespace.items():
            if isinstance(val, int):
                val *= 2
            new_namespace[key] = val
        return super().__new__(cls, name, bases, new_namespace)

class MyClass(metaclass=DoubleAttrMeta):
    x = 5
    y = 10

print(MyClass.x)  # 10
print(MyClass.y)  # 20
```

Here is another example of a metaclass that registers every class created into a registry.

```
# ===== Metaclass Solution =====
class RegisterMeta(type):
    registry = []
    def __new__(mcs, name, bases, attrs):
        cls = super().__new__(mcs, name, bases, attrs)
        mcs.registry.append(cls)
        return cls
```

The problem is, decorators could achieve this same goal without the use of black magic (and it's often cleaner too).

```
# ===== Decorator Solution =====
def register(cls):
    registry.append(cls)
    return cls

@register
class MyClass:
    pass
```

And that kind of brings to light the biggest problem with metaclasses:

Almost 100% of the time, you will never need to touch them.

In your day-to-day development, 99% of your code won't ever hit a use case where metaclasses could be useful. And of that 1%, 95% of those cases could just be solved with regular decorators, dunder methods, or just plain inheritance.

That’s why there is that one famous Python quote that goes:

> ” Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don’t. - Tim Peters

But if you are that 1% which has a unique enough problem that only metaclasses can solve, they are a powerful tool that lets you tinker with the internals of the Python object system.

> **As for some real-world examples of metaclasses:**
>
> - Python’s “ABC” implementation uses metaclasses to implement abstract classes.
> - Python’s “Enum” implementation uses it to create enumeration types.
> - A bunch of 3rd party libraries like Django, SQLAlchemy, Pydantic, and Pytest use metaclasses for a variety of purposes.

Additional Resources

- Python Metaclasses
- What are Python Metaclasses?
- Demystifying Python Metaclasses
- Re: The metaclass saga using Python

Fin

And that’s it folks! 14 of some of the most interesting & underrated Python features that I’ve encountered in my Python career.

If you’ve made it this far, shoot me a quick message as to which ones you’ve seen before and which ones you haven’t! I’d love to hear from you.

Happy Python-ing, y’all !

> *Thank you to all the readers who pointed out and suggested improvements!*
>
> *If you notice any inaccuracies or have any feedback, please let me know!*
>
> *Contact: blog@edward-li.com*