STX NEXT

Python Tech Radar

Acknowledgements

Python Tech Radar is brought to you by STX Next—Europe’s largest software development consulting company specializing in Python.

Python has been our core technology since 2005, and it has helped us support the business growth of 300+ clients worldwide, as well as grow our in-house roster to over 550 experts across all software development disciplines.

Here’s a list of individuals who made Python Tech Radar possible:

Chief Content Officer of Python Tech Radar:
Paulina Kajzer

Additional content support:
Adam Stempniak, Jakub Grajcar

Visual support:
Iga Woytynowska, Igor Krupa

Digital support:
Adam Przewoźny

Promotional support:
Justyna Dados, Aleksandra Wykrota, Natasza Mikołajczak

With deep gratitude to the many experts who contributed to the project:
Ronald Binkofski, Marek Bryling, Szymon Darmofał, Jakub Kołaczkowski, Mikołaj Lewandowski, Marek Melzacki, Jan Pleszyński, Krzysztof Sopyła, Maciej Urbański, Łukasz Wolak

Very special thanks:
Łukasz Langa

Table of Contents

Part I

The Expert Guide

CHAPTER 01

Python's Infinite Adjacent Possibilities

Łukasz Langa

CPython Developer in Residence

@ PYTHON SOFTWARE FOUNDATION

Introduction

There’s a term in theoretical biology coined by Stuart Kauffman called “the adjacent possible.” It explains how life can evolve in a seemingly synchronized manner out of nowhere.

Kauffman believes this is because when all the necessary preconditions for the next evolutionary step are met, that step becomes inevitable. With so much replication happening, biospheres will, on average, take that step sooner or later.

Steven Johnson borrowed the term for his TED Talk, “Where Good Ideas Come From,” which was later turned into a book. In his talk and book, Johnson notes that the history of innovation follows a similar pattern.

Initially, some inventions seem impossible. However, as more discoveries are made, there comes a moment when all the necessary preconditions are met, and the impossible becomes not only possible, but inevitable.

In this opening chapter of Python Tech Radar, I’ll share my perspective as the **CPython Developer in Residence at the Python Software Foundation** to explore how the concept of “the adjacent possible” applies to the evolution of Python and has helped it become the programming language it is today.

What were the necessary preconditions that made Python’s rise in popularity inevitable?

It’s easy to forget that when Guido van Rossum announced Python and published the first source code archive for it on the alt.sources newsgroup, the Linux project hadn’t yet started and there were no web browsers or the web itself. There was no HTML, JavaScript, or Java.

The early history of Python is well documented, so there’s no need to go into detail about it here. Having said that, I’ll just note that it’s remarkable that this lightweight programming language was created before the rules of open source collaboration were established, and yet it still thrives today.

At the time, there were many other dynamic scripting languages in development, such as Larry Wall’s Perl and John Ousterhout’s Tcl, but they have since declined in popularity. So, what kept Python from meeting the same fate?

The most commonly cited reasons for Python’s popularity are its visible language features, such as its “runnable pseudocode” nature, thanks to which the language is easy to learn for beginners yet robust enough for professional use.

What were the necessary preconditions that made Python's rise in popularity inevitable?

Another widely cited reason is its ability to interact well with C code, allowing other programming languages to interoperate with Python.

The scientific community wouldn't have embraced Python to such a degree if not for its ability to call pre-existing scientific code written in C and Fortran. Guido van Rossum's early tagline for the language was that "Python bridges the gap between C and shell programming."

While all of those reasons are indeed accurate, a project can't sustain growth for 30 years based on features alone. Clearly, the community surrounding Python must have done something right.

In my view, the keys to Python's success were twofold. Firstly, Guido's openness to accept contributions from others, listen to their feedback, and sometimes even alter the course of development based on that feedback. All of that was critical for the language to evolve in a way that met the needs of its users.

Secondly, Python embraced modularity early on, enabling the growth of the Python Package Index (PyPI) and later Conda. And finding a library on PyPI that fulfilled your needs was like magic in the early 2000s.

The (very) difficult transition from Python 2 to Python 3

It wasn't all smooth sailing, though. As Python core developers gained experience, it became apparent that some of the early language design choices were flawed. Guido's running joke was that all these issues would be resolved in Python 3000.

Over time, the desire to correct past mistakes gained enough traction among the core developers and support from the community to make the Python team decide to release Python 3.0. It was supposed to be a major, incompatible release that addressed many issues in Python 2 and prepared the language for future developments.

The plan was for users to seamlessly upgrade their Python 2 programs with automatic tooling built into Python 2.7 and Python 3.0, and by 2015, everyone would have transitioned away from Python 2. However, as we all know, that isn't what happened.

The growth of Python at that time was so rapid that many users were unable to upgrade to Python 3 fast enough. The belief that Python 2 programs could be automatically translated into Python 3 proved naive and problematic. A significant amount of user code turned out to be unclear, making it difficult for automatic tools to refactor it without causing issues.

The (very) difficult transition from Python 2 to Python 3

Moreover, maintainers of popular Python libraries on PyPI weren't ready to drop support for Python 2, as all their users were still on that version. Maintaining two separate codebases for both Python 2 and 3 would have been a nightmare.

As a result, many decided to delay their adoption of Python 3, which in turn prevented end users from fully utilizing Python 3, as their preferred libraries weren't available on that version of the interpreter. The situation was getting bad, since a different, more incremental approach was clearly needed.

Overcoming the transitional challenges with polyglot Python

In my opinion, the transition from Python 2 to Python 3 surprisingly helped the language grow. When Python core developers released Python 3.1, they announced there would be no further development of Python 2 and that Python 2.7 would be its last release with only bug fixes being addressed.

This led to a period of perfect stability for library maintainers and end users, which resulted in an increase in the usage of Python between 2008 and 2015, despite the low adoption rate of Python 3. In fact, I'd argue that the low adoption of Python 3 contributed to this growth, as Python 2 was dependable and unchanging.

For us Python core contributors, the situation was dire. Our development efforts weren't receiving much community support, and the future of Python 3 was uncertain.

Fortunately, the community came up with a solution by converting codebases into a polyglot form, meaning they were refactored in a way that allowed them to work on both Python 2 and Python 3 simultaneously.

Helper tools like python-modernize and six gained widespread adoption, and libraries on PyPI gradually received support for Python 3. This allowed end users to gently introduce Python 3 to their projects. If they encountered any difficulties, they could always revert to the reliable Python 2 and try again later.

What's often overlooked is that the requirement to maintain a polyglot codebase made the Python library ecosystem significantly stronger. That's because it forced maintainers to improve their test coverage to ensure their libraries worked on both versions of Python. This was the only way to confirm that the polyglot code was working as intended.

Overcoming the transitional challenges with polyglot Python

The need to run tests with multiple Python versions led to the development of tools such as virtual environments, Tox, advanced CI integrations, and so on—things we strongly depend on even today. Those tools and that infrastructure allowed Python to continue growing steadily.

Python 3 fully replaces Python 2 as the community standard

While the community remained on Python 2, core developers kept adding new innovative features to each subsequent version of Python 3. Finally, with the release of Python 3.6, critical mass was reached, and end users realized that the effort the migration required was worth it to unlock the plethora of Python 3-specific features like Unicode-first text, type annotations, asyncio, and of course f-strings.

In April 2020, Python 2 officially reached its end-of-life. The community was now armed with robust CI systems that could test code on multiple Python versions concurrently. Their test suites were strong and their line coverage was adequate, which allowed Python 3 to continue growing incrementally. This bridged the gap between the two versions and also prevented stagnation.

Exploring the Python 3 landscape and looking ahead

Fast forward to early 2023, and the future of Python looks brighter than ever before. There are over 13 terabytes of package data on PyPI and the computing world is transitioning from the dominance of Intel's x86 architecture.

Our excellent infrastructure tools not only prepared us to build binary packages for the rapidly growing ARM platform, but Python itself also runs exceptionally well on Apple Silicon, with noticeable performance gains over Intel-powered hardware.

Performance has always been an important part of Python, and this trend continues with each new release. Python 3.6 was faster than Python 2.7, and every subsequent version of Python 3 has only widened this gap.

This progress was further accelerated by Microsoft's investment in Python, with its Faster Python team—led by Mark Shannon and including Guido among its members—working to improve the performance of the language.

Their efforts have already paid off tremendously in Python 3.11, which runs the official benchmark suite at speed.python.org 32% faster on average than Python 3.10, with some benchmarks close to twice as fast.

Python 3 fully replaces Python 2 as the community standard

And the best part is that these significant gains are just the tip of the iceberg. Python 3.12 is already showing even more promising results, since work is well under way on the PEP 659—Specializing Adaptive Interpreter. This also provides the foundation for more extensive internal changes within the interpreter, including a long-awaited just-in-time compiler (JIT).

At the same time, other bold initiatives are being taken to tackle the limitations that have hindered Python's adoption in compute-intensive applications. One such limitation is the reliance on the Python GIL (global interpreter lock), whose removal is to be directly addressed in PEP 703.

The adjacent possibilities of Python

Python is now the go-to language for artificial intelligence research and practical application of machine learning and data science. Attempts to rival it have failed so far, since Python's unique set of strengths makes it an invaluable scientific tool.

At the same time, Python is the most popular programming language taught in schools. It also runs in web browsers and on microcontrollers, and can be embedded in mobile applications.

This presents us with a very attractive outlook: when it comes to Python, the adjacent possibilities seem to be infinite. You can develop your idea with Python whether you're interested in artificial intelligence or primary education, web development or mobile applications, accounting software or video games, Docker containers in huge data centers or tiny battery-powered devices.

Best of all, the language is not tied to any for-profit corporation, and its growth path is community-led. Through its open license, acceptance of community changes, its Python Enhancement Proposal (PEP) process, and the annually elected steering council, you can be sure it won't suddenly disappear or shift direction.

Python—the open-source language for everyone

The openness of Python allows you to participate in its development. From minor documentation fixes, through fixing a bug or two in your favorite standard library, to more impactful contributions as a core developer—it's all available to voluntary contributors. In the past 12 months alone, we grew the core engineering team at Python Software Foundation by seven new developers.

Python—the open-source language for everyone

You can now also contribute to Python’s core development through funding. The PSF employs a growing number of individuals working full-time on Python-related projects. As the CPython Developer in Residence, I’m lucky to be one of them. I sincerely hope we can grow this position to more developers, and this is where sponsoring the Python Software Foundation is crucial.

Last but not least, I want to leave you with the same sense of wonder I feel when I look at the big picture. Evolving from humble beginnings as a tool meant to bridge shell scripting and the C language, into its current form as a crucial component in the advancement of science and industry, Python has been and continues to be there for you—free, easy to use, running everywhere, and with infinite adjacent possibilities.

CHAPTER 02

Will Python Ever Become Fully Asynchronous?

Jan Pleszyński
Solutions Architect

Szymon Darmofał
Senior Python Developer

@ STX NEXT

Introduction

Nowadays, modern applications have to support an increasing number of simultaneous operations and users. It's a massive problem for software developers. This is why we have to look for methods that will help us utilize hardware resources effectively and make applications as high-performing as possible.

Let us take you on a journey through concurrent concepts and solutions in the Python ecosystem. In this article, you'll find out about the history of Python concurrency solutions and learn answers to questions such as:

- Why were they superseded by more modern solutions?
- How do these modern solutions apply in the vast Python ecosystem?

On top of that, we'll show you what's behind the corner soon to be introduced. Spoiler alert: the future looks bright!

What is concurrent computing?

To hold a proper discussion about concurrency, we first need to define it. Let's see what a renowned expert has to say on this subject.

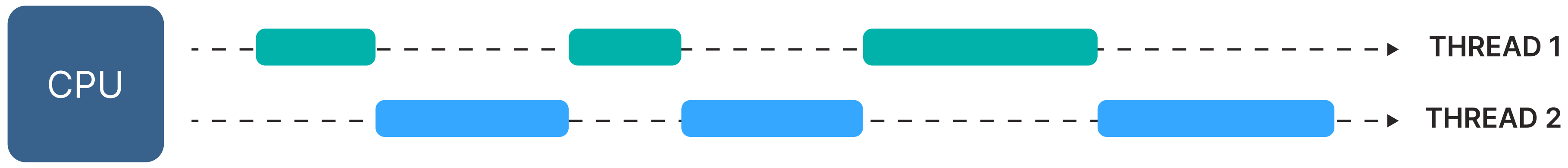
According to Rob Pike:

> *Concurrency is the composition of* ***independently executing computations.***
>
> —Rob Pike

The core feature of concurrency is dealing with a lot of computations at once, but not always at the same time.

Concurrency

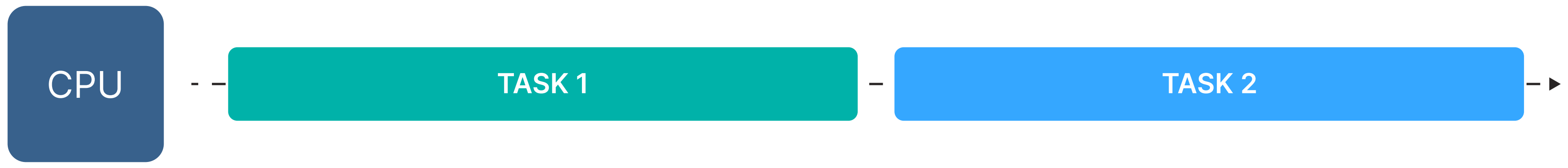
Making progress on more than one task—seemingly at the same time.



What is concurrent computing?

From the perspective of a concurrent system's user, it seems that these concurrent computations are indeed executed at the same time. But this is only an illusion created by the fact that CPU frequency is orders of magnitude greater than the perception of any human being.

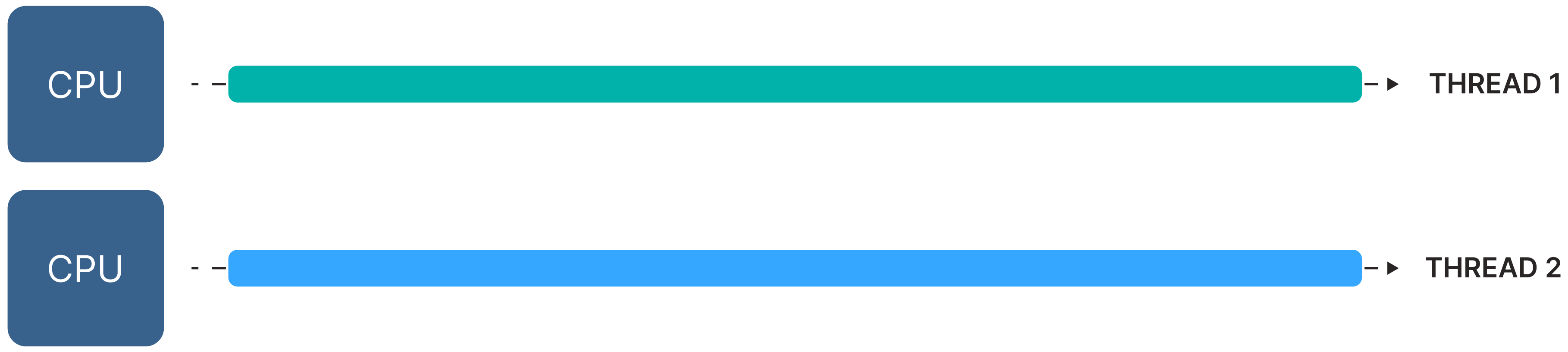
Sequential computing



The opposite of concurrent computing is so-called sequential computing. In this paradigm, all tasks are completed one after the other. So the latter task can't start before the previous one is finished.

Parallel computing

Making progress on more than one task at the exact same time.



A concept closely related to concurrent computing is parallel computing. These two are very often confused. In fact, they're quite similar, but they're not the same. Let us explain how parallel computing differs from concurrent computing.

Parallel computing happens when a task is executed simultaneously on two CPUs. There's no illusion here and computations are indeed executed at the same time.

This chapter won't cover the topic of parallel computation; it's just mentioned here to clear up any confusion between parallel computation and concurrent computation.

However, for the sake of completeness, we'll just give you a small clue in case you didn't know: in Python, parallel computing is done with the use of a *multiprocessing* package.

Why and when is concurrency needed?

Now that we've learned what concurrency is, let's find out why we need it. **The first reason why you may have to consider a concurrent paradigm over a sequential paradigm is to increase performance.** But why and, more importantly, when will you see the performance increase?

This won't happen in every program. The more awaiting for the completion of some other operations, the more a program's performance increases by using a concurrent approach.

The obvious situation where you'll find concurrency indispensable is a server dealing with multiple connections. In most cases, in the scope of the request, there are many I/O calls issued, e.g. to a database or other APIs. I/O operations are blocking and usually take a long time, which may be saved for performing other useful tasks.

You can clearly see that a server dealing with requests sequentially is definitely not an option, as incoming requests would block all requests that come afterward. These types of problems are best solved by using concurrency.

The second reason why you would need concurrency is the fact that some domains can only be modeled by this paradigm. These are domains that are heavily event-driven. In such domains, concurrent tasks will be responsible for responding to these events, without awaiting for other tasks to finish.

You'll find some specific examples of use cases solved by a concurrent approach in the "Use cases" section.

Python's best concurrency solutions from the past

We may confidently say that **from its conception, Python has been a language supporting concurrency.** Throughout history, its developers looked at the problem of concurrency from many different angles, and although not all have stood the test of time, we are thankful for them and you should be, as well.

Do you remember the famous quotation of George Santayana, "Those who can't remember the past are condemned to repeat it"? This is why we want to introduce you to some of these past solutions.

Threading module

From the very first version of Python (Python 1.0), the go-to concurrency tool was the *threading* module. Even though it's still valid and may be of some use, we'll only discuss

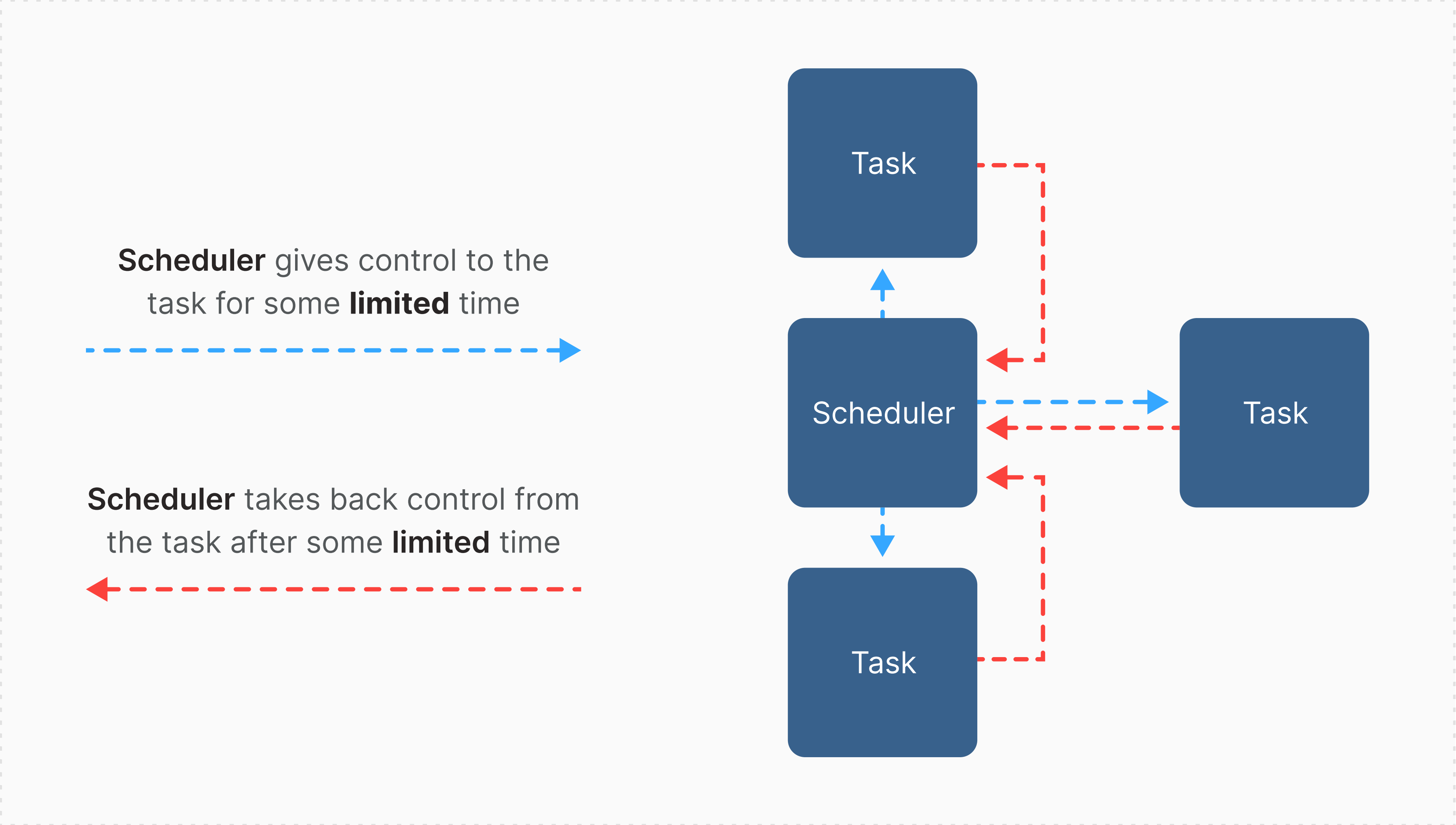
Python's best concurrency solutions from the past

it to show the humble beginnings of Python concurrency, as this won't be your tool of choice anymore.

The *threading* module makes use of underlying system threads. The basic idea here is that many tasks can be performed concurrently in threads, which are managed by the operating system. **Python threads work in a preemptive fashion.** This means that any thread may be paused during any nonatomic statement in order to pass control to the next thread.

They say that a picture (or image) is worth a thousand words, so here's a diagram:

Preemptive model



Scheduler gives control to the task for some **limited** time
Scheduler takes back control from the task after some **limited** time
Task
Scheduler
Task
Task

We can see that **it's the responsibility of the scheduler to preempt a task from what it's currently doing and give control to the next pending task.**

At a glance, this concurrency model is fairly simple to understand and use. It also solves the problem of blocking I/O calls—whenever a blocking I/O call happens in one thread, the rest of them can continue to operate unaffected.

However, threading also has many inherent downsides, making it very difficult to use correctly. These downsides stem directly from the previously mentioned characteristic of threads—preemptiveness.

An example of issues with threads preemptiveness

How can these features cause problems? A simple example will clear any confusion right away:

```python
class TransactionParty:
    def __init__(self, initial_funds: int):
        self._funds = initial_funds

    def _log_deduction(self, funds: int):
         time.sleep(0.01)
         print(f"Deducted {funds}")

    def deduct(self, funds: int):
        if self._funds - funds < 0:
            raise InsufficientFundsError
        self._log_deduction(funds)
        self._funds -= funds

    def add(self, funds: int):
        self._funds += funds

    def __str__(self):
        return f"Current funds: {self._funds}"

def transfer(
    payer: TransactionParty,
    payee: TransactionParty,
    amount: int
):
    payer.deduct(amount)
    payee.add(amount)
```

Here we have a situation where money is transferred between two parties. In the `deduct` method, we have a call to the `_log_deduction` method, which takes some time. Let’s look at the sequential invocation of the code:

```python
def sequential(
    payer: TransactionParty,
    payee: TransactionParty,
    amount: int,
    transactions: int,
):
    for _ in range(transactions):
        transfer(payer, payee, amount)

def main():
    payer = TransactionParty(200)
    payee = TransactionParty(0)

    sequential(payer, payee, amount=100, transactions=3)

    print(f"Payer: {payer}. Payee: {payee}")
```

When the code is called sequentially, it works correctly. Having attempted three times to deduct 100 units of a currency, the deduct method raises `InsufficientFundsError`.

We can imagine that, for some reason, it would be required to run the code concurrently. So the thread counterpart would be:

```
def concurrent_threads(
    payer: TransactionParty,
    payee: TransactionParty,
    amount: int,
    transactions: int,
):
    threads = [
        threading.Thread(
            target=transfer,
            args=(payer, payee, amount)
        )
        for _ in range(transactions)
    ]
    for t in threads:
        t.start()
    for t in threads:
        t.join()


def main():
    payer = TransactionParty(200)
    payee = TransactionParty(0)

    concurrent_threads(payer, payee, amount=100, transactions=3)

    print(f"Payer: {payer}. Payee: {payee}")
```

This time, money is successfully deducted three times leaving the payer with negative funds. So the code that worked correctly once it was run sequentially is now broken—and we’re left with the necessity of debugging the issue. An issue that may be extremely hard to find in a large, convoluted codebase!

What causes the concurrent code to malfunction is the fact that the operations of checking the condition `if self._funds - funds < 0:` and updating the funds `self._funds -= funds` aren’t atomic together.

As we mentioned before, threads work in a preemptive manner, so the context switch between these two statements can happen at any time. We have made the context switch certain by adding a fake `_log_deduction` method, which sleeps for some time.

Nevertheless, with enough load on the server, the context switch would have inevitably happened between checking the condition and updating. This is the exact issue with the `add` function.

Add the fact that all threads operate on the same memory and you have a recipe for disaster, the causes of which aren’t easily traceable.

What are the core issues with preemptive concurrency?

To be honest, the problem presented here isn’t inherent to preemptive concurrency but to concurrency in general. However, old thread-based solutions have major problems compared to future Python concurrency approaches:

- Thread-based solutions don’t make it obvious where the problem may be. Future additions to Python syntax expose places where we can expect this kind of trouble.

- They amplify the problem with concurrent change of a shared state, since context switches can happen at any line of code—even during single nonatomic statements, as in the example of the “+= operator.” So you are forced to be extra careful and think about these minute details when you code.

This is what the creator of the Lua programming language, Roberto Lerusalimschy has to say about this issue:

> ***“Second, and more importantly, we didn’t (and still don’t) believe in the standard multithreading model, which is preemptive concurrency with shared memory: we still think that no one can write correct programs in a language where ‘a = a + 1’ isn’t deterministic.”***
>
> —Roberto Lerusalimschy

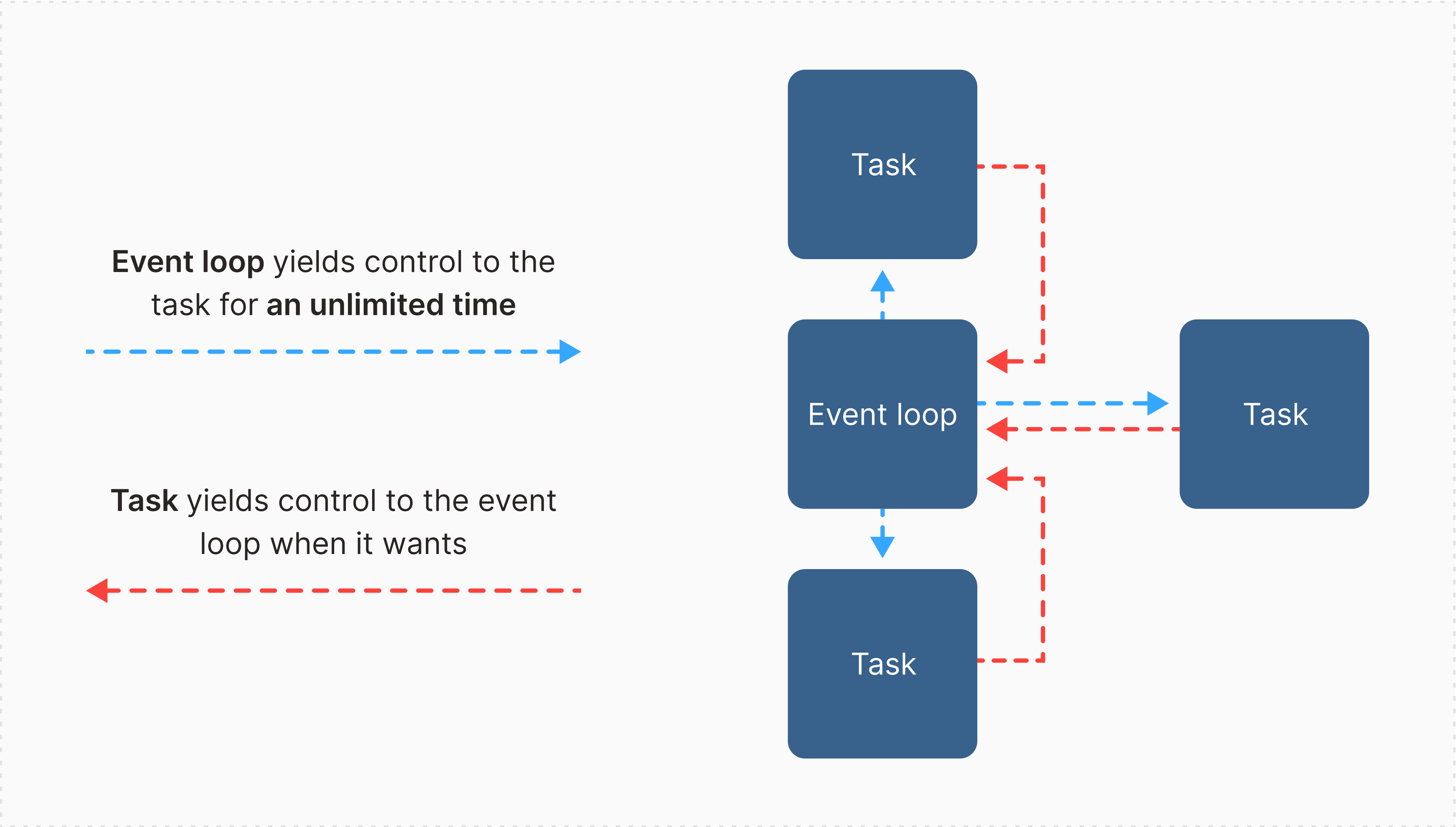
As we can see, threads didn’t happen to be the ultimate solution to solving concurrent challenges in Python. What caused the worst headaches while using threads was their preemptiveness.

We’re sure that if you used Python threads for something more complicated than parallel API calls, you must have been battered by surprising errors caused by preemptive concurrency. We know your pain. It has happened to us many times, as well.
So where did we, as a Python community, head from there?

Learning from past attempts, Python developers noticed that there was a better way to handle concurrent computing: the cooperative concurrency model. How does it differ from the preemptive model offered by a standard threading module? With this approach, concurrent tasks switch only when they decide it's time to switch.

This is completely different from the preemptive approach where, if you recall, tasks could be switched by the underlying OS at any time.

Cooperative concurrency model


Event loop yields control to the task for **an unlimited time**
Task yields control to the event loop when it wants
Task
Event loop
Task
Task

Let's follow the cooperative model diagram. We can see that a special construct called an **event loop** yields control to the task for an **unlimited**—although preferably short—**time**. It's the responsibility of the **developer** to write a task in such a way that it will yield control back when it waits for a result of a blocking operation.

For this model to work well, all tasks have to be non-blocking and yield back control to the event loop frequently. **This behavior is the reason why we call this model "cooperative."**

In recent years, the Python community has developed many frameworks to support cooperative concurrency: Tornado, Twisted, or Eventlet—just to name a few. However, their popularity was never high and there wasn't enough support in the Python syntax for the concept of cooperative concurrency was unwieldy.

asyncio

In 2014, Guido van Rossum, the creator of Python, introduced ***asyncio***—a standard package to Python itself, which was meant to be the go-to solution for solving concurrency problems.

In the early days, asyncio was very similar to the previous cooperative concurrency packages. Following in the footsteps of its predecessors, it used generators and special decorators to yield control of other tasks.

To facilitate the cooperative model better, a whole new Python syntax was introduced soon afterward: *async def* and *await* keywords. That addition was what really kicked off the asynchronous revolution in Python. More on this in the “Asynchronous proliferation” section.

An example of cooperative concurrency

Let’s move on to our transaction example from before, but this time written using the cooperative approach.

```python
class AsyncTransactionParty:
    def __init__(self, initial_funds: int):
        self._funds = initial_funds

    async def _log_deduction(self, funds: int):
        await asyncio.sleep(0.01)
        print(f"Deducted {funds}")

    async def deduct(self, funds: int):
        if self._funds - funds < 0:
            raise InsufficientFundsError
        await self._log_deduction(funds)
        self._funds -= funds

    def add(self, funds: int):
        self._funds += funds

    def __str__(self):
        return f"Current funds: {self._funds}"


async def atransfer(
    payer: AsyncTransactionParty,
    payee: AsyncTransactionParty, amount: int
):
    await payer.deduct(amount)
    payee.add(amount)
```

```
async def asynchronous_transaction(
    payer: AsyncTransactionParty,
    payee: AsyncTransactionParty,
    amount: int,
    transactions: int,
):
    coros = [
        atransfer(payer, payee, amount)
        for _ in range(transactions)
    ]
    await asyncio.gather(*coros)


def asynchronous_main():
    payer = AsyncTransactionParty(200)
    payee = AsyncTransactionParty(0)

    asyncio.run(
        asynchronous_transaction(
            payer,
            payee,
            amount=100,
            transactions=3,
        )
    )

    print(f"Asynchronous transaction. Payer: {payer}. Payee:
{payee}")
```

When run, this code also suffers from the same problem that the thread version had: no exception is raised and the payer ends up with a negative amount of money. **So as you can see, cooperative concurrency isn't a magical solution to all concurrency problems.** However, it's a lot easier to reason about than preemptive concurrency.

What's important this time around is that the source of the problem is highlighted and much easier to debug. It's enough for us to track all the places where code is awaited and after a while, we're pointed to the `deduct` method where there's a race condition between checking the balance and updating it.

Some classes of problems are eliminated, though. For example, in the asyncio version of the code, there's no possibility for a race condition to occur in the `add` function. There's no awaiting there, so we may be certain that the method is safe to call concurrently.

Cooperative vs. preemptive concurrency

Let's summarize this comparison of preemptive and cooperative concurrency by answering the following question: **why is the cooperative approach better than the preemptive one?** There are three main reasons:

1. As the switch happens only in precise moments, the cooperative approach avoids the bugs, race conditions, and other nondeterministic dangers that frequently occur in nontrivial threaded applications.

2. For the race conditions that are unavoidable and inherent to concurrent problems, cooperative concurrency highlights the places where they may happen, therefore making them much easier to debug.

3. The cooperative model offers a simpler programming paradigm to support many thousands of *simultaneous* socket connections, including being able to handle many long-lived connections for newer technologies like WebSockets or MQTT for Internet of Things (IoT) applications.

Asynchronous proliferation

Python's asynchronous world is still growing. Every day, thousands of people contribute to making Python more asynchronous and faster.

The most rapidly growing part of the ecosystem is **web development frameworks.** Some of them are created to be lightweight and super fast, whereas others are all-in-one giants. **If you want to find out which frameworks are the best choices for asynchronous programming, read on!**

Micro web frameworks

We can describe them in two phrases: lightweight and super fast. In most cases, only simple amenities are provided, such as an HTTP client and server with base helpers to process requests and responses.

Don't be misled, though! These frameworks can be used to build very scalable solutions. Why scalable? Because enforced project architecture won't limit you, since it doesn't exist!

It's completely up to you how you structure your codebase and what dependencies you use. This is why more complex frameworks use them as a foundation to build upon.

The most prominent examples of this group are: aiohttp, sanic, and Starlette. They're not very popular, but any of them is a perfect choice for small applications (e.g. microservices).

If you need a super fast HTTP client or server, and developer-friendly utilities don't matter to you, don't hesitate—trust one of these.

High-performance, developer-friendly frameworks

Such frameworks are often built on microframeworks. They're a little slower but bundle more tools for developers. As a result, they speed up the development process and are better for maintenance.

They don't include every possible amenity, though. Still, as a developer, you have to be familiar with the Python ecosystem to build a complex solution using them. You won't find built-ins like ORM, user authorization, etc. in this tier. Having said that, they're prepared to connect with many packages, and there are often ready-to-use third-party solutions to help you out in case of any problems.

Representatives of this group are: FastAPI, Falcon, Molten, tornado, and hug. **If you need a fast, easy-to-develop, maintainable solution with a big community, pick one of those.**

We have mentioned "more tools" above. Okay, but what does it mean? What are those tools? Let's take FastAPI, for example—the most promising framework in this group. What does FastAPI give to developers? Among others:

- Automatically generated API documentation in the OpenAPI and ReDoc formats
- Great editor support, thanks to type hints and docstrings
- Validation of data based on Python type hints
- Great dependency injection
- Ready-to-use middlewares

All-in-one frameworks

In this group, you can find frameworks that provide you with an extensive ecosystem, third-party or built-in (e.g. ORM, user authentication, and authorization modules).

Asynchronous proliferation

The development process using these frameworks is fast, but the final product isn't as scalable as implemented in the previous library types.

Here we have frameworks like Django, Flask, and Pyramid. They're the best choices for MVP applications when you want to have working software fast and you aren't afraid of suboptimal performance.

That being said, the aforementioned frameworks haven't been rewritten to be fully asynchronous yet. This is because they have a vast legacy codebase. It will take some time to implement every module, class, and method in a new way. We can observe that framework authors handle this transformation in two ways: full or step-by-step rewrite.

For example, in Flask and Pyramid, you can use async features only with asgiref `sync_to_async` and `async_to_sync` helper functions (you can read more about it in the "Migration guide" section). Unfortunately, this approach requires a lot of work.

This may be the reason why the creators of Flask decided to reimplement this framework and share it as a new one—Quart, which is recommended for building fully asynchronous Flask-like projects.

Django goes another way. Its contributors have been trying to reimplement every part of the framework to be more asynchronous. The first implementation of asynchronous Django came with version 3.0, which brought asynchronous views. Currently, in version 4.1, we have the ASGI server, asynchronous views, and the first iteration of asynchronous ORM.

However, migration is ongoing, so in many places, developers have to remember about sync-to-async and async-to-sync adapters to make the whole application work correctly (you can read more about it in the "Migration guide" section).

Other tools

Frameworks aren't the only tools that are becoming asynchronous. We can observe migration to asynchronous code in ORMs and database drivers, message queue clients, networking libraries, and testing frameworks. Every new reimplementation of a library makes Python more asynchronous. **We won't have fully asynchronous applications as long as the ecosystem isn't implemented in the new way—the asynchronous way.**

Use cases of asynchronous Python

Python is a good fit for many use cases. Does this statement apply to concurrency, as well? As we've mentioned in the "Why and when is concurrency needed?" section, we need it to improve the performance of an application that involves many I/O operations. **So it makes sense to ask: does your code really need concurrency?**

If you have to process data in batches or pipelines, doing it asynchronously may be a waste of time. The same applies to the command line programs. Why? Because you'll probably want to transform data from one form to another by executing synchronous steps. Your program won't move on without results from the previous step. To sum up, **all operations that use the CPU intensively don't fit in an asynchronous way.**

Where does concurrency fit, then? Whenever you have many operations that can be executed independently. Web applications and APIs are the best examples of it. Does one user have to wait for the answer when another one asks about something different? Definitely not! These operations can be run independently, meaning they can be run asynchronously, too.

The same rule applies to web scrapers and API aggregators. These tools mostly don't have to wait for the previous operation to finish. They can ask the provider for the next object and process it concurrently.

Are there any other use cases? Yes, IoT. Let us describe it using an example.

You want to turn on the lights in the entire house. If you implement it synchronously, you'll send the message to every bulb and wait for the response before you call the next one. What's the effect? The lights turn on one by one. Nice for one room but it can take minutes to illuminate the entire building. The solution is to call all the bulbs asynchronously. You don't need to wait for the responses and your program will work "at the speed of light."

Is that all? Definitely not. There are many cases where asynchronous Python will be a good fit. Not as many as Python itself has, but still a great deal. We're sure that after reading this section, you'll find a whole bunch of new use cases. For now, let's move on to the migration guide.

When would you want to migrate to asynchronous Python? In most cases, this should happen once the performance of your application decreases (because of blocking operations your code performs). You'll quickly find out that the best way to speed up your application is to make it work concurrently.

The idea of a major refactor may be daunting. We've undergone this process and completely understand your fear. Luckily, it's never too late to migrate to asynchronous Python.

Of course, there are many things to migrate. Don't worry, though. We'll break down the required steps for you.

The first rule to keep in mind is that **you can't just call asynchronous code from synchronous code directly** (and otherwise). This is why you shouldn't migrate the entire codebase at once. It's best to perform it in small steps.

The following examples will help you conduct the migration in a safe way. Fasten your seatbelts and let's go!

Synchronous to asynchronous

Firstly, let us show you how to migrate synchronous code to an asynchronous one. **The first step is to add an *async* keyword before the function definition.** Run the code and observe what happens:

```
async def async_function():
   print("Hello!")

def main_function():
   async_function()

main_function() # RuntimeWarning: coroutine 'async_function' was never awaited
async_function()
```

As you can see, the Python interpreter will warn you that the coroutine object isn't awaited. Let us fix this issue:

```
async def async_function():
   print("Hello!")

async def main_function():
   await async_function()
```

```
asyncio.run(main_function()) # Hello!
```

This example may seem trivial, and you're right—it is. What we wanted to show is that once you start introducing async functions, you have to consistently do it all the way to the last function in the call stack.

Just remember, don't apply it blindly everywhere. Do it only when the function at the bottom of the call stack is a blocking operation. Replacing this blocking function with an async counterpart will unlock the full potential of concurrency.

Sometimes you may be faced with the necessity to call synchronous code from an asynchronous context. For example, it may happen in the case of calling a third-party library, which doesn't provide an async interface. Calling such code from an asynchronous function doesn't produce any warning.

What's more, it will freeze your event loop completely. Because of that, all other coroutines will have to wait rather than happily run in a concurrent fashion. You have to always remember that async is turtles all the way down.

Adapters

While migrating the codebase, you may find yourself in a "halfway-through" situation where part of your code is synchronous and the other part is asynchronous. It may be complicated to distinguish which way you have to call a specific function. Moreover, the code will be further migrated. Luckily, you can prepare an adapter to deal with this problem. But first, let's investigate the issue:

```
async def async_function():
  await asyncio.sleep(1)
  print("Asynchronous!")

def sync_function():
   time.sleep(1)
   print("Synchronous!")

async def adapter(func):
   return await func()

await adapter(async_function) # Asynchronous!
await adapter(sync_function) # TypeError: object NoneType can't be used in 'await'
expression
```

As you can see, we're not able to call the synchronous function with the await keyword. How do we deal with it? You can use asyncio `iscoroutinefunction()` to check if a function (or method) has to be called with the `await` keyword:

```
async def adapter(func):
   if asyncio.iscoroutinefunction(func):
       return await func()
   else:
       return await asyncio.get_event_loop().run_in_executor(
           func=func, executor=None
       )
```

The adapter fixed the most annoying migration problem!

```
await adapter(async_function) # Asynchronous!
await adapter(sync_function) # Synchronous!
```

Asgiref helpers

Does the previous example solve all sync-to-async migration problems? Unfortunately, no. Sometimes you can't change the code you are calling. It can be a third-party library or your own code that you can't alter for whatever reason.

What to do then? Check out **asgiref**. This Python library provides helpers to translate communication from sync-to-async and async-to-sync. Let's see how it works in practice:

```
from asgiref.sync import sync_to_async
from third_party_library import external_sync_function

@sync_to_async
def external_sync_function_wrapper():
   external_sync_function()

async def main():
   await external_sync_function_wrapper()
```

Asgiref is delivered by the Django project, but you can use it standalone. A full list of supported or incoming asgiref implementations can be found in this documentation.

You may be wondering right now: is migration really worth the hassle? We have seen that throughout Python's history, best practices for concurrency have changed a few times. Does it mean that in the near future you'll be forced to migrate once again?

The future of asynchronous Python

Have you ever wished you could predict the future? If the answer is "yes," then follow along. We'll reveal the secret of fortune-telling. At least when it comes to Python.

The best place to find information about the nearest future of Python is the Python Enhancement Proposals site (widely known as PEPs).

Over there, you'll find a list of upcoming changes as well as documents describing Python standards, ideology, best practices, etc. It will tell you a lot about Python's future. **Now, let's investigate PEPs together and check out the key info on asynchronous implementation.**

The future is now: Python 3.11

Python 3.11 brings massive changes in concurrency implementation. On the one hand, the old asynchronous implementation is finally defunct. On the other hand, asyncio finally gets on par with its newer peer trio in regard to implemented features.

Let's discuss these changes together.

Asyncore removal

The first one seems unimpressive but is still very important: PEP 594—Removing dead batteries from the standard library. **It's about cleaning up the deprecated asynchronous implementation using asyncore.**

PEP 594 definitely closes the "first async standard library in Python" chapter in the history of Python. The modern async Python uses asyncio as its asynchronous base. What does it mean? Asyncio is finally becoming the only standard library for Python asynchronous programming.

Exception groups

You'll find a revolutionary change for Python concurrency in Python 3.11. It's broadly defined in PEP 654—ExceptionGroups and except*. At last, we can put the nightmare of clumsy exception handling for concurrent operations behind us!

In the past, this feature was the advantage of *curio* and *trio*. In asyncio, it was very problematic to deal with the concurrent tasks exceptions. How did it look before Python 3.11? Let's take a look:

```
async def get_files():
   await asyncio.gather(
          get_file('https://some_external_service/file2312'),
          get_file('https://some_external_service/file3423'),
          get_file('https://some_external_service/file4542'),
     )
```

Here, we are trying to download three files. The important part is calling the `asyncio.gather` method. Because of that, these three `get_file` commands will be executed at the same time—concurrently.

What happens if every function from the example above raises an exception? Unfortunately, only the first exception will bubble up and asyncio will detach from the *gather* function, so you'll lose reference to your tasks (they will run in the background).

Of course, there is a workaround for this problem, but it isn't perfect and requires more time and effort from the developer to catch and process exceptions.

This is where PEP 654 comes in. It delivers the ExceptionGroups, TasksGroups, and except* syntax to improve this particular situation. Thanks to this feature, asyncio will raise an ExceptionGroup that contains all of the exceptions raised instead of one exception. How does it work in practice?

Firstly, we have to make a small change to the code. Instead of using `asyncio.gather`, we will use `TaskGroup`, which we describe further in the "Structured concurrency" section. Now, our example is ready to catch all of the exceptions at once:

```
try:
    async with asyncio.TaskGroup() as task_group:
        task_group.create_task(
            get_file('https://some_external_service/file2312')
        )
        task_group.create_task(
            get_file('https://some_external_service/file3423')
        )
        task_group.create_task(
            get_file('https://some_external_service/file3578')
        )
except* SomeError:
# Do something
except* AnotherError:
# Do something
```

Of course `except*` is optional and backward compatible. You can still catch single exceptions using the good old except. If you don't catch all of the exceptions from the ExceptionGroup, the rest will bubble up and can be caught (or not) in other parts of the code. Quite neat, isn't it?

If you'd like to know more about it, check out Łukasz Langa's webinar at the PowerIT Conference hosted by STX Next.

Structured concurrency

What's more, in Python 3.11, the structured concurrency concept known from trio will be finally available, too. **In short, it's an idea that implies enforcing concurrent tasks to be managed by an outer supervisor.**

In the case of asyncio, this supervising object is called ***TaskGroup*** (just for reference, in trio they called it ***nursery***).

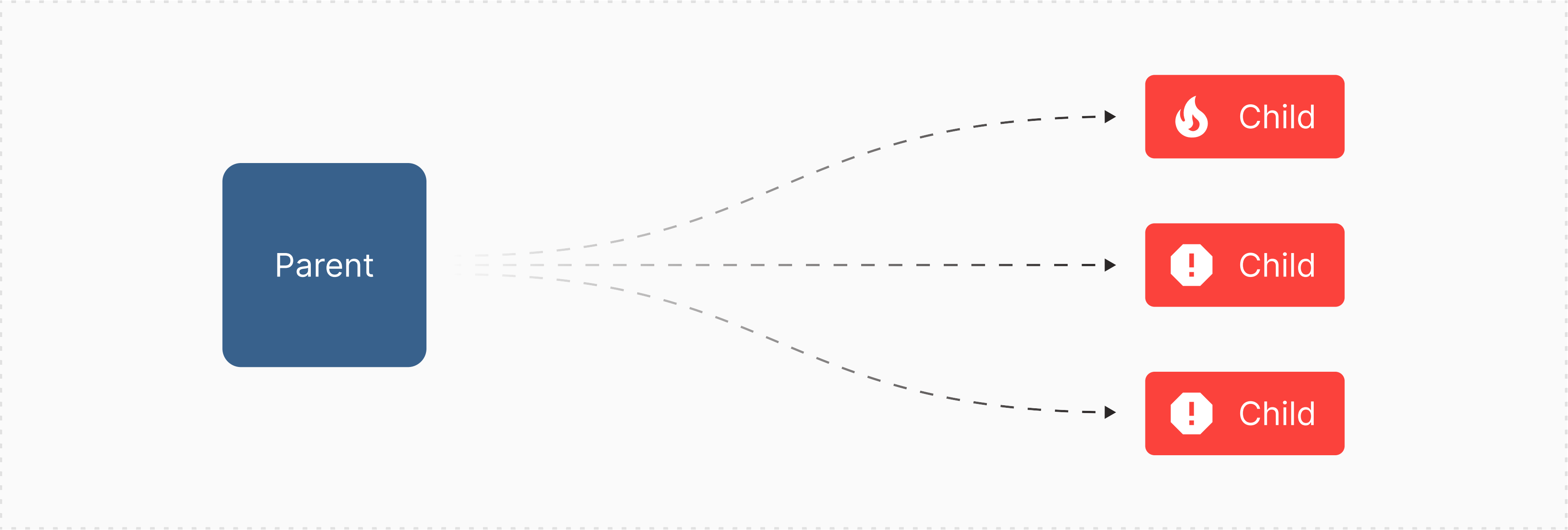
So now, whenever one of the tasks in the TaskGroup fails with an exception other than CancelledError, all other tasks in this TaskGroup are canceled. We'll show you how it works on a simple example:

```
import asyncio


async def task_1():
    await asyncio.sleep(0.1)
    raise RuntimeError
```

```
async def task_2():
    try:
        while True:
            await asyncio.sleep(0.1)
    except asyncio.CancelledError:
        print("Task 2 got canceled")


async def task_3():
    try:
        while True:
            await asyncio.sleep(0.1)
    except asyncio.CancelledError:
        print("Task 3 got canceled")


async def main():
    tasks: list[asyncio.Task] = []
    try:
        async with asyncio.TaskGroup() as task_group:
            tasks.append(task_group.create_task(task_1()))
            tasks.append(task_group.create_task(task_2()))
            tasks.append(task_group.create_task(task_3()))
    except* RuntimeError:
        print("Got RunTime error but that was expected!")

asyncio.run(main())
```

We can see the definition of three tasks, wherein the first one raises `RuntimeError` after 0.1s. Because all three are run from within `task_group`, when `task_1` fails, `task_2` and `task_3` are canceled. Using `asyncio.gather`, `task_2` and `task_3` would have to be canceled manually. This can be easily missed and may be a source of subtle bugs.

Using TaskGroup, we can be certain that all tasks are finished once the context manager scope has exited. This works for all children, which are spawned by TaskGroups created from within the parent. You can imagine it as a tree of tasks that all get canceled whenever one of them fails unexpectedly.



TaskGroup raises ExceptionGroup, so you may handle whichever exceptions you need with new `except*` syntax. However, if the exception raised is `SystemExit` or `KeyboardInterrupt`, then it will be raised as `Exception` rather than `ExceptionGroup`. This is because these two exceptions are a valid way to terminate your program and are very often handled by libraries using *except* statements.

Timeouts

Another great concept that Python 3.11 borrows from trio (trio calls it *cancel_scope*) is called *timeout* in asyncio nomenclature. Let's jump straight into an example, and we'll talk about its intricacies a bit later:

```
import asyncio


async def task_1():
    await asyncio.sleep(10)


async def task_2():
    await asyncio.sleep(7)


async def main():
    try:
        async with asyncio.timeout(2), asyncio.TaskGroup() as
task_group:
            task_group.create_task(task_1())
            task_group.create_task(task_2())
    except asyncio.TimeoutError:
        print("Timeout happened")


asyncio.run(main())
```

Just like the aforementioned TaskGroup, timeouts are implemented as an async context manager. The principle of this operation is simple: nothing in the scope of *timeout* can take longer than the value passed to *timeout*, otherwise the inner tasks will get canceled.

Now, you may be scratching your head, thinking that asyncio already had a mechanism like this. Indeed, we had `asyncio.wait_for`, but it was far less powerful because it canceled only one coroutine, which was directly passed.

Timeout will now obediently cancel all tasks in its scope and all nested tasks that might have been invoked by tasks from the upper layer. (Remember the task tree concept from TaskGroup? It worked exactly the same.)

Additionally, the return value of `asyncio.timeout` can be assigned to a variable and used to programmatically change the deadline of *timeout*. `asyncio.timeout` **is definitely a tool that will simplify the logic of your concurrent applications.**

What changes are on the horizon?

There are some changes that won't be applied in the next Python release, but it's very possible they will come in the near future.

- PEP 568—Generator-sensitivity for Context Variables concerns the problem of context variables when using generators. It isn't a big change for asynchronous Python, but the implementation of asynchronous generators may change because of it.
- A similar change may affect iterators as PEP 533—Deterministic cleanup for iterators mentions. According to the PEP, this change should be integrated as soon as possible. Currently, resource cleanup when using iterators isn't effective.

In some situations, it may release resources after garbage collection, and not when the iterator stops work. There's a workaround for a non-asynchronous code, but it can't be applied to asynchronous generators. That's why it's an urgent case.

Our view of Python's asynchronous future

One thing is certain: Python will be more asynchronous in the future. The largest growth will be visible in web application development. What's more, we're sure that Django contributors will finish their work on making the Django framework fully asynchronous.

This will be a big step and we'll probably soon come to a point where the asynchronous way becomes the standard way for developing web applications and APIs. There are still many things to rebuild in an asynchronous style, but we're sure that it will inevitably happen.

We can even observe this process right now. For example, SQLAlchemy is becoming fully asynchronous. Currently SQLAlchemy's async is a beta feature, but it's just a matter of time.

As you already know, the async style of programming is viral. Once applied in one place of the codebase, it forces parts of the other layers to be migrated, as well. This is why, over time, we're bound to see newly written libraries (including Python standard libraries) being created as asynchronous. Everyone who wants to build an asynchronous application will need them.

This “need” for asynchronous libraries will push their development forward.

Summary and final thoughts on asynchronous Python

Python’s journey in concurrency development has been long and exciting. Finally, we have arrived at the place where we can call Python a very well-suited language for asynchronous development.

Its ecosystem has adapted to the concurrent programming paradigm remarkably quickly, and now we have a plethora of async packages to use, with the new and shiny structured concurrency and syntax for exception groping on top of that. These additions make the implementation of concurrent programs easier than ever.

That in return is what makes Python one of the best languages in the domains of web development, scrapping, IoT, and many others where the asynchronous paradigm is prevalent.

CHAPTER 03

The Business Impact of Python 3.11's Performance Boost

Maciej Urbański

Expert Python Developer

@ STX NEXT

Introduction

Python is a widely used, high-level programming language known for its simplicity and flexibility. However, it isn't necessarily renowned for its speed.

Nevertheless, due to its numerous advantages, it's a popular choice for many different types of projects, including web development, data analysis, machine learning, and scientific computing.

Recently, a new version of Python, Python 3.11, has been released, bringing with it long-awaited performance improvements. This release not only enhances the developer experience by introducing extra language features, but also offers real cost savings for businesses that use Python in their operations.

In this chapter, I'll explore the areas where Python 3.11 will have the greatest impact and discuss how businesses can benefit from upgrading to the new release.

Additionally, I'll share my personal strategy for when and how to upgrade to Python 3.11, drawing on my years of experience and insights gained from discussions with my coworkers.

Python and its complex relationship with performance

When choosing a programming language, there are several things to consider, such as how well it performs in a specific domain, whether it will still be widely used in the next 10 years, and if it provides the necessary performance.

While Python is often praised as *the* language of choice when it comes to quickly prototyping, developing MVPs, and expanding them into full-fledged platforms, one of the main criticisms has always been its performance.

In toy benchmarks, Python falls short when compared to "close to the bare metal" languages like C, or even more advanced languages like Java or C# that also offer garbage collection.

Python's key performance challenges

One of the main challenges with Python's performance is its dynamic nature, which means that many operations can't be optimized at compile time, but rather have to be evaluated at runtime.

Python and its complex relationship with performance

This dynamic nature, which allows for fewer statements required to implement the same business logic, is one of the things that leads to slower execution times, particularly for larger or more complex programs.

Unlike statically typed languages such as C++ and Java, Python is dynamically typed, meaning that variable types are determined at runtime. This flexibility can make Python code easier to write and maintain, but it also results in slower performance compared to statically typed languages.

Another factor that contributes to Python's performance issues is its reliance on garbage collection. To manage memory efficiently, Python uses a garbage collector to automatically free up memory that's no longer being used by the program.

While this means that developers don't need to worry about memory management, it also leads to additional, often unpredictable, delays in program execution and higher resource consumption during execution of the program.

In practice, this means that while developing a product in Python may be cheaper, the infrastructure costs needed to run it may be higher.

Python's solutions to performance issues

Python—or, more specifically, its de facto standard implementation, the CPython interpreter—has made significant progress in improving performance and introducing solutions to these issues since its creation. Having said that, it still often falls short when compared to other languages in terms of execution time, especially those that are compiled, like C or Java.

To counter these issues, the CPython interpreter has a number of features that can help circumvent Python's limitations. For example, a long-standing and effective technique for speeding up Python is... not to use Python, but rather C extensions, which are compiled code exposed as Python modules, bringing C-like speed to the Python world.

This allows for the creation of low-level APIs in a highly performant programming language and high-level logic in an easy-to-write language like Python. The NumPy package is the most famous example of this technique in action. **This is how Python has been able to maintain its popularity in the machine learning world, where high computational throughput is essential.**

Python and its complex relationship with performance

However, if your project is primarily written in Python and you can't use generated computational libraries to represent your business logic, this technique may not be of much use to you.

The trade-off between development and performance costs in Python projects

Even in an age where Moore's Law no longer governs yearly CPU speedups, development cost is often a more important factor than performance cost, especially in new and upcoming projects. Thus, most companies choose Python over more CPU-efficient languages and likely will continue to do so.

Once your product gains traction and you need to support more users, the cost of growing your infrastructure increases and the question of performance becomes more important. This is when Python's slowness becomes more noticeable, and you begin to look for a possible solution.

How big companies use hacks and optimizations to save money when using Python

One way to demonstrate Python's lack of speed is to show how many projects and hacks have been developed to address this very problem.

The most well-known project to make Python faster is an alternative interpreter named PyPy. While aiming to be a drop-in replacement for CPython, PyPy is always a few versions behind. For example, the latest Python release is 3.11, while PyPy currently supports only Python 3.9.

This means that not all new and shiny features of Python are available if you go with that interpreter, and some libraries may not even work at all. Especially C extensions can be troublesome to run under PyPy, which is ironic because a faster interpreter may prevent you from running the most performant Python modules available.

There are many more such initiatives, all with some caveats. To name a few: Numba (JIT for a subset of ScientificPython), mypyc (type-annotated Python-to-C-extension compiler), Nuitka (different compromise similar to mypyc), nogil (CPython fork without the GIL proof-of-concept).

Unfortunately, none of these are a good "default" choice for your next Python project.

How big companies use hacks and optimizations to save money when using Python

Python compatibility, as often defined by, “Does it behave like CPython?” is one of the main factors that prevent the development of a performant Python interpreter or (trans)compiler. Therefore, most of them either focus on a subset of the language or openly state that they aren’t fully compatible.

This is an issue, since Python’s main strengths are its native language features, which enable its expressiveness, as well as its vast library of reusable libraries built over the years and accessible to anyone through PyPI.

A good indicator that both Python and its performance are important in a commercial context is the attention that big companies pay to this problem.

Companies implement hacky solutions to save on infrastructure costs. These hacks are often very specific to a particular workload and can only be achieved after pouring countless hours into debugging and performance testing.

One of the best examples would be Meta (Facebook) who worked hard on their own fork of CPython, called Cinder, which they use to run Instagram. While it was released as a way to give back to the community, it was never intended to be a fully compatible and safe drop-in replacement. Some improvements from it have been merged into the main CPython branch, but others couldn’t be, as they would break backward compatibility.

Another example of business efforts to address the issue was Instagram who famously disabled the garbage collector for their Python web servers in order to improve the performance of their Python code. This move allowed them to reduce their Django-based app server fleet by 10% while handling the same number of users.

That being said, not every company can dedicate this much development time to resource optimization. **The main purpose of choosing Python in the first place is to lower the cost of software development by reducing the amount of time spent by engineers implementing new features**. So what has happened and changed in that area?

Python’s ambitious goals for improvement

CPython developers are very aware of the performance of the Python interpreter and closely monitor it, as it’s easy to introduce a feature that may contribute to overall slowness, which would be counterproductive.

Therefore, CPython is benchmarked every time a new change (big or small) is introduced and the results are published on speed.python.org.

Python's ambitious goals for improvement

The vision of a faster CPython

In 2020, Mark Shannon, one of the CPython core developers, publicly announced a 4-stage plan to make Python 5 times as fast as it was before the release of version 3.10.

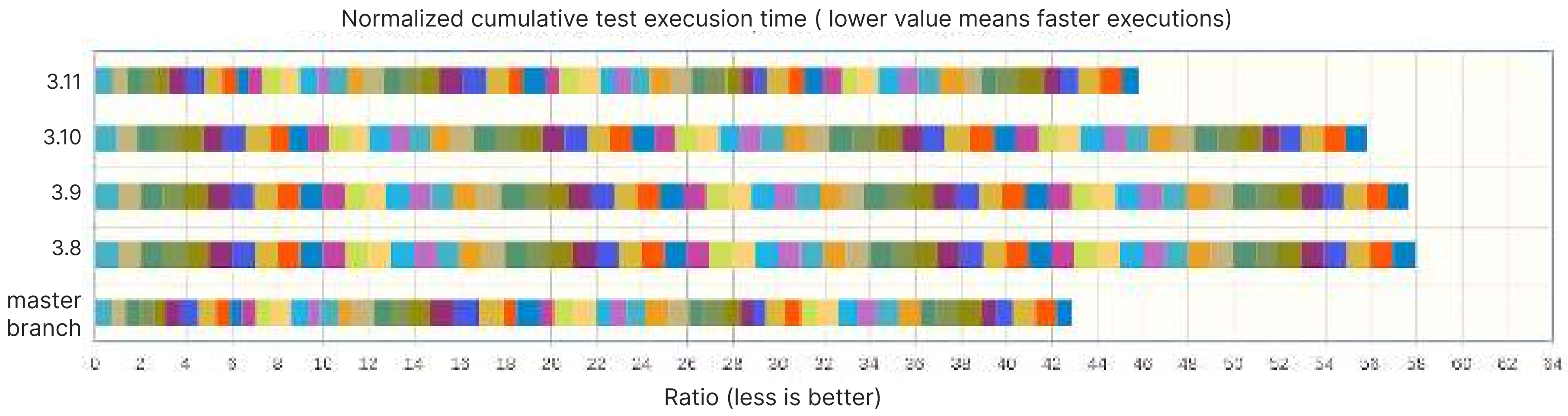
Such a high speedup is almost unprecedented, but the basis of the plan is quite simple. Each stage is designed to deliver approximately 50% speed, thus the cumulative speed after all four stages is expected to be around five times the original speed, or, as the original proposal specified, "1.5 ** 4 ≈ 5."

This isn't an effort that can be done as a volunteer project. Thankfully, as noted before, commercial companies have a vested interest in making Python fast.

To that end, Microsoft has employed Guido van Rossum, the original author of Python and now retired Python leadership figure, along with Mark Shannon and a team of other engineers with the purpose of making this a reality.

The current state of things: Where is Python now in terms of speed?

By visiting speed.python.org, you can already see the results of these efforts:



CPython core developers are constantly striving to make the CPython interpreter even better, with each recent version of Python becoming a little bit faster. However, with the latest Python 3.11 release, things have improved significantly. The future looks similarly optimistic, as the so-called "master" development branch again shows a sizable improvement.

On that note, let's take a closer look at what's available today in Python 3.11.

The reported speedup isn't uniform and highly dependent on the benchmark used, but it's almost always visible and ranges from 10% to 60%.

How CPython developers are making Python faster

This speedup is, of course, only the beginning and not the end goal for "Faster CPython Team" and other CPython core developers. In addition to these efforts, CPython developers are also working on implementing new features and technologies that can improve the performance of Python programs.

For example, they're working on implementing support for Just-In-Time (JIT) compilation, which can greatly improve the performance of certain types of code by compiling it to native machine code at runtime.

For multithreading, which was always painful due to the Python GIL, hope comes in the form of PEP 703, which foresees GIL being "optional" by Python 3.12. This would make threading in Python a viable way to implement concurrency.

While GIL had always been a part of CPython, this problem was largely considered solved. This was done by using concurrency models other than threading for machine learning and other types of computation that need maximum performance only achievable by sharing memory between workers. However, it still was an issue. This PEP represents the first step in the main Python implementation to address it.

The tricks to make Python faster don't end at the interpreter, as language syntax itself can also help Python programs be more efficient, such as the somewhat controversial "walrus" operator (introduced in Python 3.8) or the new "match" statement (3.10).

While it will always be a goal to keep Python syntax neat and readable, we can also expect more of such improvements in the future if they contribute to the goal of making Python a more efficient language.

Overall, the CPython developers are going to use a combination of strategies to make Python faster, including optimizing the interpreter, making changes to the language itself, and implementing new features and technologies.

These efforts are aimed at improving the performance of Python programs and making the language more attractive to developers, business owners, and end users alike.

Expected gains and savings of Python 3.11 in the real world

In this section, I'll take a closer look at the impact Python 3.11 has had on companies worldwide. First, I'll discuss potential performance improvements in your project, then move on to the business benefits of Python's latest release.

How much speedup can I expect in my Python app?

The performance improvements in Python 3.11 can have significant benefits for projects that use Python in their tech stacks. **Unlike more specialized optimizations, CPython 3.11 is between 10–60% faster than the 3.10 release in various benchmarks across the board.**

This means that your app doesn't necessarily need to deal with machine learning or big data to see the improvements. In fact, since these should have already been covered by C extensions, you're more likely to see these improvements in the native Python implementation of your business logic.

As a result, you'll be able to process more with less, reducing the amount of time and resources required for these tasks. This could lead to cost savings in terms of reduced computing power and infrastructure costs, as well as improved efficiency and productivity for your business.

In addition to the potential cost savings, Python 3.11 is also meant to enhance the user experience for developers and users of Python. The new features and improvements in the release are designed to make it easier to write high-performance Python code, making the language more attractive to developers and users alike.

What everyone gets, though, is more responsive applications. Regardless of the cost of running your application, if it's faster, your user will be happier. Now, does that mean you can expect all of your HTTP requests to be processed 10–60% faster?

Again, it depends. The easiest way to check is to just test it out. But if you can't and have to guess, analyze the current bottlenecks of your applications. Is the most time spent in Python code or maybe on I/O like querying a database? Depending on this answer, you'll know how much speedup you can expect.

How do application deployment models influence cost savings from using faster Python?

While switching to Python 3.11 may not automatically reduce infrastructure costs, it can have a significant impact on serverless applications.

Expected gains and savings of Python 3.11 in the real world

When using most serverless cloud offerings, the main part of the bill is directly proportional to execution time (e.g. Google Cloud Functions, Azure Functions, AWS Lambda). **Therefore, if your app runs 10–60%, you can expect your bill to be lowered proportionally to the processing time reduction, excluding any storage and third-party services costs.**

However, most Python projects are not run under a serverless model due to the high startup costs of Python apps (which, by the way, are one of things improved by 10–15% in Python 3.11).

In this case, the savings from faster Python may not be as significant, particularly if the instances (servers) are manually allocated. It doesn't matter much whether your project uses bare-metal hardware, VMs (e.g. AWS EC2), or some cluster services (e.g. AWS ECS). What matters most here is whether your project is in a place where you use some kind of autoscaling solution, which automatically scales up your infrastructure horizontally with the workload.

If so, then you should observe, at best, similar results as with the serverless model. The more server instances you have while using autoscaling, the more noticeable the results will be. If we're talking about fewer than 10 concurrent instances, don't expect to see a difference.

Conversely, if your instances are manually allocated, then there most likely isn't much for you to gain here in terms of infrastructure costs. My assumption here is that if they're manually allocated, then you have fewer than 10 instances, so you can't just "cut" 10% of resources.

You're better off treating the speedup you gain as more room to breathe before you have to scale up again, assuming the success of your project and more users or workload in the future.

Of course, you should monitor your infrastructure and app loads using basic tools like Nagios and Zabbix, or more in-depth Datadog/New Relic-like products, and make decisions based on data gathered through them.

Assessing and managing risks and benefits when upgrading to Python 3.11

To reduce the risks involved in upgrading, it's important to have a continuous integration tool like Github Actions or Jenkins and monitoring solutions like Sentry or Datadog.

Expected gains and savings of Python 3.11 in the real world

These tools will allow you to properly judge the tipping point of the costs of running older versions of Python (i.e. dependencies compatibility and release maturity) versus newer releases (i.e. speed improvements and development convenience features).

This will be especially useful later, as you'll have to make similar decisions with upcoming releases that show significant promise in terms of speedups and backward compatibility.

It's also important to note that the performance gains (10%+) from the interpreter alone may not necessarily translate into a significant difference in your application's performance.

Factors such as I/O bottlenecks, infrastructure costs, and application scaling should also be taken into consideration when assessing the potential impact of upgrading to Python 3.11. Additionally, you shouldn't expect savings in infrastructure if the costs aren't directly proportional to application performance (i.e. you don't use serverless or autoscaling).

Nonetheless, the improved performance of the interpreter will always provide more breathing room for your solutions until you need to look into application performance.

Final thoughts on the business impact of Python 3.11's performance boost

The release of Python 3.11 offers significant performance improvements and cost savings for businesses and organizations that use Python in their operations. These cost savings can translate to larger profit margins.

However, it's important to consider the specific needs and workloads of your project before deciding to upgrade to Python 3.11. How to determine if upgrading is worth it?

Think about the long-term perspective of your project and any dependencies that may not yet be compatible with Python 3.11. If patching these dependencies or waiting for them to become compatible is worth the investment of time and resources, then it may be best to hold off on upgrading.

On the other hand, if your project utilizes a fleet of servers or serverless functions, upgrading to Python 3.11 as soon as possible to take advantage of the cost savings may be the better choice. If that's the case, upgrading is definitely worth it, as you'll be leaving money on the table otherwise. However, if not, being patient might make more sense.

Final thoughts on the business impact of Python 3.11's performance boost

Ultimately, it's important to weigh the potential benefits and costs of upgrading and make a decision based on the specific needs of your project.

If you don't feel ready to adopt the new release just yet, my advice would be to add it to your issue tracker and let your developers pick Python 3.11 migration tasks whenever it's most convenient for them. They're the people best equipped to estimate the cost of such a migration. Having that ticket set up will also serve as a reminder for you that the migration should happen at some point.

CHAPTER 04

Maximizing Value with Python: MVP Development in the Software Industry

Łukasz Wolak
Senior Python Developer

Marek Melzacki
Senior Python Developer

@ STX NEXT

Introduction

In today's ever-growing software industry, most businesses are in need of reliable, highly functional applications to support their in-house procedures.

Companies that invest in software want to see results fast to ensure they're putting their money to good use. An idea is just an idea until it's put into practice. You can study the target group and estimate potential profits, but these are just speculations until your product is built and your clients decide if it's a worthwhile investment.

As developers, we prove our value by delivering high-quality code, though without bringing business value to your app, quality doesn't mean much. To increase our chances of providing you with the best services, we should focus on technical skills while also improving our understanding of your business and its context.

A bright future awaits developers who invest their time in developing communication skills, learning about business problems, and writing the best code. Being Python developers, we have the advantage of leveraging the various possibilities and solutions that Python offers.

As of 2023, Python continues to be one of the most popular programming languages in the world, with a gigantic community and a multitude of packages and libraries. It can be used for writing automation scripts, web development, data processing, machine learning, and many more.

We'll guide you through the complex process of developing an MVP in Python and introduce the business cases you may find in your projects. We'll begin with the definition of an MVP (Minimum Viable Product), ways to think about it, and questions to ask yourself before starting. Next, we'll provide a technical overview of MVP development with examples of case studies to show you the approaches we've taken, so you can adapt them to your needs.

What is an MVP?

These days, everybody's building or planning an MVP, so we're sure you've already heard that term multiple times. But what does it actually mean?

An MVP is a product, or, to be more precise, a version of a new product with just enough features to be usable and verifiable by users or potential customers. The goal here's to launch your product as quickly as possible and collect the maximum amount of feedback with the least effort on the team's side. After receiving enough feedback, you'll know if your product or idea has any real potential.

What is an MVP?

An ideal MVP should be:

- built fast,
- of high quality,
- cheap to develop.

Realistically speaking, it's an accomplishment to achieve even two of these characteristics. We can build a good product quickly, but it won't be cheap. We can also build something cheap and fast, but the quality may be questionable. Finally, we can try building a quality product at a low expense, but time may not be on our side.

As it's almost impossible to achieve all three criteria, it's important that you strategize and determine the optimal course of action. So, how can we make it happen?

Why build an MVP? What do you gain from it?

The purpose of an MVP is to test if your idea or product will work as expected and if your clients will like it. Your team will want to gather feedback from them as quickly as possible and ensure that your product helps you achieve your goals.

It's more effective to collect feedback from actual users of the product rather than relying on hypothetical responses from surveys. Your picture of the product might greatly differ from how it's perceived by others.

Even showing mockups and describing how your app works doesn't provide a true sense of the product for people outside of the project. That's why building a working version of the product for them to try out is the best way to gauge its effectiveness in solving their problems.

Why should developers think about the business context?

At this stage, we're not discussing technicalities. Our focus is solely on the business aspects because that's what an MVP is for. But you might be wondering, should software developers even care about these things? Why are we talking about business instead of delving into technical aspects?

Archimedes once said, "Give me a lever long enough and a fulcrum on which to place it, and I shall move the world." In our opinion, if you want to grow as a developer and become an expert in your field, you need to understand things that go beyond technical knowledge. Building an MVP is one of those things.

Why should developers think about the business context?

As developers, we have the skills and capabilities to bring revolutionary ideas to life. However, the crucial thing is knowing where to start and where to go. We should all work on big ideas that have the potential to benefit our projects and businesses. To achieve this, we must leave mediocrity behind and focus on things that truly matter.

To illustrate this point, let's consider an example of how to identify profitable features and those that will only drain energy and resources from the project.

Sample MVP idea in practice

Imagine you're building a platform that allows users to exchange items with one another. Users can store all their tradeable items in the system, and the platform generates revenue whenever users exchange goods. To make this process as easy and straightforward as possible, we want to provide users with multiple communication options.

The main point is that users need to be able to perform transactions. To facilitate this, the platform supports three communication options:

- direct messages within the app,
- comments under the postings of the exchangeable objects,
- a new feature exclusive to the app: audio/video calls.

The creator of the idea assumed that implementing audio/video calls would revolutionize communication and make the negotiation process faster and easier, ultimately leading to increased revenue.

Having said that, before implementing functionalities like audio/video calls, we need to consider the potential impact. We can roughly estimate which communication feature will be more costly to develop, but the client may still insist on implementing that particular feature.

It's important to recognize that not all users will utilize this feature. If only a small number of users end up using it after we've invested significant time and effort into implementing it, then it won't have helped us achieve our primary objective of making it easier and faster for users to perform transactions.

Therefore, we suggest creating a button like, "I want to use the audio/video feature." This will provide us with the necessary data to decide what percentage of users might want to use it. If enough people click the button, then it's worth expanding the feature.

Why should developers think about the business context?

This approach allows us to make informed decisions about which functionalities to prioritize while keeping the users' needs and objectives in mind.

What do business experiments mean for software developers?

Simply put, "I want this feature" is a business experiment. The goal's to gain knowledge about the users and verify feature ideas.

Developers can benefit greatly from understanding user behaviors and tendencies. By monitoring the system or conducting A/B tests, we can determine which parts of the app are being used more often.

These areas are likely to be expanded in the future, so it's essential to make the code base there easier to develop. Additionally, we may need to optimize the code to ensure that it performs well and consumes fewer resources.

It's important to avoid implementing code that may not be useful at all. To do this, we need to understand the business side of the project and the implications that a given feature will have.

When in doubt, don't be afraid to ask questions! Over time, as we become more familiar with the project and the needs of the users, our intuition will improve and guide us through each step of the development process.

What do you gain from developing an MVP?

Building an MVP allows you and your team to verify:

- if you're going in the right direction,
- if your users are using the application as intended,
- if your product is meeting customer needs,
- if the way your team is solving problems makes sense,
- if there are any other issues that your users might find necessary to implement,
- if it's worth starting to fully develop your product now.

That being said, those benefits aren't limited only to the business side of the coin. They also have implications for the software development process.

Why should developers think about the business context?

Let's go back to the situation where only a very small number of users utilize a feature that took a lot of effort to implement, such as the audio/video communication feature. In that scenario, we need to investigate what went wrong. Some possible reasons might be that:

- the users weren't aware of the feature,
- the feature was hard to find,
- the feature wasn't marketed effectively,
- the users simply didn't like it and preferred other communication options.

Therefore, getting user feedback as soon as possible is crucial. As developers, we should focus on building features that matter to users and bring value to your product. We should cut unnecessary expenses on features that don't work and allocate the resources where the return on investment will be better.

MVP vs. proof-of-concept vs. prototype

Many people use the terms MVP, proof-of-concept, and prototype interchangeably, but they're actually different things. Expert-level developers who want to understand the business context of programming should be able to use them correctly. Here are some quick definitions:

MVP

As we mentioned earlier, an MVP is a fully functional product that's ready for deployment. Although it may have only a few features, it's usable by users outside of the team or company. An MVP solves a problem or provides the functionality it's supposed to verify. It also handles all possible situations that the user might find themselves in and all their interactions with the system.

Proof-of-concept

A proof-of-concept (POC) is used to verify whether something is possible in our project, technology stack, and so on. POCs are mostly used as internal technical projects.

For instance, suppose we want to incorporate an external audio/video call provider into our app. To that end, we would integrate some library or package with our system or a system with a similar structure and tech stack to determine if it's even possible or how easy it would be. In that case, we would mostly be doing coding work and configuration.

The end result of the POC would be a working integration with this specific service provider. However, it's also crucial to define what we want to prove in the POC and what's unimportant in this process.

All we want to prove here's that it's possible to make an audio or video call using our app. As a result, the proof of concept shouldn't focus on the frontend side or user experience in general.

Most of the unhappy paths and potential problems the user might encounter while attempting to connect will be ignored. Errors, timeouts, connection issues, and other things will be addressed later when we decide to implement it fully. For now, we'd only demonstrate that it can be done. The next step is to bulletproof the concept.

Prototype

If we agree that the POC is mostly oriented toward code and technicalities while largely ignoring the UX, then the prototype will be its exact opposite. Prototypes are primarily clickable mockups that show how the process or functionality will be implemented from the user's perspective. They are low-code or no-code solutions that help stakeholders understand the concept or design.

Figma, Zeplin, and Adobe XD are excellent tools for creating prototypes, so it's easy to imagine that they will mostly focus on design and UX: clickable elements take you from one page to another, and the prototype simulates how the system will work and what will happen every step of the way.

What do we gain from understanding these differences?

It's essential to know the definitions of those terms so that you understand the state of the project your team's supposed to build. Your individual developers may be using different names than you for different things.

You might want to create a POC, but your developers could be talking about a full-fledged MVP. This could lead to incorrect estimations and problems later on. It's in your best interest to find out what you're working with.

Please be aware of these differences, as we'll be discussing developing a full MVP in the following sections.

Approaches to different MVP types and development process

Once an MVP has served its purpose, there are two main approaches you can take: you can either develop it further with the same technology stack and tools, or discard the MVP and leverage the knowledge gained to create a new product with new tools and technologies.

Let's take a closer look at both options and explore their pros and cons.

1. Discarding an MVP

Although some may consider this a radical idea, discarding an MVP can actually be very time-efficient compared to the traditional software development process.

Once you know that the idea has caught your interest, your team can start full development with the knowledge they didn't have before building the MVP. With this approach, it's important to remember that the product may evolve into something unexpected later on, which may cause some problems.

2. Developing the MVP further

The second approach means that once you've created an MVP and obtained feedback, it's time to move on and focus on developing a full-fledged product.

The biggest difference between the first and the second approach is the short- and long-term commitment to the code base. Here, you need to consider whether the code base and product are easily extendable, modifiable, scalable, and maintainable. These are critical factors to take into consideration when creating an MVP that won't be discarded once it serves its purpose.

Starting from scratch may not always be cost-efficient, so it's crucial to make the right decisions when it comes to technology and architecture. Here are some questions to consider before delving deeper into developing the MVP further:

- "Does my team have the necessary skills to further develop the product?"
- "How will my project scale up? Can I scale up only one part of my project?"
- "Can I later migrate it to a different server or cloud?"
- "Will the technologies, libraries, and frameworks I chose have long-term support?"
- "How long do I think the lifecycle of the project will be?"
- "Does my company have any existing infrastructure?"

Approaches to different MVP types and development process

- “How will I migrate the data if the databases change?”
- “Should the whole project be developed using the same technology?”
- “What type of developers should my team consist of? Backend developers, frontend developers, data scientists, data engineers, or machine learning engineers?”

It’s important to keep in mind that there will always be tradeoffs between short-term wins with poor fundamentals in the long-term and slower development with a solid foundation for the future. That’s why we’ll explore different approaches and help you determine the best options for you and your project in the next sections.

What are the tools and frameworks for building an MVP?

Python offers a comprehensive set of tools and frameworks for building MVPs in the software industry. In this section, we’ll explore those tools and frameworks, highlighting their key features and how they can be used in the development of minimum viable products.

Our focus will be on comparing and evaluating those tools and frameworks from the perspective of maximizing value, so you can choose the best solution for your MVP project.

Frameworks for building an MVP

Whether you choose to discard or continue developing your MVP, several frameworks and libraries for web development can be used. The most popular ones are Django, Flask, and FastAPI.

Django is a good choice for building full-stack apps and APIs, while Flask is great for anything that might be outside the typical CRUD logic. FastAPI is similar to the Flask approach, but has additional tools and libraries and has been gaining in popularity over time.

In addition to these frameworks, there are several libraries, packages, and tools that can be integrated into Python projects in the fields of data engineering and machine learning. NumPy, Pandas, and TensorFlow are some examples of those, and they provide a rich ecosystem for building data-driven MVPs that can help you create powerful and valuable products.

What are the tools and frameworks for building an MVP?

When cost is one of the major factors and you choose the discard-MVP option, Django is the most common framework used to build the solution. This framework comes with "batteries included," offering developers the ease of rapid development of features, data models, and other necessary functionalities.

Django is well-suited for situations where the primary focus is on building an application that works around data input, and custom business logic is a secondary consideration. This framework offers robust and comprehensive capabilities for data management and is well-suited for building MVPs that require data-centric functionality.

Deploying your application: strategies and considerations

Deploying your application is a crucial step in the software development process, as it allows users to access your solution. The choice of deployment flow and platform depends on various factors, such as the expected usage patterns and the resources available.

For example, if you want to verify the success of your MVP quickly, a Platform as a Service (PaaS) solution like Heroku can be a simple and effective option, as it provides a hosted platform with built-in components like the database.

Alternatively, if you have access to building blocks from a cloud provider, a deployment platform from that provider may be a better choice, since it can offer more flexibility and control.

For instance, deploying a containerized application to a Kubernetes engine using community-defined Helm charts can simplify cluster management, and using a DBaaS solution can provide a scalable and reliable database engine.

While this approach may be less cost-efficient for running the application 24/7, the "pay as you go" approach for billing the infrastructure used in the early stages of presenting the application to end users could be better for you.

Architecture approaches for building an MVP

The architecture of the code that we use to build an MVP is critical to the success of the product, and it largely depends on the people involved and what's most suitable for the current project. **When it comes to Python and MVPs, Django is one of the most widely used frameworks, and its implementation of the Model–View–Controller (MVC) architecture is particularly effective.**

What are the tools and frameworks for building an MVP?

Model–Template–View (MTV) separates the data model, user interface, and control logic of an application into separate components, making it easier to manage and maintain.

However, in the MVC architecture, there's a lot of coupling, which may make it easier to write code at first, but could lead to issues during later adjustments and maintenance. When building an actual application, it's crucial to think very carefully about code coupling and to separate each domain or application. In essence, you should strive for a modular monolith that can be separated into smaller applications later on.

To further reduce costs and improve efficiency, it's a good practice to use bootstrap templates so that minimal work has to be done on the frontend side, and developers can focus on providing value.

Another approach is to separate the backend and frontend teams when a mobile application is a part of the MVP solution. The team can decide on an API layer to communicate between the server-side part of the solution and the mobile app that runs on people's devices.

When considering the development of an MVP for mobile devices, it may be beneficial to create a Progressive Web App (PWA). PWAs provide a native app-like experience on mobile devices, but are web applications that are typically accessed through a browser. They can be installed on a device for offline use.

The advantage of PWAs is that they share 100% of the code base with traditional web applications, making it easier to develop and maintain the frontend across multiple platforms. A PWA is a good approach for MVP development, as it allows for a seamless user experience on mobile devices while leveraging the existing web development skill set.

Developer tools for building an MVP

In addition to selecting the right code architecture and deployment strategy, having the right set of tools is critical for efficient software development. While the choice of developer tools may vary depending on the project and developer preferences, certain features such as code completion, static code analysis, and debugging tools can help speed up the coding process.

One popular choice among developers is PyCharm, which offers a comprehensive set of features for Python development. However, for those who prefer a more lightweight development environment, Visual Studio Code is an excellent alternative that can be customized with plugins to improve the developer experience.

Alternatively, developers who work in pairs and value tight cooperation may find Fleet, a new collaboration-first IDE from JetBrains, to be a good choice.

In addition to traditional IDEs, some developers may prefer a text editor like Vim with a customized configuration to make it a full-fledged IDE comparable to PyCharm or VSCode.

Testing the MVP

Testing is a critical component of building an MVP and it’s essential to ensure that the MVP is thoroughly tested on both the code level and the final solution level. **In the Python development ecosystem, the go-to frameworks for preparing necessary tests are unittest and PyTest.**

However, it’s equally important to test the overall behavior and predict test scenarios. To accomplish this, developers can turn to various tools, such as Selenium, Cypress, and Cucumber, which provide APIs to simulate user activity in a browser.

These tools help guarantee that the application performs as expected, enabling developers to identify and address issues before they escalate into major problems. The use of these tools is critical for verifying MVP functionality and ensuring a smooth user experience.

By simulating user activity, potential issues can be identified and resolved before they impact the end user experience. Additionally, automating testing accelerates the testing process and reduces the risk of human error, resulting in a more reliable and robust product.

Design tools for building an MVP

Having a well-designed user interface can significantly speed up the development process of any application, including MVPs. Adobe XD, Figma, and InVision are some of the most commonly used tools for creating designs that can be easily shared with a team of developers.

Clickable prototypes are an excellent way to visualize and test the product, ensuring that it meets the desired goals and requirements. By using UI component libraries such as Material UI, Ant Design, or CSS frameworks like Bootstrap or Tailwind, developers can quickly create a functional MVP with a polished user interface. These tools offer a variety of pre-built components and templates that can help accelerate development.

Moreover, there are many themes and templates available on the market that can be customized to fit the specific needs of an MVP. These themes often include custom components like social media cards, carousel galleries, and embeddable maps, which can help developers create a user-friendly and visually appealing product quickly.

How to choose tools and frameworks for building an MVP?

Building a successful product starts with developing an MVP, and selecting the appropriate tools and frameworks is a critical decision that can significantly affect your project's outcome.

In this section, we will explore the different approaches to selecting the right tools and frameworks for your MVP, emphasizing the importance of considering your strengths and weaknesses, consulting with others, and outsourcing.

Approaching the choice of tools and frameworks

When building a product, it's important to start by assessing your existing resources and considering what tools and frameworks will best support your project's vision and goals. This includes taking into account your team's skills and experience, as well as any limitations or biases that may impact your decision-making process.

Consulting with experts

To make the most informed decisions, it's also essential to consult with others, whether it's through seeking input from colleagues, hiring outside experts, or researching best practices and industry standards.

By doing so, you can stay informed about the latest advancements and potential pitfalls to avoid. Consulting with specialists who have recently started a new project and/or have extensive experience with a similar technology stack can help you understand which libraries or frameworks are worth considering.

On the other hand, you'll learn about the architectures that may seem attractive at first, but become difficult to develop or maintain over time.

Through expert consultation, you can uncover potential issues and learn about the pros and cons of your choices to proceed accordingly. Remember, the goal is to have a clear understanding of your options.

How to choose tools and frameworks for building an MVP?

Outsourcing

Outsourcing is another option to consider when selecting tools and frameworks, particularly if you don't have the necessary expertise in-house. Outsourcing allows you to tap into specialized knowledge and resources, providing an opportunity to work with experienced professionals who can guide you through the process and offer solutions tailored to your unique needs.

Outsourcing not only provides teams of developers, but also an entire ecosystem of support, including Product Owners, Scrum Masters, and Quality Testers. An experienced Product Technology Consultancy (PTC) team can provide an architecture based on their expertise, helping you understand the advantages and disadvantages of different approaches.

By taking a thoughtful and deliberate approach to selecting your tools and frameworks, you can help ensure that your MVP is built on a strong foundation that will support your product's success in the long run.

Approaches to building an MVP

Developing an MVP involves choosing an approach that is best suited to the specific needs and requirements of your project. There are several options available, including utilizing a pre-made project starter or custom development from scratch.

Project starter

A popular approach is to use a pre-prepared project starter that includes many of the common elements and configurations required for most MVP projects. This significantly reduces the initial setup time, providing a faster development process.

A sample project starter may include:

- a Dockerized Django app,
- a Dockerized frontend app,
- ready-to-use database configuration,
- and a makefile with commonly used commands.

Although this starter would still require some tailoring to the specific requirements of your project, it's a time-saving option compared to starting from scratch.

MVP approach suited for a planned product

At this point, your stakeholders and development team should analyze the pros and cons to choose an MVP approach that's best for their planned solution. If the MVP is expected to be built upon later, it's important to spend more time analyzing the solution architecture during the early stages of design and development. This is crucial to ensure the maintainability of the MVP in the long-term development of the full product.

Pitfalls of keeping an MVP that's too good to discard

On the other hand, despite the initial decision to discard the MVP after serving its purpose, it may sometimes still become the foundation of the final product. This can happen due to underestimating the budget required for further development or because the MVP is well-coded and requires only minor refactoring to be further maintained.

However, you may have concerns about the potential cost and time involved in rewriting your MVP from scratch. It's important to consider the long-term impact of maintaining your current code base versus making updates and improvements through a rewrite.

Initially, the goal of developing your MVP may have only been to quickly deliver features, which came at the cost of code maintainability. Continuing to add features without proper attention to code quality could result in a "big ball of mud" scenario where future maintenance becomes increasingly time-consuming and difficult.

To address these concerns, it's important to weigh the short-term benefits of maintaining the current MVP against the long-term benefits of a rewrite in terms of cost, time, and quality. A focused effort on testing and refactoring may make the existing code base more extendable and maintainable. However, neglecting quality testing could lead to big changes breaking the application.

In conclusion, when building an MVP, it's essential to choose an approach that fits your specific project requirements. Going with either a pre-made project starter or custom development is a viable option. By considering both alternatives, you can build a successful MVP that meets the needs of your project and its stakeholders.

Practical examples of various MVP development approaches

In order to provide you with a tangible demonstration of how these theoretical concepts can be applied, we've prepared several hypothetical scenarios for creating a project from scratch. Treat them as guidelines to help you make better decisions on the technology stack that would best suit your needs.

It's important to note that the number of Python developers continues to increase every year, along with the popularity of the language. This means that more frameworks, libraries, and packages will be improved to meet the growing demand.

As you read through the case studies, don't just stop at our examples. Consider how your MVP might evolve or what other elements it may require in the future, not just in terms of the code base. There may be additional needs, such as preparing datasets or crawling information, that could impact your technology choices.

That's why Python is an excellent choice for building an MVP. With Python, you not only get access to the language itself, but also to the entire ecosystem surrounding it—including production-ready implementations for web development, machine learning, data science, and data engineering projects. This versatility of Python allows the same team to tackle a variety of challenges.

Without further ado, let's dive into our case studies.

Case #1: Content management and minimal business logic

In this scenario, content management and general data manipulation are the primary concerns, with a projected initial user base of 100–200 users.

- The system is expected to handle different model instances, allowing users to create, modify, and add comments to content.
- We don't expect that all users will work on their content simultaneously.
- While providing a single-page application for displaying real-time content updates is desirable, it isn't mandatory for the MVP.

Considering these requirements, Django may be a suitable choice to start your MVP. Here are some of the reasons why:

- It offers simple solutions for data model preparation, such as using existing model classes, generating migrations, and connecting to a database.
- Asynchronous processing can be added by including Celery or another task queue in the tech stack.

- User and permission management is available out of the box, and fine-tuning the permission model is possible with additional, battle-tested solutions like Guardian.
- There are many existing libraries that provide essential features, such as payment integration with Stripe or advanced search capabilities with ElasticSearch.

If the Model–Template–View approach is chosen, which is the default in Django, additional features can be added with libraries like:

- Crispy Forms, which allows forms to be adjusted to the specific needs of the solution;
- django-ckeditor and django-tinymce, which offer WYSIWYG editing capabilities for formatted content.

Opting for a separate backend and a modern, responsive frontend application may be the best approach here. The Django REST Framework (DRF) with built-in authentication, data serialization, filtering, and more will provide the necessary features.

When it comes to deploying the final solution, given the requirements of this case, it seems unnecessary to go beyond hosting the final application on a single VPS. Containerizing the app (or both apps, if the backend and frontend are split) may still be worth considering, especially since the development environment is already containerized (from the project starter mentioned before), and the changes needed for a production-ready setup would be minor. This approach will pay off the moment the MVP is successful and a single VPS or instance deployed on a PaaS isn't enough.

With these requirements in mind, we recommend the following tech stack for this case:

- Django backend with a relational database (PostgreSQL, MariaDB, or any other suitable solution supported by Django).
- Either classic template-based views or a separate frontend app in React.
- Django Admin as a back-office management panel (suitable for typical moderating actions, user management, etc.).
- Deployment on a single VPS or a PaaS solution (e.g. Heroku) with the possibility to scale up if needed.

Case #2: Content management system with custom business logic and heavy processing

Our goal is to create a system that can track changes in the market for specialized devices and adjust the prices of our products based on supply, demand, and competitor pricing.

- This system will be used by 20–50 users who will manage the same shared model instances.
- The changes are displayed as a diagram with strict rules determining which changes can occur.
- The business expects the system to handle consecutive changes occurring in succession.

After analysis, we determined that the system will likely have 2–3 data models, including authentication. The main role of the system is to process input data using multiple steps, with the decision to run a step being determined by the processed data itself.

In this scenario, a lightweight stack based on FastAPI, Flask, or Falcon will be sufficient. Here's why:

- The selected framework's lightweight overhead will allow for immediate processing of input data as soon as it reaches the API endpoint.
- The business logic can be implemented as pure Python classes, enabling thorough test coverage for all edge cases that may occur in the processing flow, separate from the API communication layer.
- The final results can be persisted using a variety of solutions other than a relational database, such as Amazon S3 or a NoSQL database, as long as it meets the system's expectations.

We recommend using a NoSQL database, such as MongoDB, because it allows related models to be stored as a single document, reducing the risk of data integrity issues.

The proposed tech stack for this solution includes:

- FastAPI or another lightweight framework;
- pure Python for the business logic;
- deployment on a VPS, with containerization as a potential benefit, but not a requirement;
- MongoDB or another NoSQL solution that can store related models as a single document.

Case #3: Integration between services (service broker)

Our objective is to aggregate data from multiple sources, compare and sort them by specific criteria, and display them on a unified platform. It's critical to avoid serving users with data older than 5 minutes while striving to be as up-to-date as possible.

To achieve this, we require a lightweight API framework such as FastAPI or other suitable alternatives. Below the API layer, we need a per-data provider service to handle specific cases for each data source. For aggregation of partial results, we can use Redis as a temporary cache with low performance overhead.

Since we don't need to persist processed data, the key challenge is to establish an effective cache invalidation strategy to ensure data freshness. We must prioritize fresh data where possible and be prepared to handle limitations from external services.

As the number of users increases, we may need to scale up the infrastructure. Therefore, we'll deploy the application in a containerized cluster, allowing us to scale up or down easily in response to traffic changes.

In this project, our tech stack will include:

- a lightweight API framework (FastAPI/Falcon/Flask or any other suitable alternative);
- containerized deployment (most likely on a PaaS solution such as Heroku) that allows us to scale our application quickly as traffic increases;
- plain Python code to fetch and manipulate data using libraries such as Requests or HTTPX to obtain the required information;
- in some cases, we may use Scrapy or another web crawling solution to gather data from sources that don't provide an API but are still valuable to us.

To ensure scalability, we'll deploy the application on a Kubernetes cluster. Additionally, the frontend application that serves our API should provide a single-page, auto-refreshing experience that takes into account the maximum latency of 5 minutes.

Case #4: Data engineering or data science project

The objective of a data engineering or data science project is to build an Extract, Transform, Load (ETL) system that can gather data from various sources, transform it into a uniform format, and store it in one or multiple databases.

This may involve:

- calling API services regularly,
- collecting data from crawlers on different servers,
- integrating with a data provider to gather data.

Scalability is a crucial factor to consider when building such a system due to the substantial amount of data that needs to be processed. To enable easy deployment across multiple servers and to manage API calls, the focus should be on a lightweight and stateless service with minimal overhead. A load balancer should also be used to distribute incoming requests evenly across multiple servers.

Apart from processing data, there may be a need to aggregate it and run cron tasks at regular intervals, such as viewing data statistics in a dashboard or as graphs. Additional tools such as Metabase or PowerBI can be used to accomplish this.

For search functionality, creating your own search engine would be an overkill for the MVP. Instead, opt for ElasticSearch, along with a search API that translates queries to ES queries to return the data. This approach allows all queries to be logged, and access to data can be granted or rejected based on the user's subscription or access permissions.

The proposed technology stack for this project includes:

- mini web frameworks like Flask or FastAPI to build the API endpoint;
- Docker for easy deployment and scalability;
- pure Python code with the option of using PyPy for optimized performance;
- SQLAlchemy and Alembic for database management;
- a load balancer to distribute incoming requests evenly across multiple servers.

To create an efficient and effective data collection system, it's essential to focus on:

- high scalability,
- a lightweight project structure,
- the possibility of breaking down parts of the system into microservices.

Case #5: Hosting a machine learning model

In this case, we have an ML model used for making API calls to predict the result based on the data provided by the model.

As seen in the previous case study (#4), users won't produce a lot of content. However, some additional information may be generated as a byproduct of an API call. For instance, you may want to log every prediction (if it's GDPR-compliant) or simply acknowledge the fact that a user received a prediction, such as a subscription for a specific number of calls per month or a free prediction every hour.

Practical examples of various MVP development approaches

As you can see, there isn't much content management involved here, except for the business logic that limits and handles the API calls. This will generate a lot of data, but it will only be one type of data.

ML models are usually memory-intensive, so it's a good idea to avoid a system that requires a lot of CPU power. Instead, a virtual machine with more memory than CPU speed is needed.

It's also essential to consider the duration of the prediction. If you want to have an MLaaS (Machine Learning as a Service, which isn't an official aggregation yet), it's necessary to create a model that's fast, lightweight, and provides accurate predictions.

Despite what we've previously mentioned, it may be necessary to have several specialized models. For instance, one model could categorize the problem and choose a specific ML model to solve it.

Therefore, instead of a one-size-fits-all model, you can have a specialized collection of models for each specific task. It's important to keep in mind that it may not always be possible, but as software developers, it's crucial for us to remind ML engineers that using multiple different models is also an option. Having one excellent model might not be enough if it needs to have more characteristics to allow your project to be monetized more easily.

The proposed technology stack here would be:

- a mini web framework like Flask or FastAPI;
- SQLAlchemy and Alembic;
- a machine learning model handling prediction tools, such as PyTorch, scikit-learn, NLP tools, CV tools, NumPy, and Pandas;
- a caching tool for the same queries to optimize the model usage time.

Final thoughts on maximizing value with Python

A well-crafted minimum viable product can bring immense value to your business. Software developers play a crucial role in helping companies like yours determine if your idea has the potential to succeed. To design a suitable MVP, your team needs to know the details of the planned product.

Final thoughts on maximizing value with Python

Python provides a flexible framework for various architectures, such as CMS-like applications, service brokers, machine learning APIs, and data science projects. The language enables us to add ML features to existing projects and adapt to different requirements. Additionally, most Python programmers can manage multiple projects, allowing them to deliver solutions that are both innovative and efficient.

With the growing popularity of Python, there's a vast repository of existing packages that can be used to build your final solution. Given the rising number of Python specialists every year, we expect this trend to continue.

Part II

The Developer Report

Introduction

In 2022, Python reached an all-time high of 17.18% market share and took first place in the TIOBE Index, continuing its run as one of the most popular programming languages in the world.

Python has undeniably taken the lead in a wide range of tech-related fields: web development, data analysis, machine learning & AI, system administration, software testing & prototyping, desktop development, network programming, and many other domains. This shows that it's the top industry choice at the moment.

That made us think: how amazing would it be to bring together over 100 Python experts to gather all their knowledge of Python's current state as well as the changes coming in the future?

So that's what we did. We asked our Python specialists about the situation of the language, its future, and the most interesting trends emerging in the area.

Below you'll find the results of the first official Python Tech Radar survey conducted at STX Next, Europe's largest Python software house. Our talented group of software developers who live and breathe Python answered 30+ questions to share what's coming up for the most popular programming language.

Check out the report to find out their invaluable insights on what's going to shape the future of the Python world!

7	30	100+	5
experts	**questions in the survey**	**total responses**	**areas**

REPORT 01

Who answered the survey?

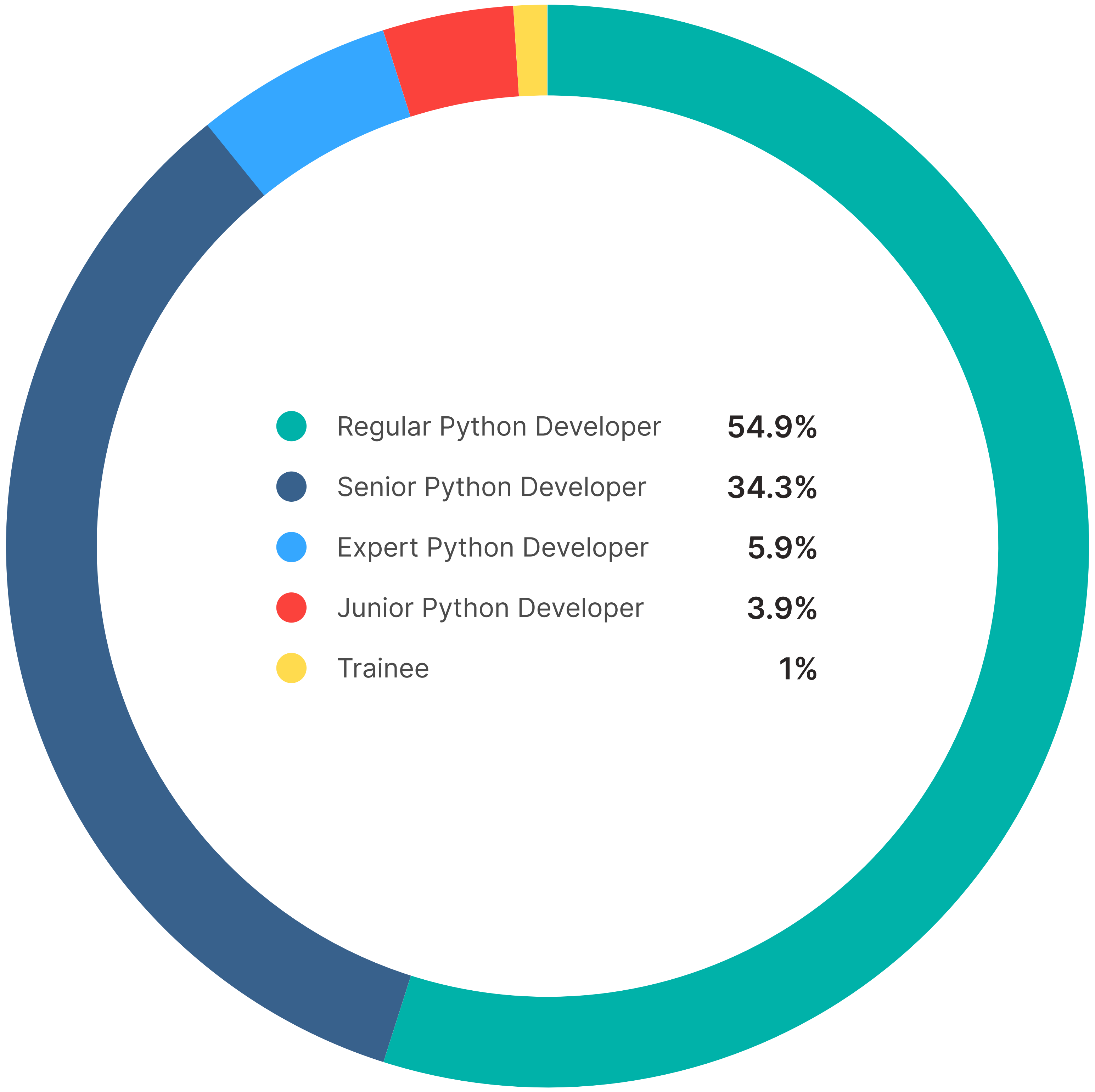
Seniority and experience
of the respondents

Regular and senior Python developers shape the industry

Monitoring the expertise of developers utilizing the tools and languages companies rely on is a key aspect of understanding and improving the skills of their teams.

Our survey results show that over half of the respondents identify as **regular developers (54.9%)**, while a significant portion identify as **senior developers (34.3%)**.

The high percentage of regular, senior, and expert developers (overall 95%) among the respondents suggests that our company, as well as the industry as a whole, holds a wealth of talented and experienced professionals well-versed in Python.



What is your seniority level?

● Regular Python Developer

54.9%

● Senior Python Developer

34.3%

● Expert Python Developer

5.9%

● Junior Python Developer

3.9%

● Trainee

1%

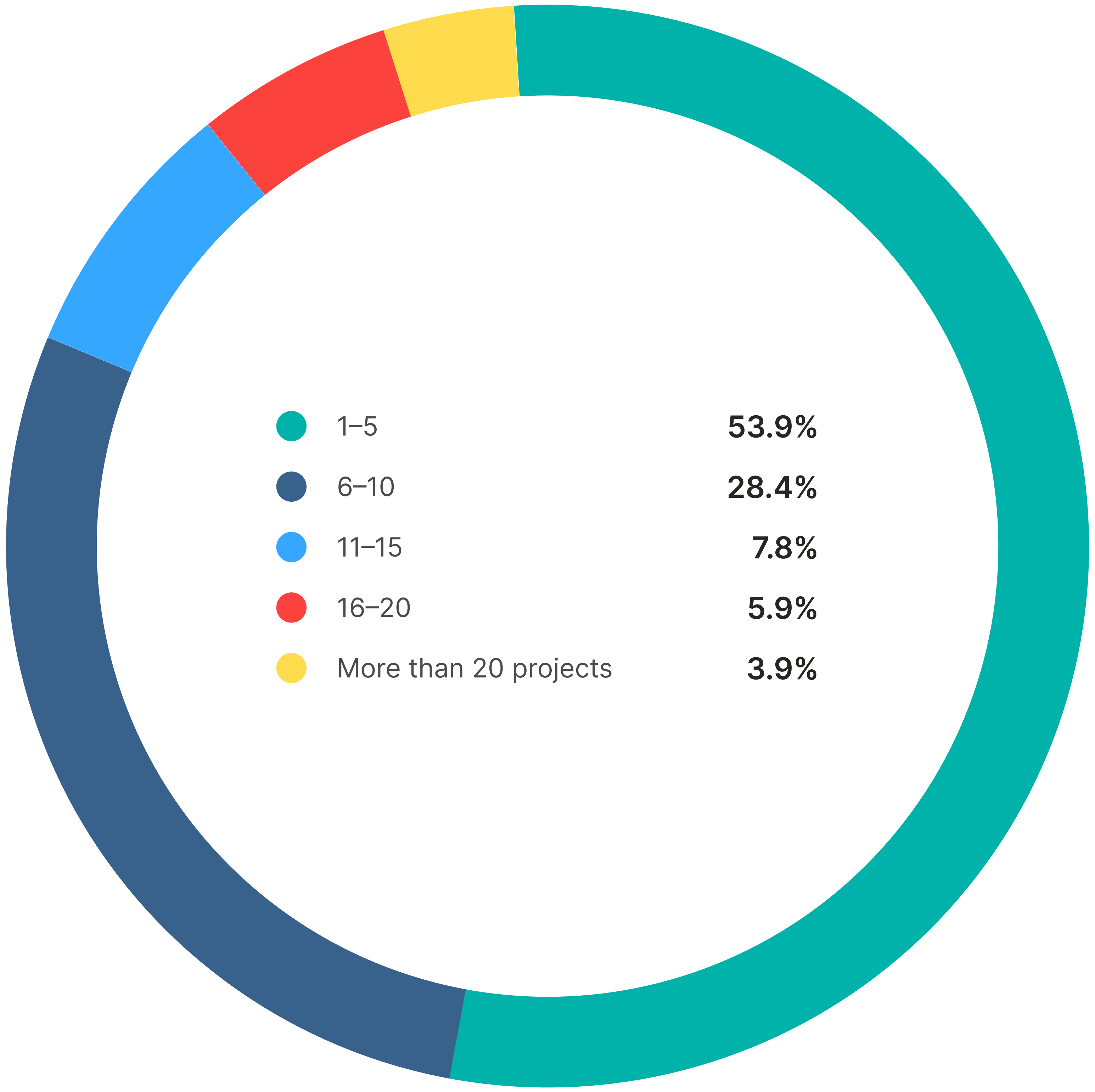
Real-life, commercial experience counts

One of the key measures of a developer's experience and expertise is the number of commercial projects they have worked on. In our survey, we asked the respondents how many such projects they have worked on so far.

The results show that the majority of respondents have worked on **1–5 projects (53.9%)**, while a significant portion have worked on **6–10 projects (28.4%).** A smaller percentage of respondents have worked on **11–15 projects (7.8%),** with **5.9% having worked on 16–20 projects and 3.9% on more than 20 projects.**

Overall, these results show that **almost half (46%) of the respondents have worked on more than 5 commercial projects,** attesting to the strong foundation of knowledge within the industry and the talent and professionalism of Python developers.

It also highlights the respondents' commitment to delivering high-quality software solutions for their clients.



In general, how many commercial projects have you worked on so far?

● 1–5
53.9%

● 6–10
28.4%

● 11–15
7.8%

● 16–20
5.9%

● More than 20 projects
3.9%

Expert commentary

A glance at the years of experience, seniority, and number of completed projects can provide valuable insight into a developer's work style and aid in evaluating the qualifications of potential team members.

Typically, developers work on one commercial project per year, this pattern is evident up to the senior level. If the number of completed commercial projects significantly exceeds the number of years of experience, it may suggest a lack of depth in the programmer's engagement or that the project was small-scale, which doesn't favor the effective development of skills.

On the other hand, high-level experts sometimes work on multiple projects per year. This is often a part-time, long-term engagement as an advisor or reviewer of technical decisions made by a team. It can also be a short, but intense, intervention engagement.

The results also reveal the profile of an average senior developer, a professional with at least 4 years of experience who has completed more than 5 commercial projects. Developers who call themselves seniors but do not meet these criteria should be thoroughly checked before being added to the team.

Marcin Zabawa

Director of Core Services

@ STX NEXT

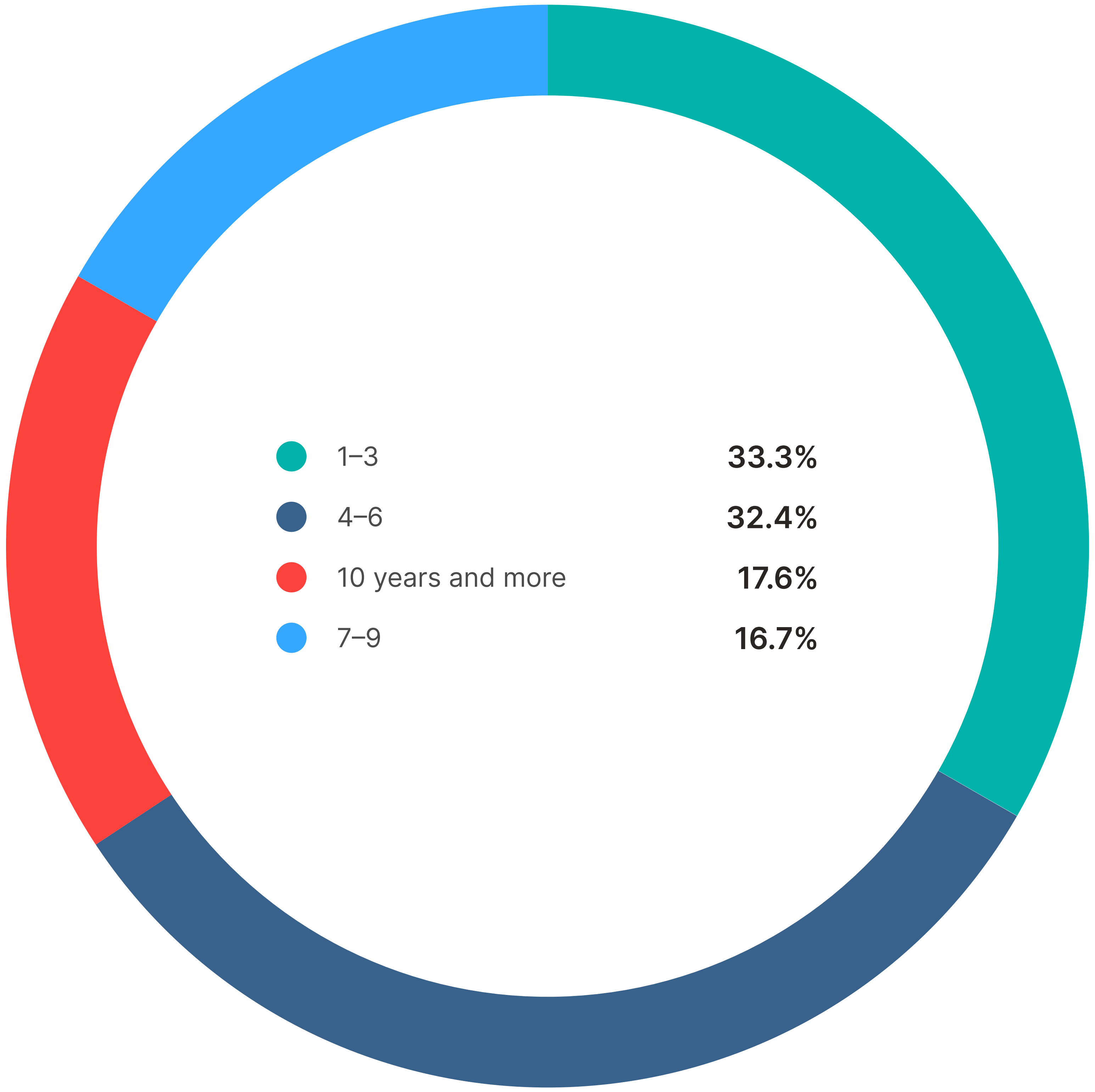
Diverse perspectives drive successful teams

When it comes to evaluating a developer's expertise, the number of years of experience they have in coding is an important consideration. In our survey, we asked respondents to share their experience level in this area.

The results show that **33.3% of respondents have 1–3 years of experience,** while **32.4% have 4–6 years.** Almost 1 in 5 developers have 7–9 years of experience (16.7%), and the same is true for those with **10 years or more (17.6%).**

These results demonstrate the wide range of experience levels among the developers, with a large portion having more extensive experience in the field—**65.8% of respondents have more than 5 years of experience.**

All in all, the majority of the survey respondents have a significant amount of Python experience, with some fresh perspectives from less experienced developers mixed in.



How many years of experience do you have in coding in general?

● 1–3	● 4–6	● 10 years and more	● 7–9
33.3%	32.4%	17.6%	16.7%

Expert commentary

Years ago, Robert C. Martin estimated that the number of programmers doubles every five years, leading to the conclusion that half of the people working in the field have less than five years of experience.

When a company has more experienced developers than the industry average, it must be aware of the challenges related to this situation.

The modern world relies heavily on software, which creates a high demand for new software engineers who are both reliable and proficient. To meet these requirements in an industry with a high turnover of inexperienced programmers, a company must develop a strategy of building healthy, balanced teams where craftsmanship practices are emphasized and experienced professionals guide younger colleagues. Less experienced developers are also important as they bring in a breath of fresh air, original ideas, and a non-conformist attitude, which experienced experts sometimes lack.

The graph illustrates a healthy balance in experience distribution. This approach requires substantial knowledge and experience in management, but the results are worth it.

Mikołaj Lewandowski
Expert Python Developer

@ STX NEXT

REPORT 02

Python now

State of the language
and general usage

Python's simplicity and versatility wins over developers

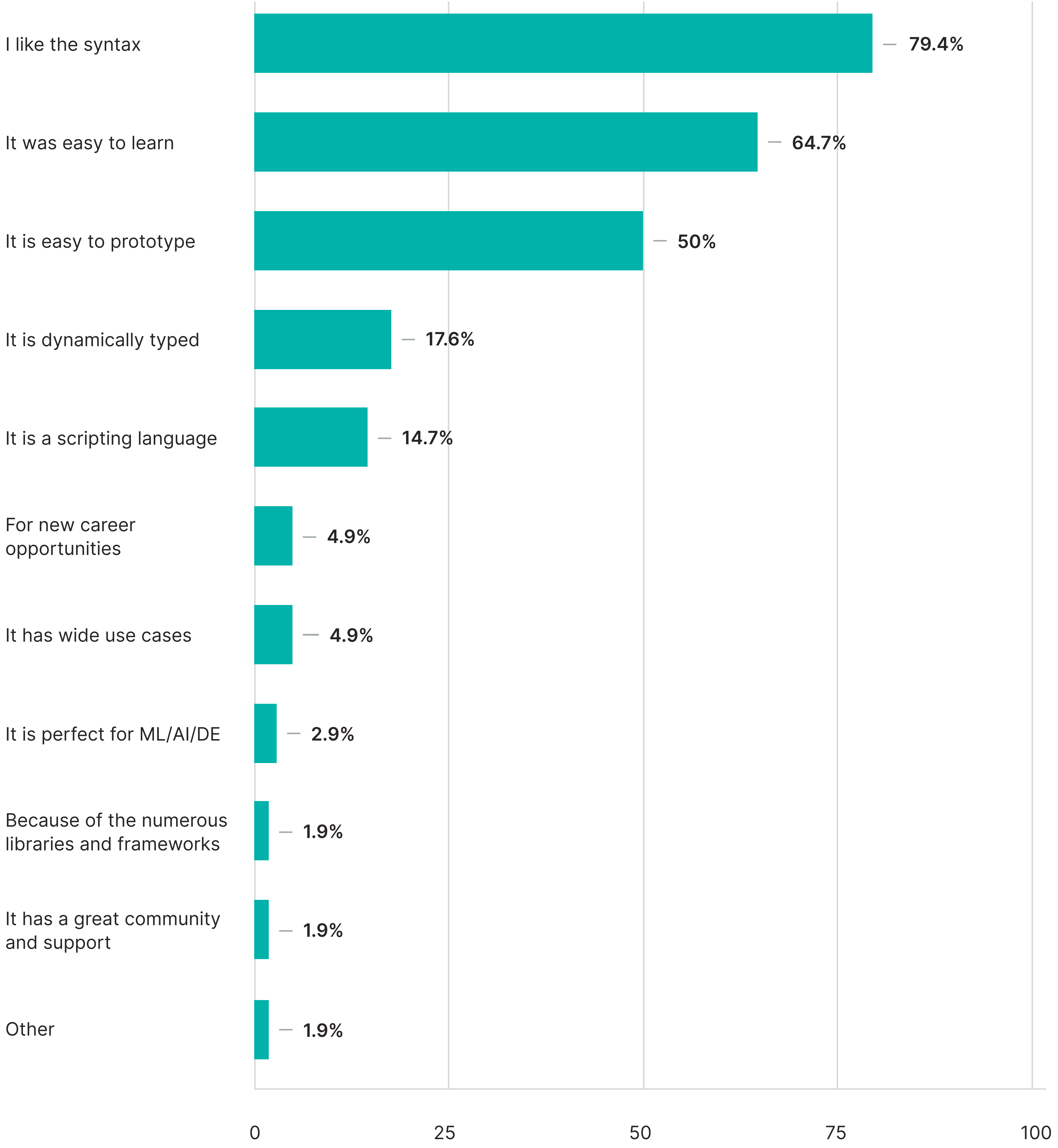
Python is an incredibly powerful and widely used programming language; developers love it because it's simple, versatile, and accessible. Businesses of all sizes and industries, on the other hand, are drawn to Python for its reliability, flexibility, and scalability.

In our survey, we asked the respondents why they chose Python as their go-to language.

The results reveal that an overwhelming majority **(79.4%) chose Python for its clean and concise syntax,** which makes it easy to read and write code. In fact, many of the developers **(64.7%) cite Python's ease of learning as a key factor in their choice.** For others **(50%), the ability to quickly prototype** ideas is a major advantage of Python.

Why did you choose Python?

(select max. 3 answers)



Developers take notice of Python's evolving ecosystem

Python has come a long way in recent years, and developers have taken notice. In our survey, we asked the respondents what has changed in Python for the better.

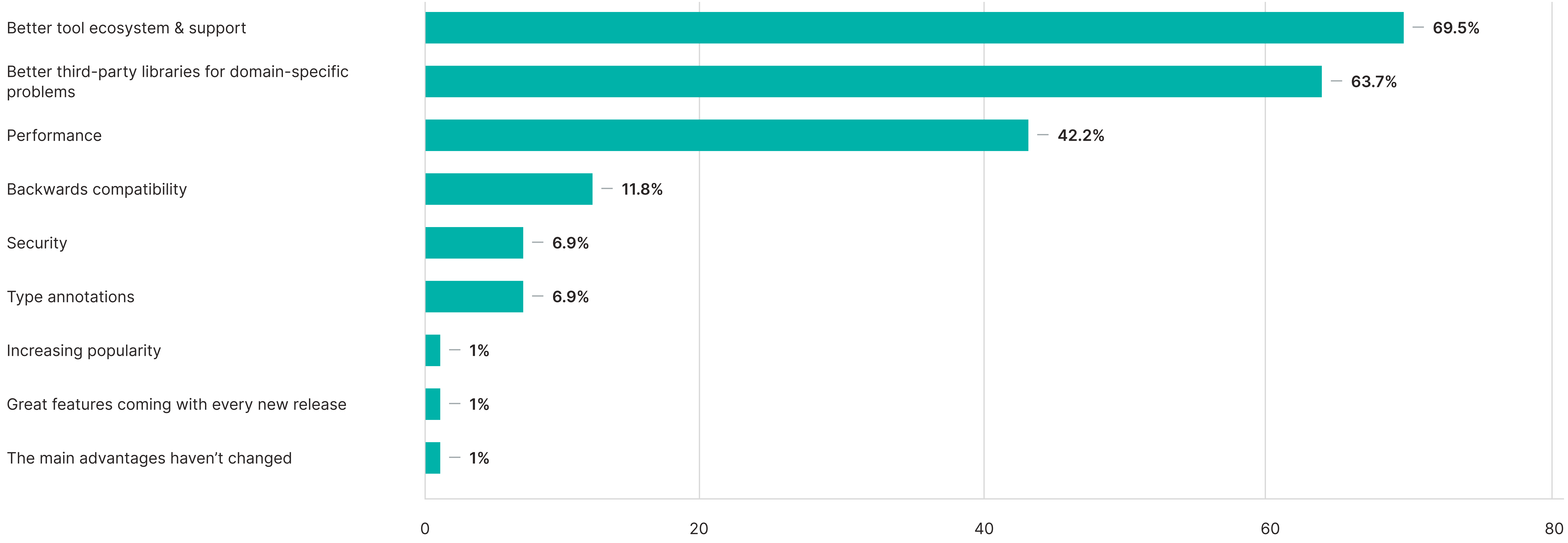
The majority of respondents **(69.5%) cite the better tool ecosystem and support** as a key reason for Python's recent growth and success. A similar number **(63.7%) point to the availability of better third-party libraries for domain-specific problems.**

Other notable improvements include **performance enhancements (42.2%), backward compatibility (11.8%), and increased security (6.9%).**

Overall, the respondents' insights highlight the dedication of the Python community to supporting and advancing the language, which is a big part of the reason why both developers and businesses are choosing Python as their programming language of choice.

What has changed in Python for the better in recent years that makes it a good choice when it wasn't before?

(select max. 3 answers)



Unique performance issues remain a key challenge for Python users

Python is undoubtedly a top programming language at the moment, but it comes with its own set of unique challenges.

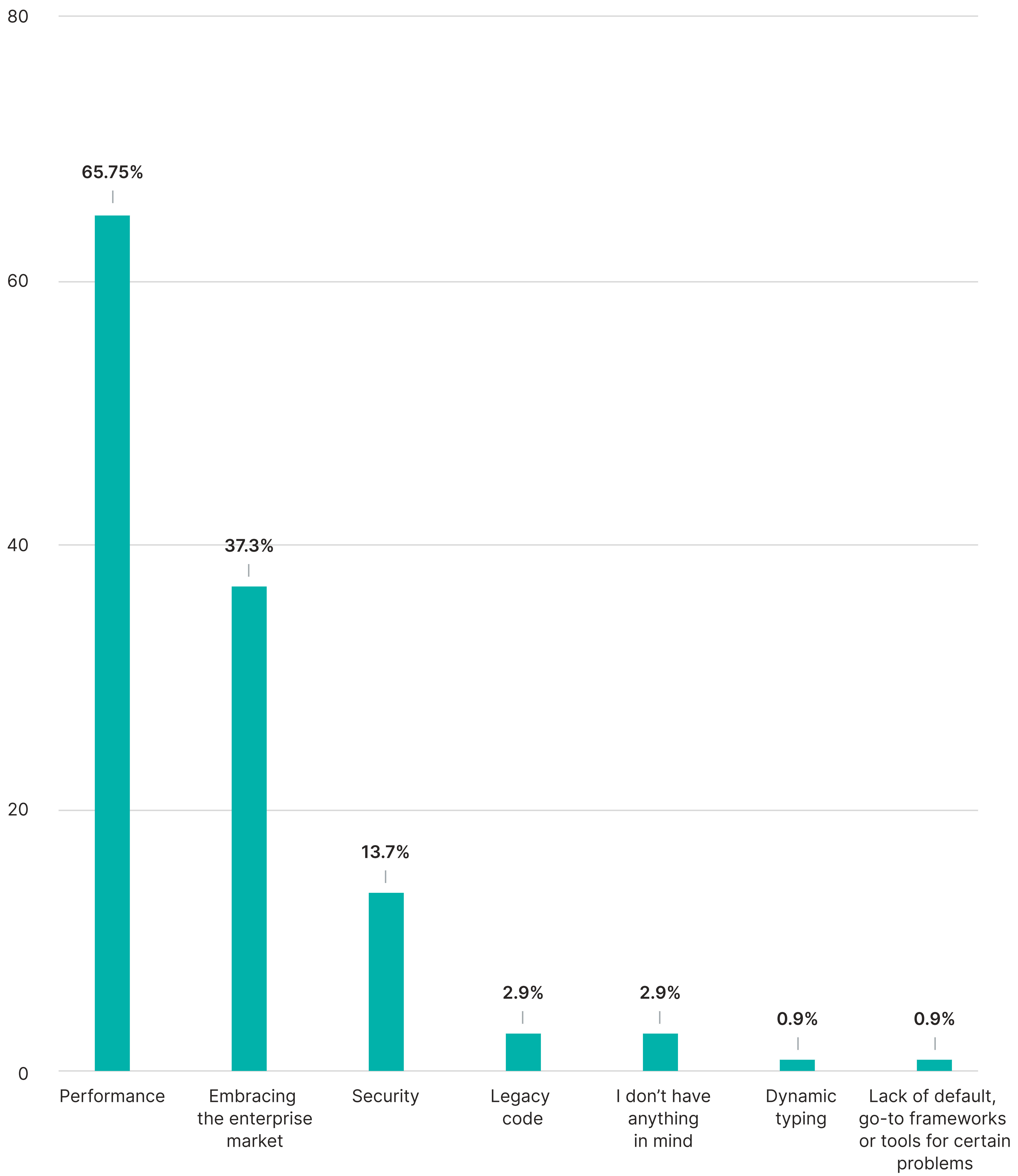
Every software development company knows the difficulties of staying up-to-date with the latest technologies and best Python practices. The best way to address this issue is to ask the team about the biggest challenges they currently face when using it.

The majority of respondents **(65.75%) cite performance as a key challenge**, while a significant number **(37.3%) identify the difficulty of embracing the enterprise market** as a major challenge. These results highlight the importance of optimizing code and improving performance when working with Python. They also speak to the ongoing efforts to expand the use of Python beyond smaller companies and startups and into larger, more established organizations.

No language is perfect, and Python is certainly no exception. But Python's unique position as an open-source and widely adopted language means it's well-positioned to proactively address any challenges that the future may bring.

What are the biggest challenges of using Python right now?

(select max. 2 answers)



Expert commentary

What makes Python particularly appealing now is exactly what has been achieved by initially sacrificing its performance. Great features such as a low learning curve, fast prototyping, and a unique syntax are inherent in the nature of Python, and performance was the cost that had to be paid for that.

However, Python's performance is a widely debated topic and it's not a problem that remains unaddressed. The performance of Python is continuously improving, and users are recognizing and appreciating this. This survey clearly shows this.

The same is true for the enterprise market. While Python is well-suited for it, there are a few challenges that arise when using it in this context. Once again, however, some of them are old labels automatically, though wrongly, associated with this language.

That being said, with the constant evolution of Python, these challenges are being mitigated. This is sure to lead to increased use in various areas, many of which will be closely related to the enterprise market.
I think that's inevitable.

Marcin Zabawa
Director of Core Services

@ STX NEXT

Python is closely associated with web development

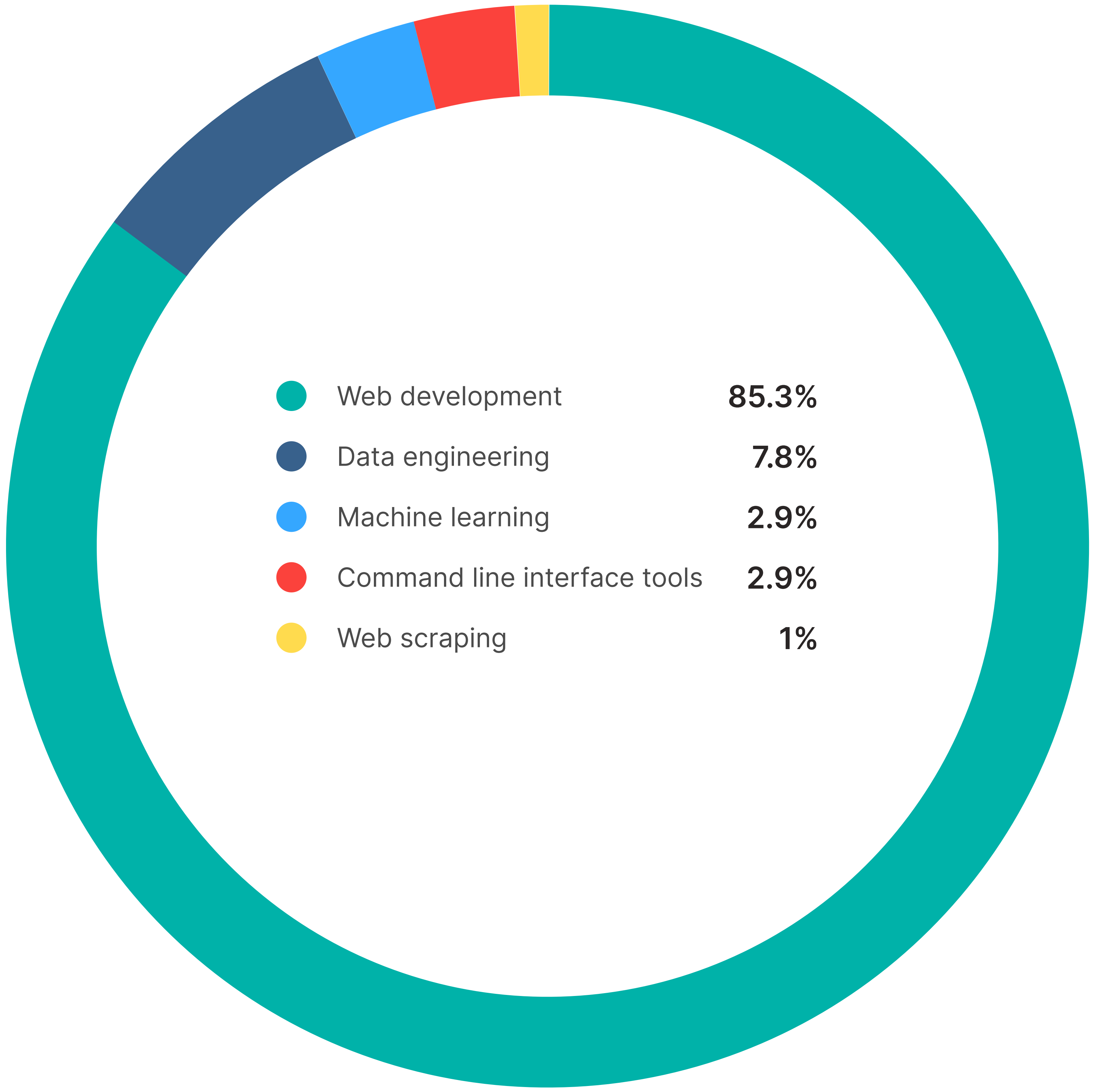
Python is a language that has it all. From web development to data engineering and machine learning, there's nothing it can't handle.

So it comes as no surprise that developers use it for a wide range of projects and purposes. That's why in our survey, we asked the respondents what exactly they use Python the most for.

Unsurprisingly, **web development emerges as the top use case for Python, with a whopping 85.3% of developers indicating that they use it primarily for this purpose.** This is a testament to Python's strength as a language for building robust and scalable web applications.

A smaller number (7.8%) use it mainly for data engineering, while a few (2.9%) use it primarily for machine learning or command line interface tools.

From building dynamic websites to harnessing the power of data and machine learning, Python is an invaluable tool for the continued growth of the world of tech.



What do you use Python the most for?

● Web development

85.3%

● Data engineering

7.8%

● Machine learning

2.9%

● Command line interface tools

2.9%

● Web scraping

1%

Developers choose the best Python versions to stay current and backwards compatible

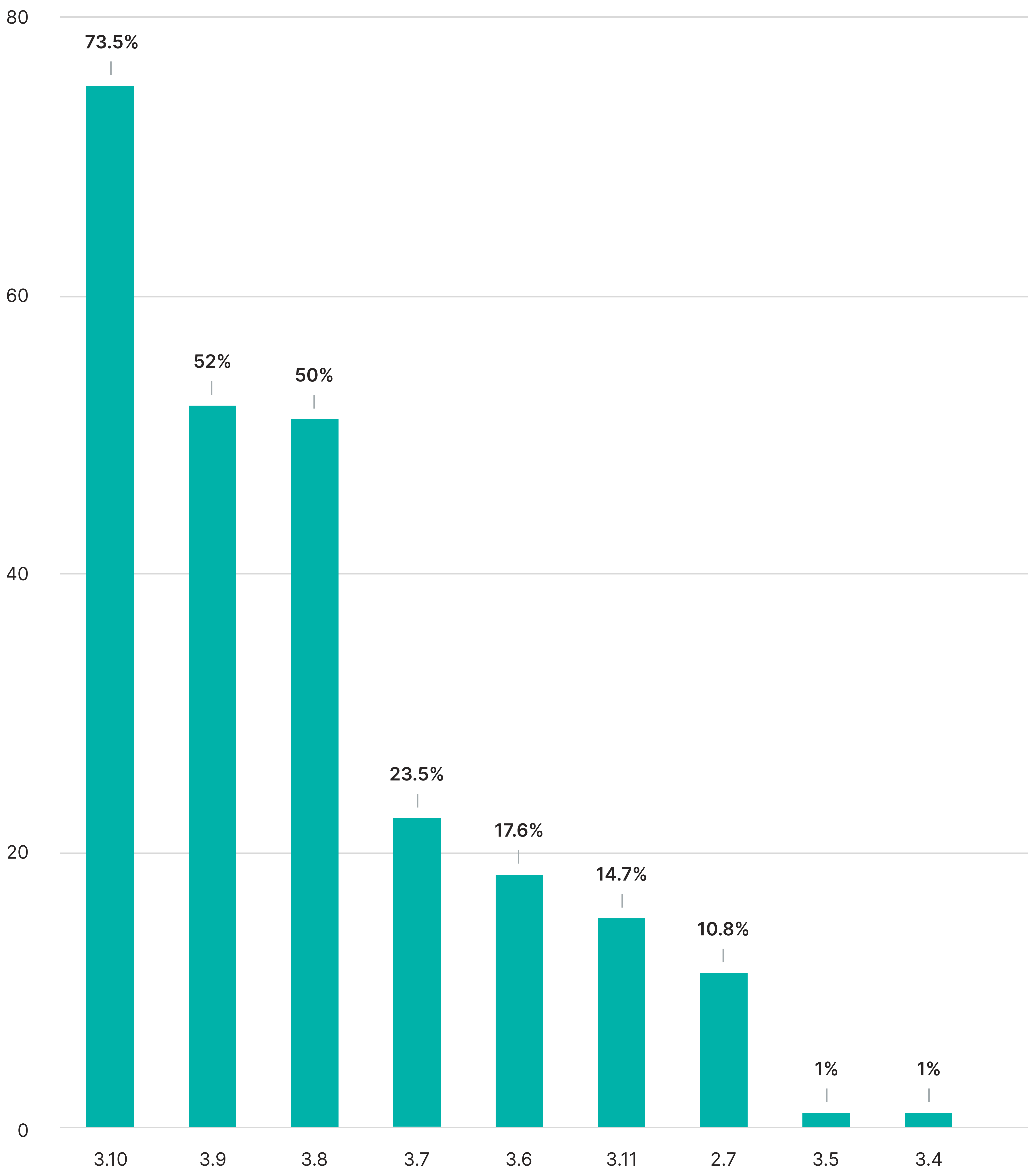
It's clear from the results of our survey that **the majority of the respondents are using fairly recent (but also well-established) versions of Python, with 3.8 and 3.9 being the most popular at 50% and 52%, respectively.** This is a good indication that the surveyed developers are staying current with the latest developments in the language and are well-equipped to handle the demands of modern software development.

However, it's also interesting to note that a significant portion of respondents are still using version 3.6 and 3.7 at 17.6% and 23.5% respectively. We can even see version 2.7 being used. **This suggests that there may be some projects or legacy code that still rely on older versions of Python.**

Overall, the surveyed developers' usage of Python versions showcases their dedication to adapting to the ever-evolving landscape of software development, while also showing their ability to work with legacy code, an important aspect of the software development industry.

Which Python version do you use?

(select max. 3 answers)

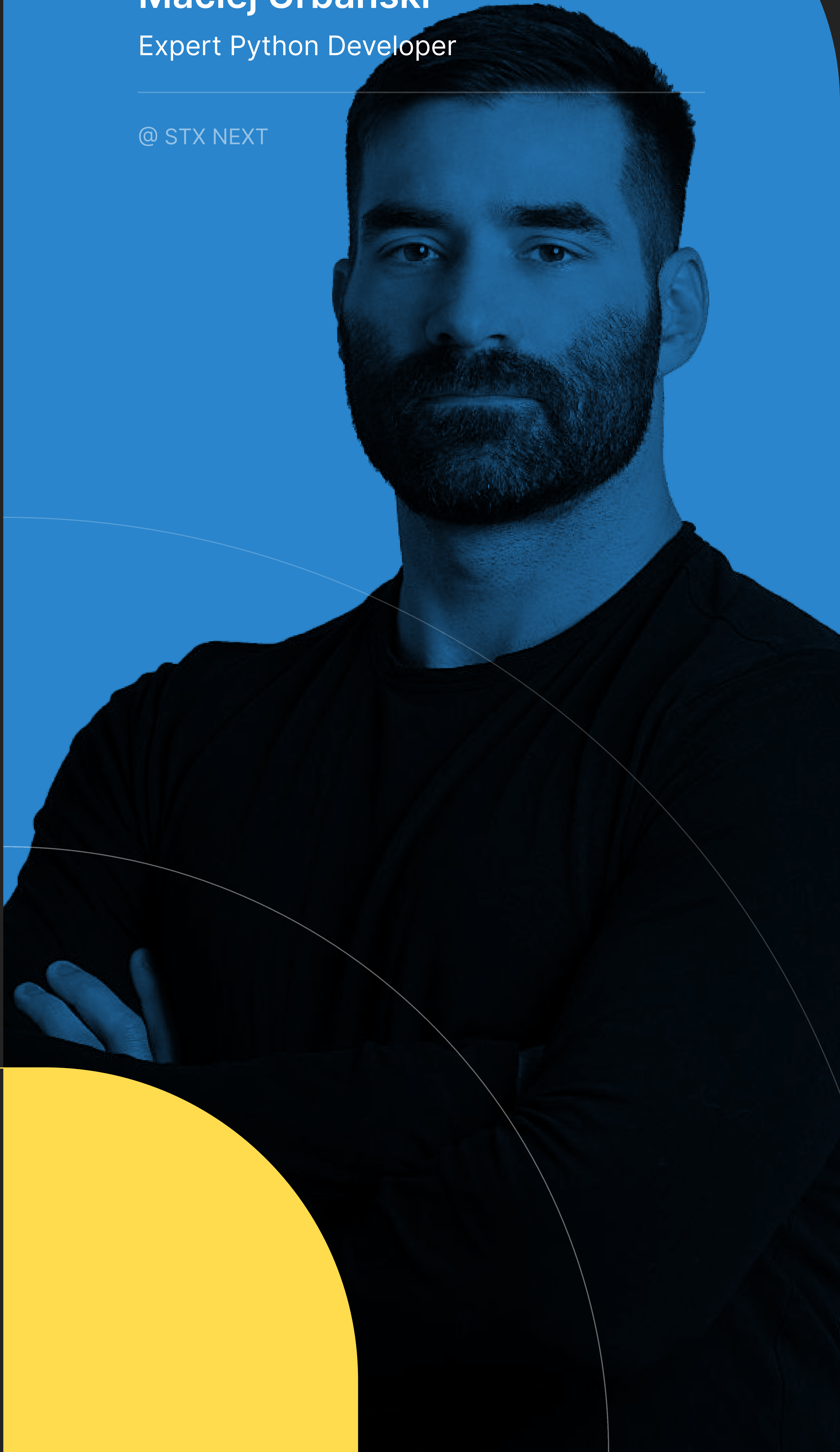


Expert commentary

A high percentage of Python 2.7 users is a sad reminder that this version is still commonly used in legacy projects despite its official End of Life. While Python 2 won't be anyone's choice for the next project, we can't forget about it just yet.

Python 3.6 was the first to surpass Python 2.7 and since its release, upgrading to newer versions has been both more rewarding and not nearly as problematic as it was when transitioning from version 2 to 3. The rapid uptake is mostly due to the "killer features" such as the "f-string" in Python 3.6. Apparently, developers appreciate the added convenience.

It's worth noting that this survey was conducted around the release of Python 3.11. So, why do we see so many users for a version that had barely made it out of the door? Python 3.11 brings significant performance improvements, which is a huge benefit for both developers and businesses using Python in their operations.



Maciej Urbański

Expert Python Developer

@ STX NEXT

Python developers embrace the evolving feature set of the language

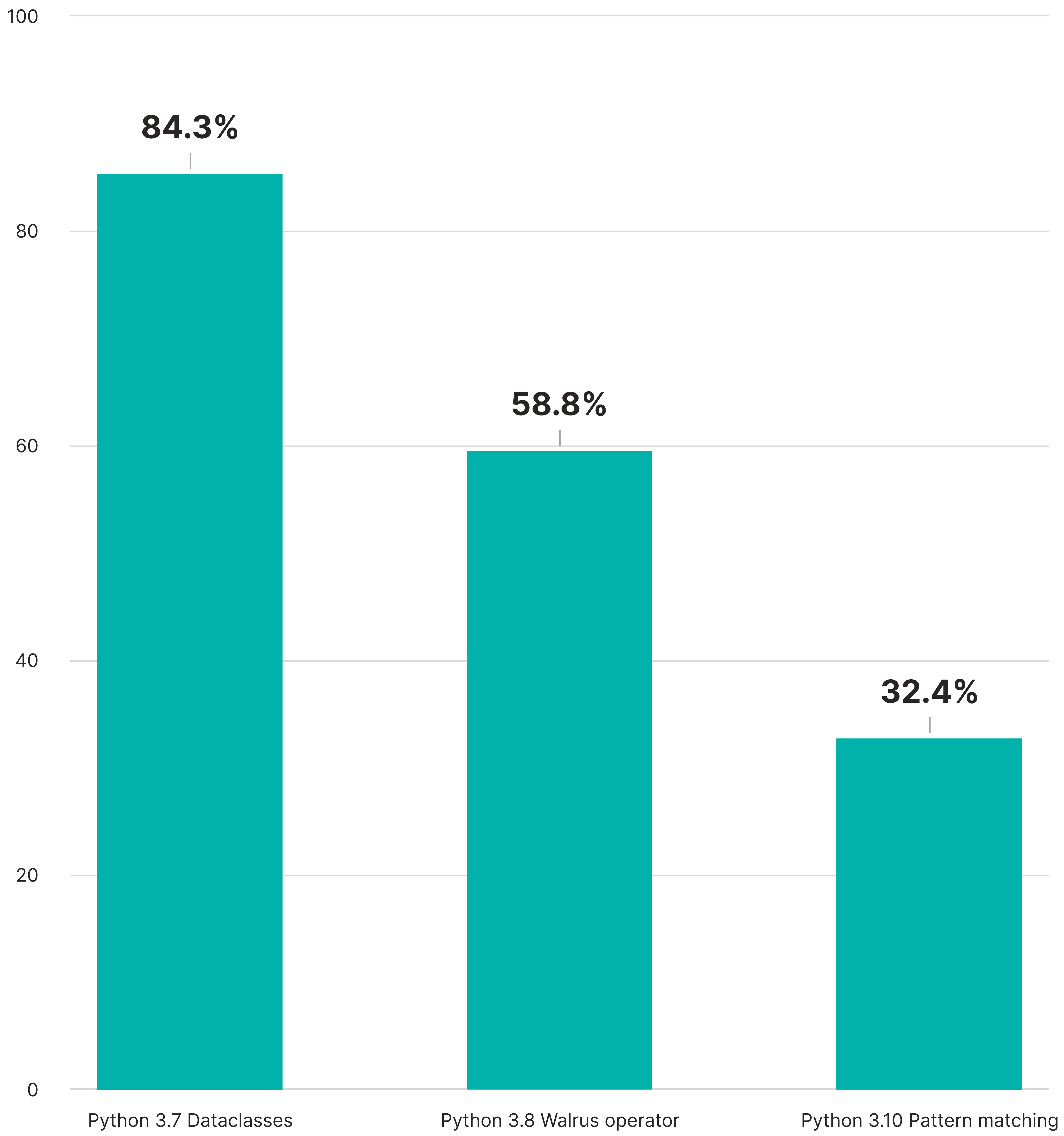
Python 3.7, which was released on June 27, 2018, introduced a new feature called “dataclasses.” These provide a foundation for object-oriented code without the need for boilerplate. **It’s clear from the results of our survey that this feature has become widely used based on the 84% adoption rate among the respondents.**

Additionally, a year later, Python 3.8 introduced the “Walrus operator,” which was met with initial controversy and led to Guido van Rossum’s resignation from his role as Benevolent Dictator for Life of the Python core development team. **Despite this, developers have come to appreciate the convenience of the := operator, with a 58% adoption rate in our survey.**

Finally, the “pattern matching” feature was released in late 2021. **Although developers have had just under 2 years to become familiar with it, 33% have already used it.** This feature greatly simplifies nested “if” statements, resulting in simpler and more concise code, which is a goal many Python developers strive for.

Which new Python features do you use?

(you can select all the 3 answers)



REPORT 03

Education & development

How the respondents grow their skills

Developers place a strong focus on maintaining their proficiency in web and software development, ML, and AI

Continuous learning is a key aspect of a successful software developer's career—of that there can be no doubt.

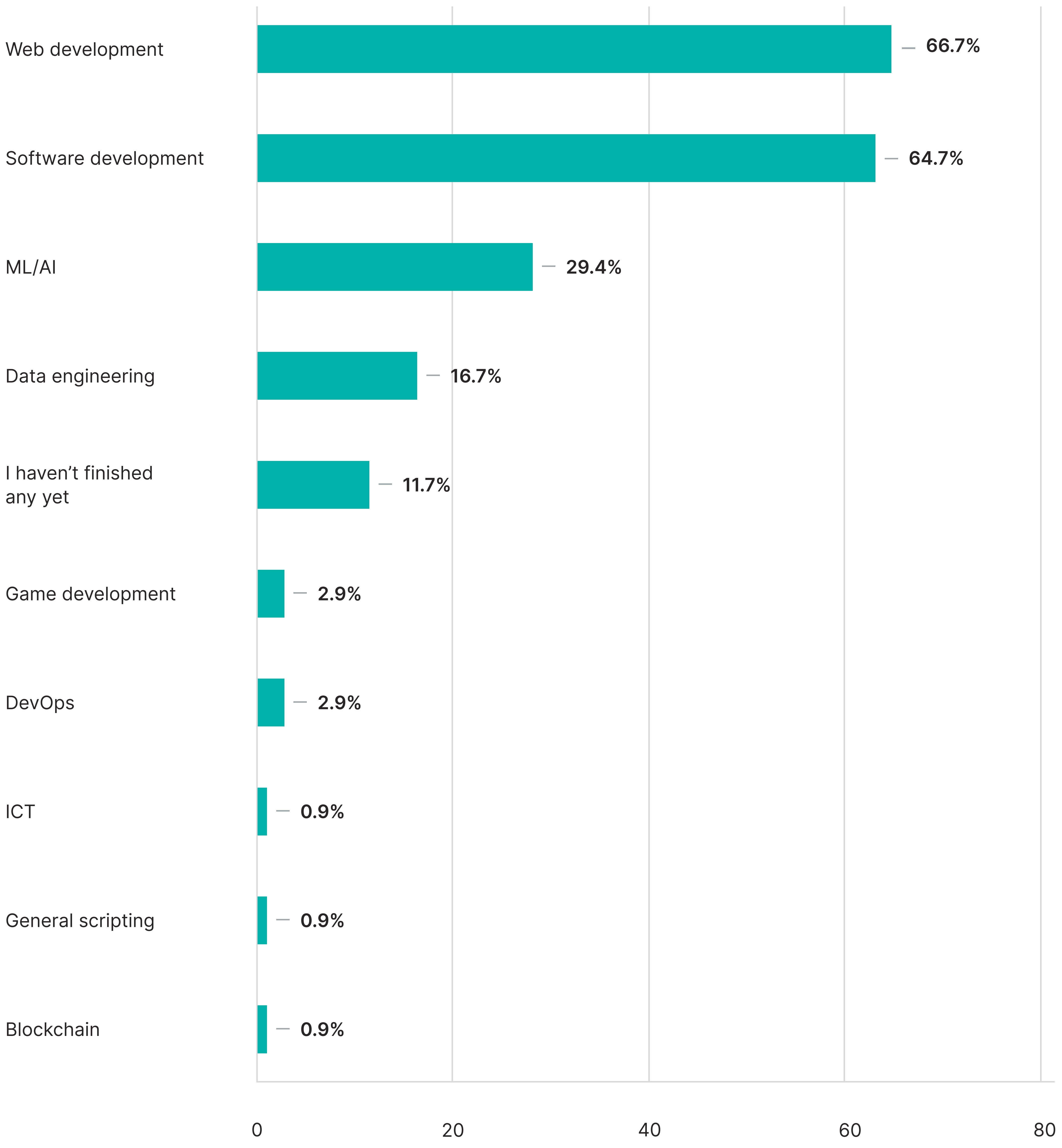
The survey results indicate that **web development** and **software development** are the most commonly pursued areas of study, **with 66.7% and 64.7% of respondents having completed courses in these disciplines,** respectively.

Furthermore, almost **1 in 3 respondents (29.4%)** has completed courses in **machine learning** and **artificial intelligence**, reflecting the increasing industry demand for these skills.

Overall, these results show that the surveyed developers are dedicated to staying up-to-date with the latest technologies and best practices, and are committed to honing their skills in rapidly evolving fields.

What courses have you finished? Select the disciplines in which you completed at least one course

(select max. 3 answers)



Self-learners dominate the programming landscape

Learning to code is a journey that's unique for each individual. In our survey, we found that self-learning is the most popular method among the respondents, with a **whopping 93.1% of Python developers saying they taught themselves how to code.**

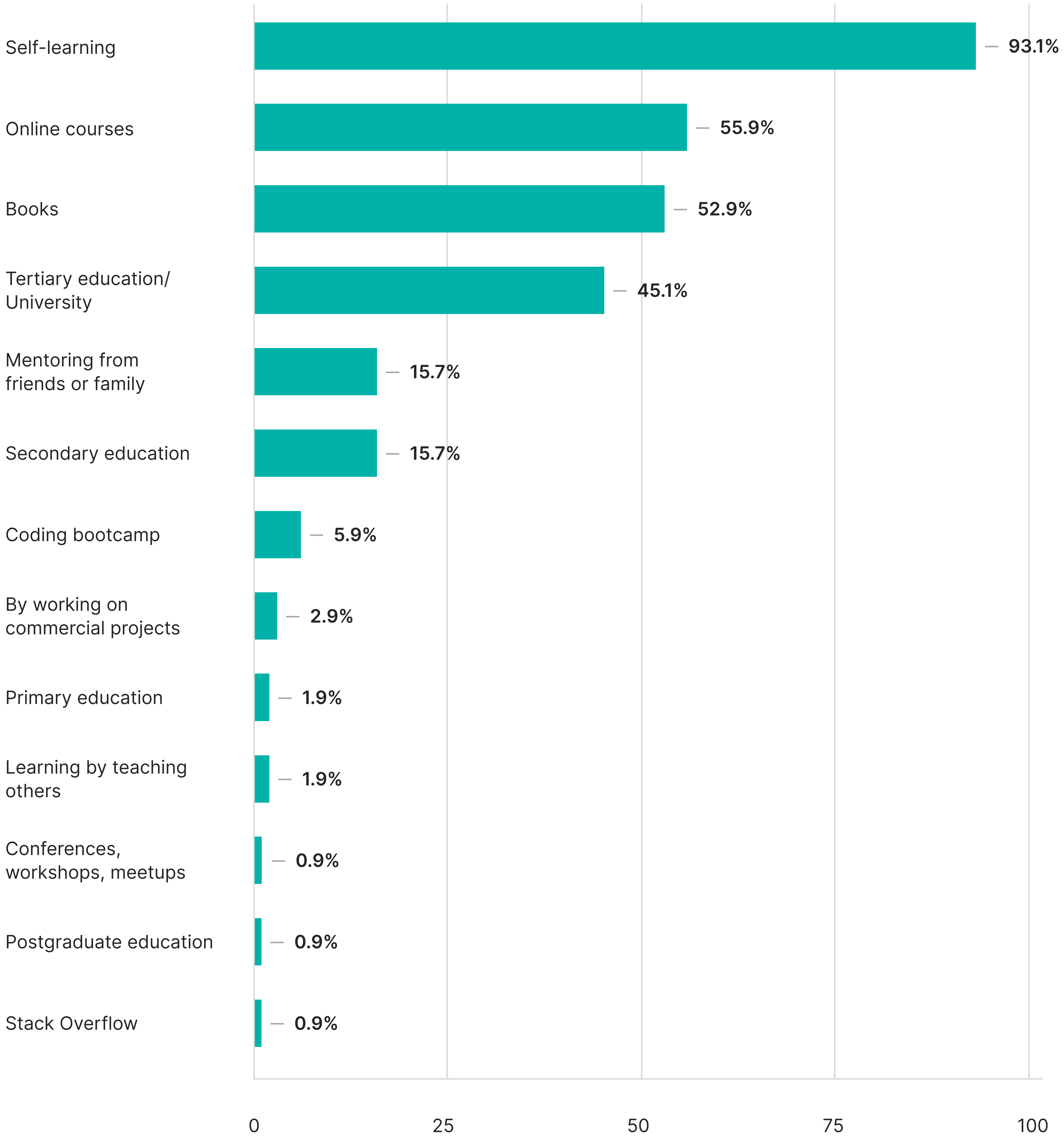
Online courses and books are also popular choices, with 55.9% and 52.9% of respondents using these resources, respectively.

For some, traditional education also played a role, with **45.1% of developers having learned to code through tertiary education or university.** Other popular methods included mentoring from friends or family and secondary education.

These results show that while there are many different paths to becoming a programmer, self-education is going to be a big part of your journey whatever path you choose—it's almost unavoidable. What's important is that the surveyed developers have the drive and determination to keep learning and growing their skills.

How did you learn programming?

(select max. 3 answers)



Python developers prioritize ML and DE in professional development

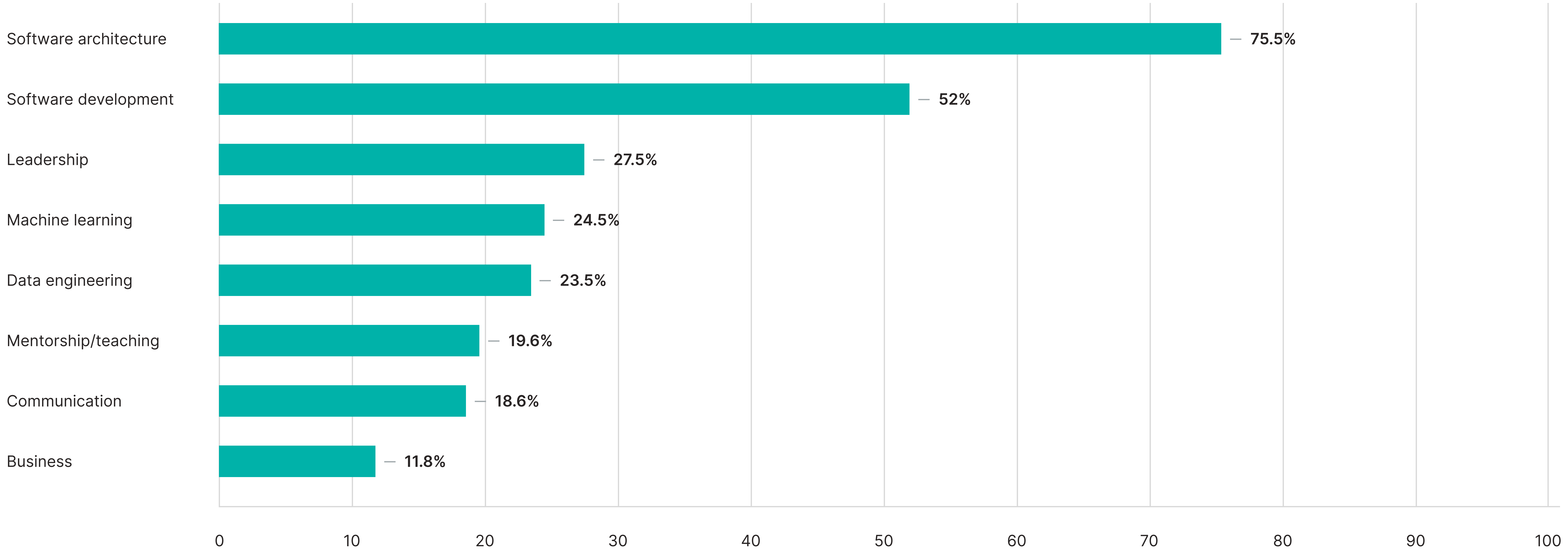
When it comes to professional development, the results of our survey show that developers are primarily focused on honing their technical skills. **A majority of respondents, 53%, indicate a desire to improve in software development**, while an even greater number, **75.5%, express a desire to improve in software architecture.**

What's important, **data engineering and machine learning also rank high on the list of desired areas for improvement, with 1 in 4 respondents expressing interest in these fields**. This reflects the growing trend in the industry towards incorporating these technologies into various applications and systems, and the surveyed developers are eager to keep up with the demands of the market.

However, it's not just technical skills that developers are seeking to improve. A considerable portion of respondents, **18.6%, aim to improve their communication skills, while others (27.5%) cite leadership and (19.6%) mentorship/teaching** as areas they would like to improve.

Which of the following areas would you like to improve the most in?

(select max. 3 answers)



Expert commentary

As the report shows, to stay ahead of the curve in the constantly advancing field of technology, developers must make a concerted effort to acquire and develop new skills, particularly in the areas of AI/ML and data engineering.

Most Python developers understand the high demand for these skills and the potential they hold for the future job market. For many, transitioning to these fields after years of experience in web development can be an exciting stage in their career path.

The survey results are encouraging, showing that many of the surveyed developers have already completed courses related to data processing. However, the journey of learning AI skills is never complete, as the field is constantly evolving with new advancements.

Investing in AI/ML and data engineering skills is a wise career move for developers and crucial for organizations in the modern business landscape. As these technologies continue to shape the way companies operate and drive innovation, having a team capable of harnessing their power will be a key differentiator for success in the industry.

Krzysztof Sopyła
Head of Machine Learning
and Data Engineering

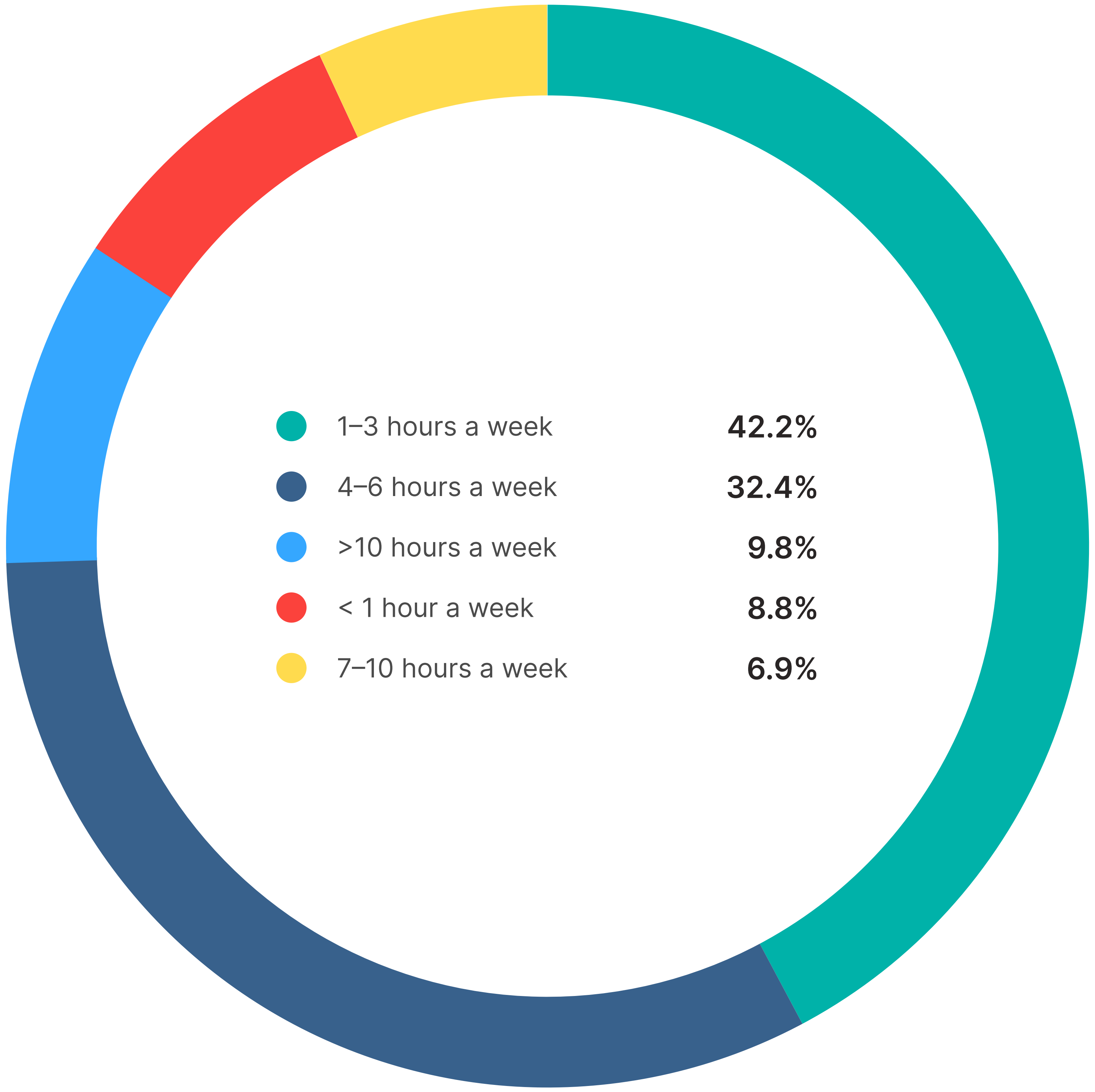
@ STX NEXT

Over 40% of developers spend more than 4 hours a week on learning

The continuous process of learning and developing skills is vital for any developer who wants to excel in their field, and our survey results show that respondents are dedicating a significant amount of their time outside of work to this pursuit.

The majority of developers **(74.6%) spend between 1–6 hours a week** on learning. **Nearly half (42.2%) dedicate 1–3 hours**, and **one-third (32.4%) of developers spend 4–6 hours** a week. Additionally, **9.8% of respondents dedicate more than a staggering 10 hours a week** to learning, which shows a strong commitment to skill advancement.

This level of dedication to learning is a testament to the importance the surveyed developers place on staying current with the latest technologies and best practices in the industry, as well as going beyond and choosing new directions.



How many hours a week do you dedicate to learning?

● 1–3 hours a week	42.2%
● 4–6 hours a week	32.4%
● >10 hours a week	9.8%
● < 1 hour a week	8.8%
● 7–10 hours a week	6.9%

Developers’ favorite training activities are hands-on and self-driven

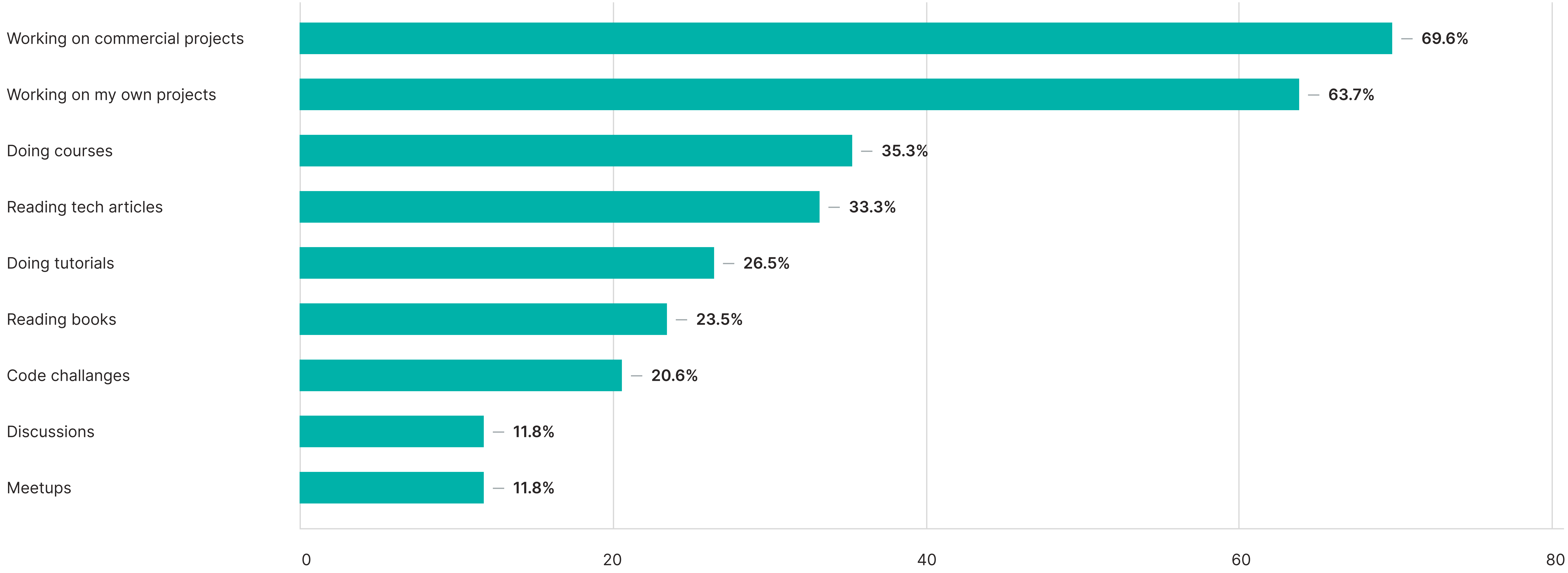
Our survey asked respondents to select their top three preferred methods for improving skills and staying current in software development. The results are clear: hands-on experience is the way to go.

An impressive 69.6% of surveyed developers choose working on commercial projects and 63.7% choose working on their own projects as one of their top 3 learning and training activities. This highlights the importance of practical, real-world experience in the software development industry.

However, around one-third of developers also choose to take courses and read tech articles, indicating that a balance of theoretical and practical knowledge may be the key to success. Other traditional methods, such as **reading books and doing code challenges are preferred by 23.5% and 20.6%, respectively.**

Which training and learning activities do you prefer?

(select max. 3 answers)



Soft skills make all the difference: challenging the stereotype of a typical Python developer

In our survey, **logical, abstract, critical and analytical thinking topped the list as the most important skill, with 32.4%** of respondents choosing it. This is closely followed by **persistence, readiness for continuous learning, and being ready for challenges, each with around 13.7%** of the vote.

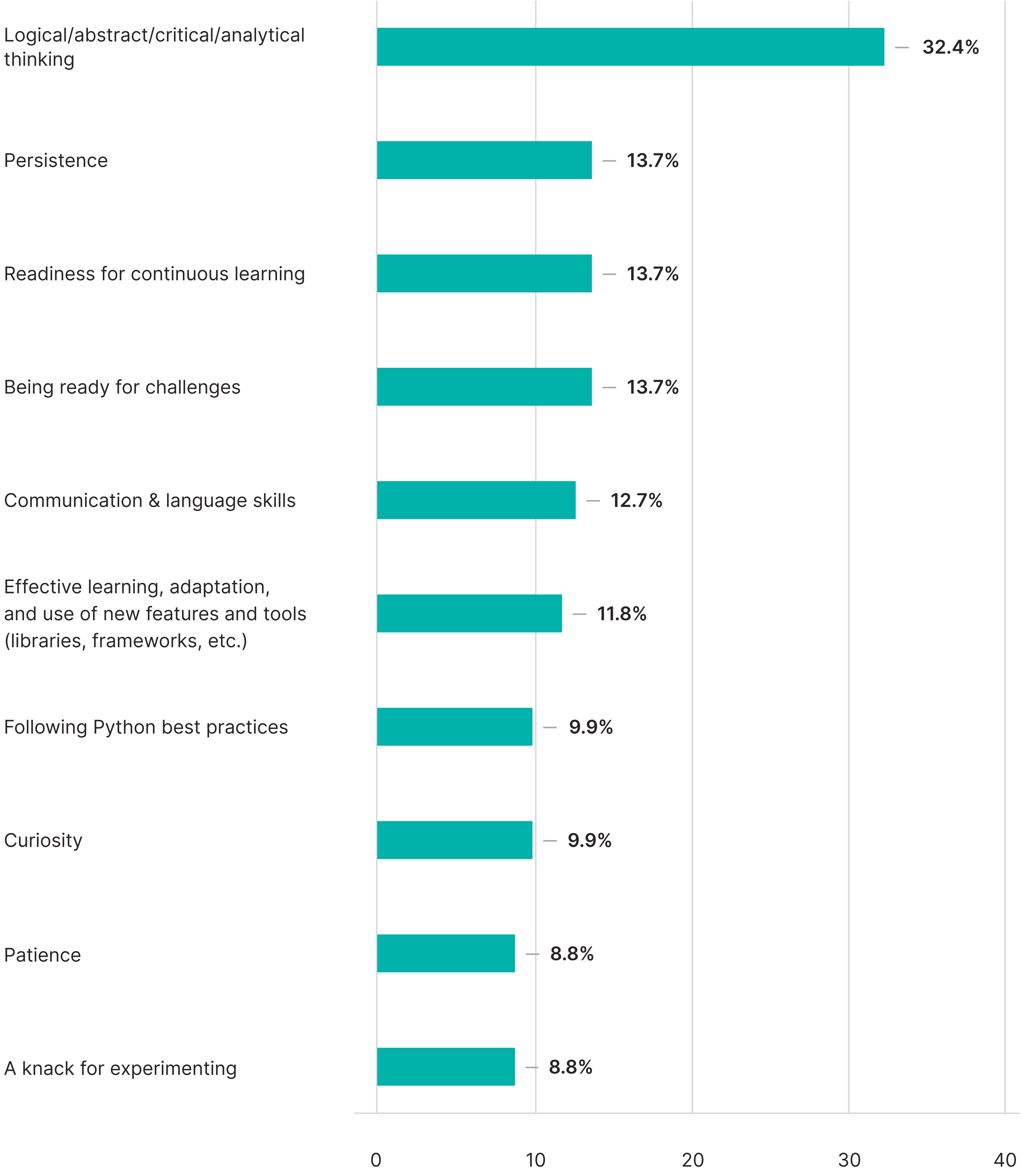
All in all, 8 out of the top 10 skills indicated as most important by our respondents could be considered soft skills, and only 2 are hard skills.

These results challenge the stereotype of a developer being someone who solely possesses technical skills. **The data suggests that soft skills, such as patience, communication, and a knack for experimenting are just as important for becoming a great Python developer.** These skills are useful for a variety of reasons, such as improving teamwork, solving complex problems, and adapting to new tools and technologies.

This indicates that the best Python developers are those who can combine technical expertise with strong problem-solving abilities and a drive to constantly improve and learn.

What skills are, in your opinion, the key to becoming a great Python developer?

(select max. 3 answers)



Expert commentary

The key is to maintain a sane balance. Imagine a Python developer who has all the cutting-edge technologies and solutions at their disposal. The urge of using all these tools can be overwhelming, but it's important to remember that they may not always align with the client's current needs. So work out a solution that will ultimately satisfy both parties.

Additionally, it's important to strive to create a work environment that fosters feedback. Publicly praise others for their hard work, provide constructive criticism, and be open to receiving feedback yourself. This will improve your work environment and encourage taking ownership of errors.

Time management and organization are also critical skills for a Python developer, especially when working remotely. Establish a routine, prioritize tasks, and set reasonable time limits. Delivering forecasts that are accurate and achievable impacts your personal credibility, therefore, while it's great to be able to deliver extra, it's important to remember that it shouldn't be at all costs.

Jakub Kołaczkowski

Senior Python Developer

@ STX NEXT

REPORT 04

Tools & technologies

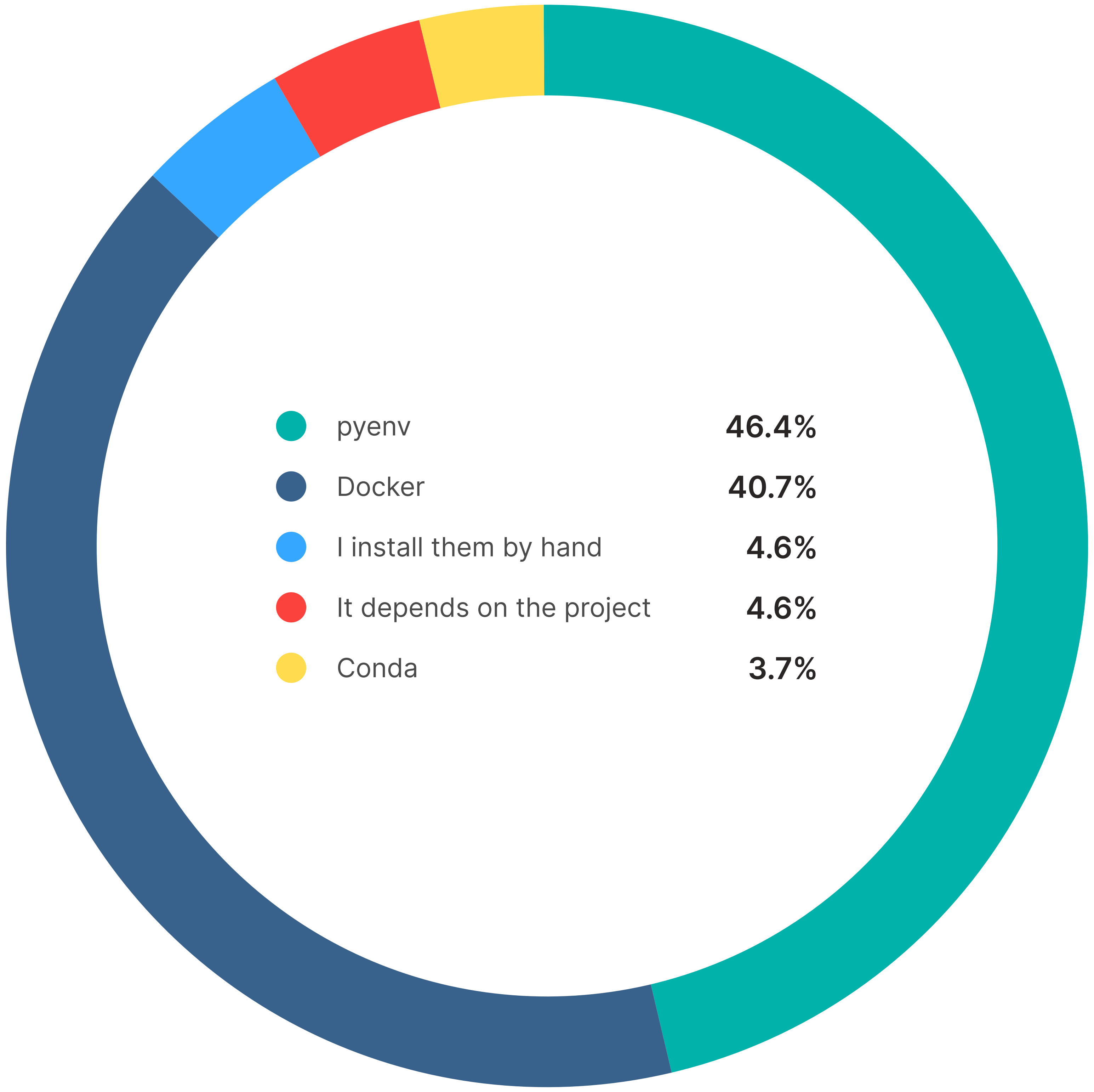
The respondents’ views
on current tech stack options

Python developers choose effortless version control

Having the ability to effectively manage different Python versions is important for Python developers. It allows them to work with the version that's most suitable for each project, without having to worry about compatibility issues. Adhering to industry standards also ensures that their projects are portable and can be easily deployed to different environments.

When it comes to our survey, **the majority of respondents use the pyenv tool, with 46.3% choosing it as their preferred method. Docker is a close second, with 40.7% of respondents using it. Installing the Python version by hand was chosen by only 4.6% of respondents, just like the option, "It depends on the project."**

The dominance of pyenv can be attributed to its ease of use and versatility, as it allows developers to easily switch between different Python versions on the same machine and manage virtual environments for each project. Docker, on the other hand, offers the ability to containerize Python applications and make them portable, which is particularly useful for deploying to different environments.



How do you manage different Python versions?

● pyenv
46.4%

● Docker
40.7%

● I install them by hand
4.6%

● It depends on the project
4.6%

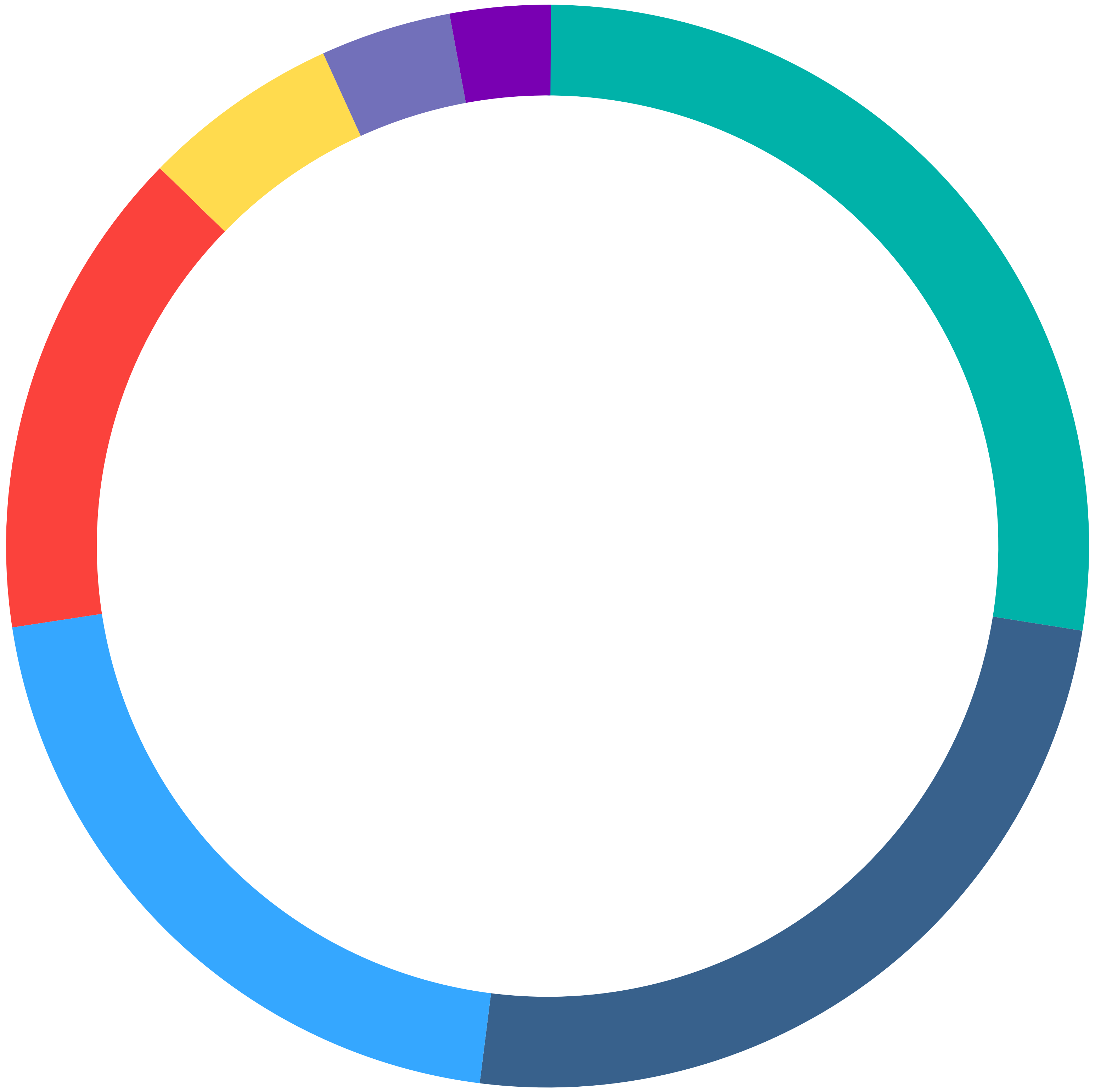
● Conda
3.7%

When it comes to maximizing the benefits of Python upgrades, balancing stability and innovation is the key

The timing of adopting a new Python release can be a tricky decision for developers, as our survey shows.

On the one hand, we have the early adopters: **27.5% of developers believe in jumping on a new release as soon as their project's CI tests start passing with it, and 14.7% wait until after the first patch release.** But the rest of our respondents take a more conservative approach: **20.6% prefer to wait for a year or longer** before making the switch. Others adopt a new release **after one newer version has been released (24.5%), after two newer versions have been released (5.9%)**, or even **until the previous version reaches End of Life (3.9%).**

This balanced distribution of votes between the two groups (42.2% early adopters vs. 57.8% with a more cautious approach) can be attributed to the fact that each project and its requirements are unique. The data also underscores the importance of careful consideration and weighing of factors such as stability, compatibility, and innovation when jumping on a new Python version.



When do you think is the right moment to jump on a new Python release?

●	As soon as CI tests for the project start passing with it	27.5%
●	When there is 1 newer version released	24.5%
●	After a year or longer	20.6%
●	After the first patch release	14.7%
●	When there are 2 newer versions relesed	5.9%
●	When the previous one reaches End of Life	3.9%
●	Other	3%

Streamlining developers' workflow: the impact of package management tools

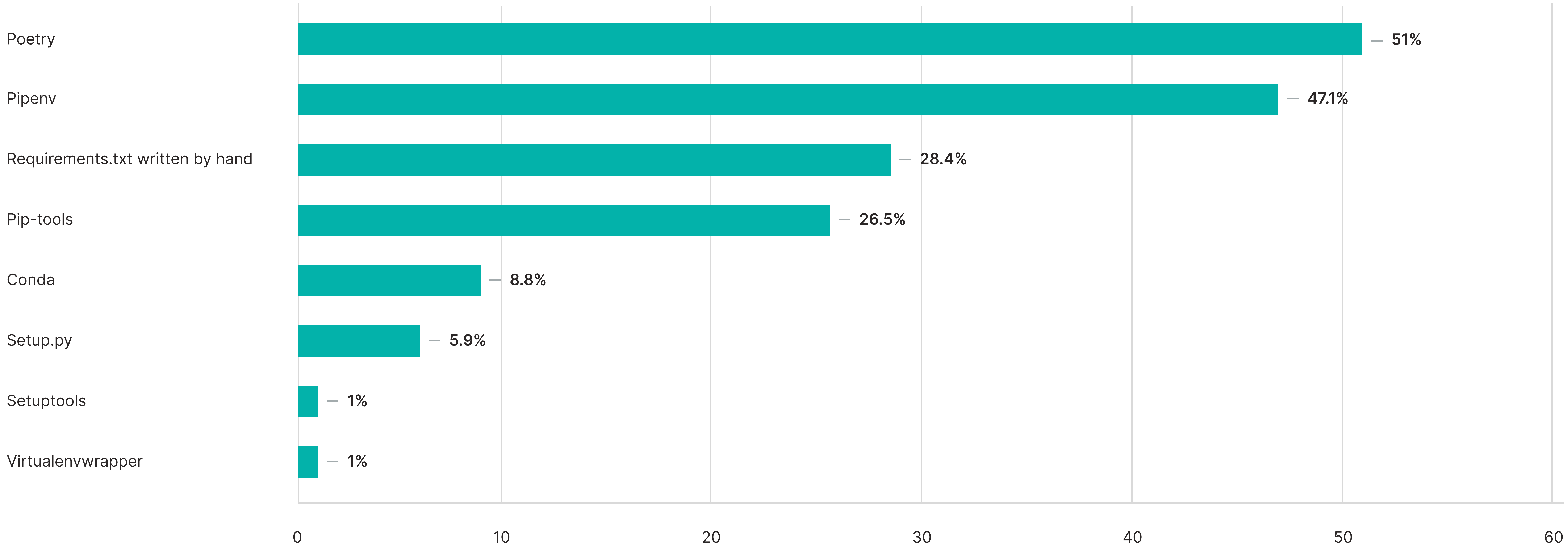
The results of our survey reveal that **poetry and pipenv are the most widely used package dependency management tools, with 51% and 47.1% of respondents, respectively.** The **other options**—including conda, pip-tools, requirements.txt written by hand, setuptools, setup.py, and virtualenvwrapper—**lag significantly behind, without reaching more than 30% popularity.**

Adhering to the industry standard of using package dependency management tools like poetry and pipenv is crucial for software development as they enable developers to work more efficiently and effectively, while also ensuring that their projects are built with the latest and greatest tools and technologies.

Poetry and pipenv offer streamlined solutions for managing packages that ensure compatibility and reduce the risk of conflicts. They're easy to use, flexible, and capable of handling even complex dependencies.
In contrast, the other options have limitations or lack the features required for modern software development practices.

Which Python package dependency management tools do you prefer using?

(select max. 3 answers)



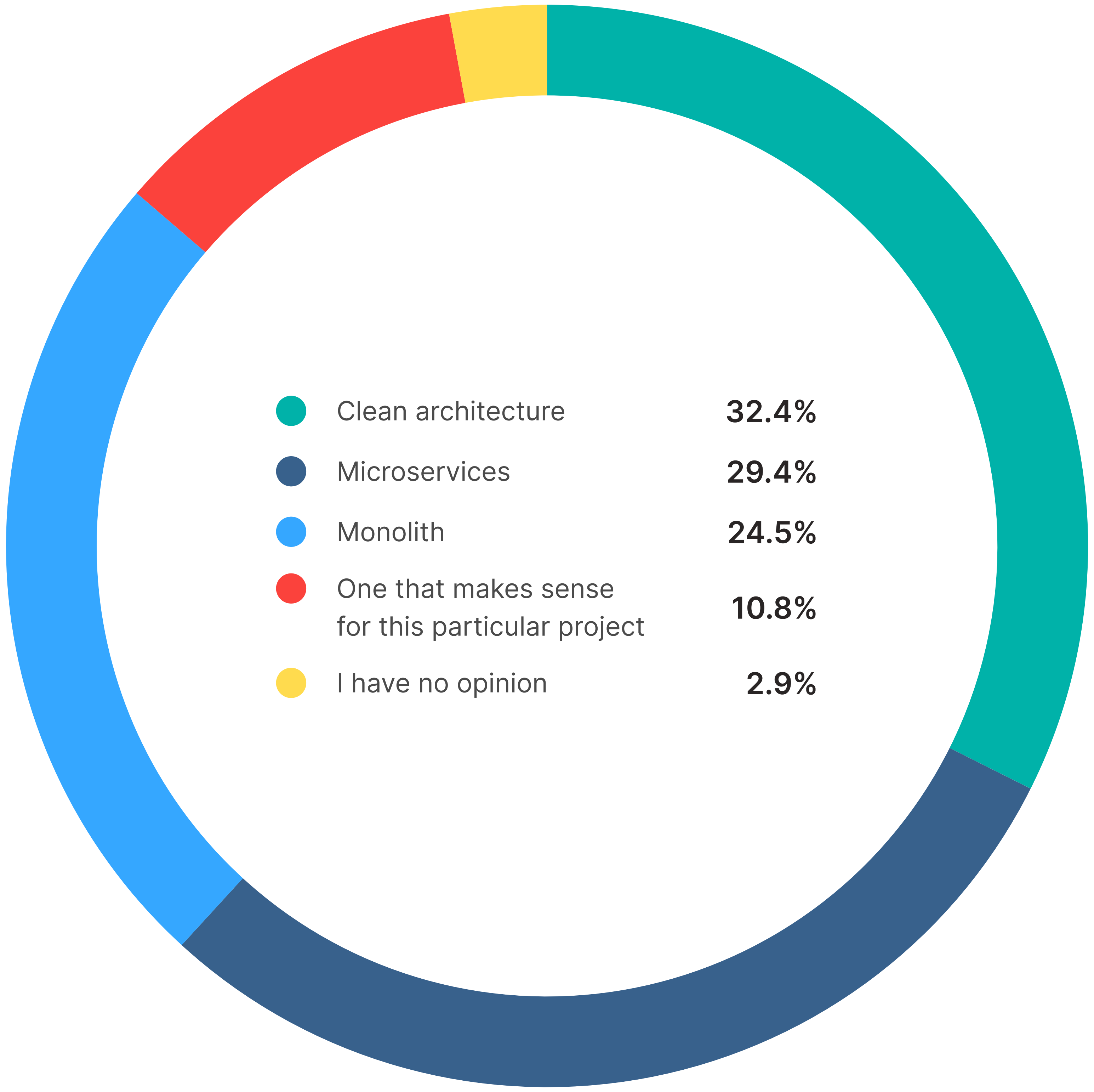
Clean architecture named the top choice for greenfield commercial projects over microservices and monolith

Architecture plays a critical role in software development, particularly in the case of greenfield commercial projects, as it sets the foundation for the entire project. A good architecture makes it easier to scale, maintain, and modify the codebase as the project evolves.

The survey data shows that **clean architecture is the most popular recommendation for greenfield commercial projects. It received 32.4% of the responses.** This option dominates the survey results because it has proven to be the most effective solution for managing complex software systems.

Microservices took second place in the survey (29.4% of responses), with monolith in third (24.5%).

What's interesting, the other options reflect a more ad hoc and less systematic approach to architecture. In some commercial projects, it may actually be wiser to take into account the specific needs of a certain project. However, our survey indicates that most developers have a favorite "go-to" architecture for new projects—though they don't necessarily agree on which one.



Which architecture do you recommend most for greenfield commercial projects?

● Clean architecture

32.4%

● Microservices

29.4%

● Monolith

24.5%

● One that makes sense for this particular project

10.8%

● I have no opinion

2.9%

Expert commentary

Yogi Berra once said, “In theory, there’s no difference between theory and practice, but in practice, there is.” I tend to agree, as personal experience often shapes our preferences.

I could say, “Choose any item from the top three answers and you’re set” if you’re unsure where to start, but a single screwdriver won’t solve all your problems. It’s important you have a variety of tools at your disposal.

As software engineers, our value lies in working with code. In practice, however, to maintain the quality of a project and ensure the continuous delivery of planned functionality, we must also invest time in continuous refactoring. This is a common theme among the top three architecture recommendations in the survey.

Adopting clean architecture, a set of guidelines that can enhance the traditional layered architecture, is a great starting point. Consider it a secret ingredient for both microservices and modular monolith approaches.

But don’t forget to continually evaluate and refine your work through refactoring. By doing so, you’ll be able to unlock your full potential as a software engineer and deliver top-quality results.

Jakub Kołaczkowski

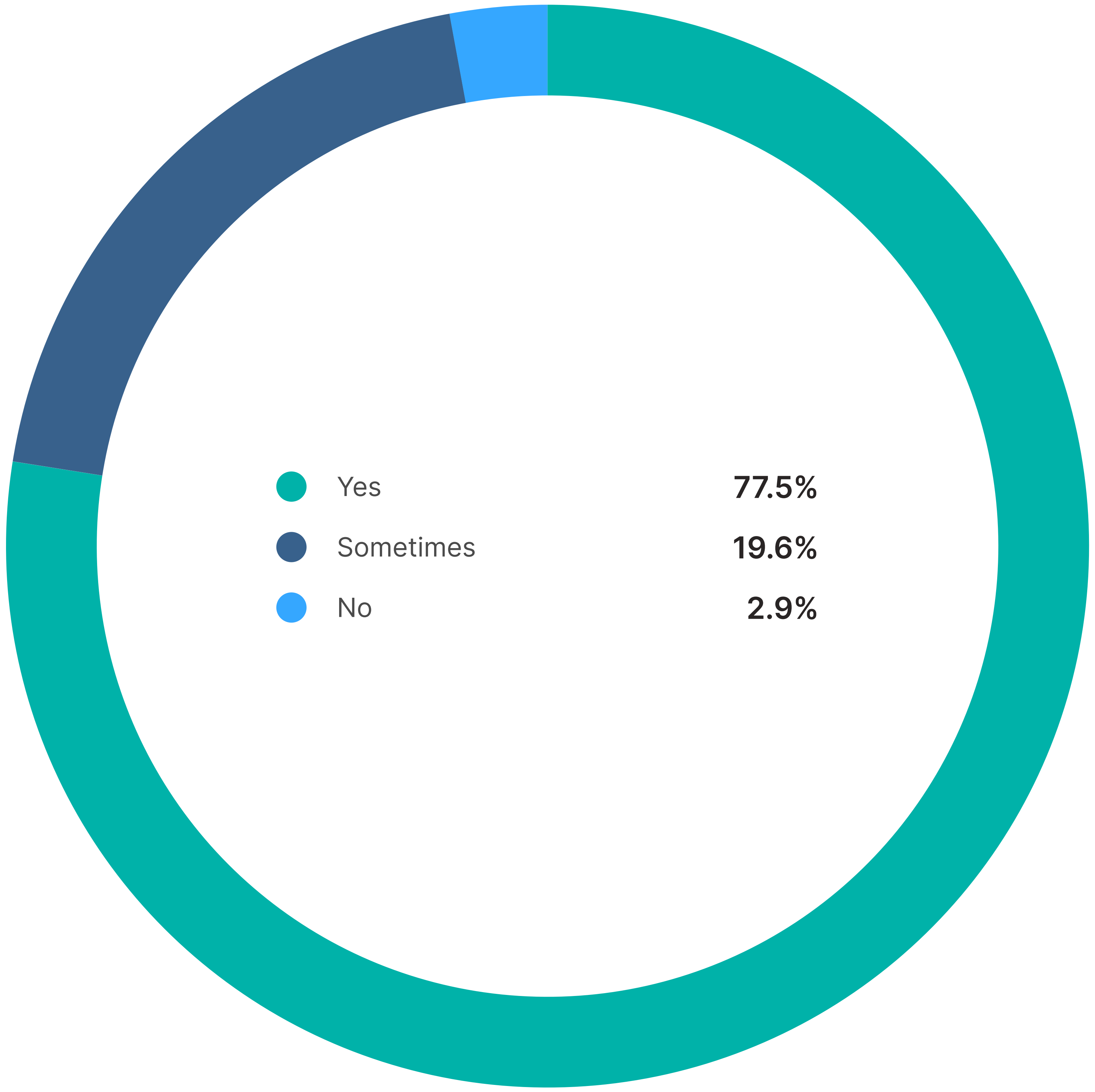
Senior Python Developer

@ STX NEXT

Type hints have become a staple in Python development

When it comes to newly started Python projects, it's clear that type hints have become an integral part of Python development. With **79% of respondents saying that they use Python type hints in their projects,** it's evident that this feature is widely adopted and valued by the Python community. This trend is especially interesting in the context of software development as a whole, where type hints are not always used.

Interestingly, **only a small percentage of respondents (2.9%) said they don't use type hints in their projects, while 19.6% use them sometimes.** This indicates that even those who don't use type hints all the time are still aware of their benefits and find them useful in certain situations.



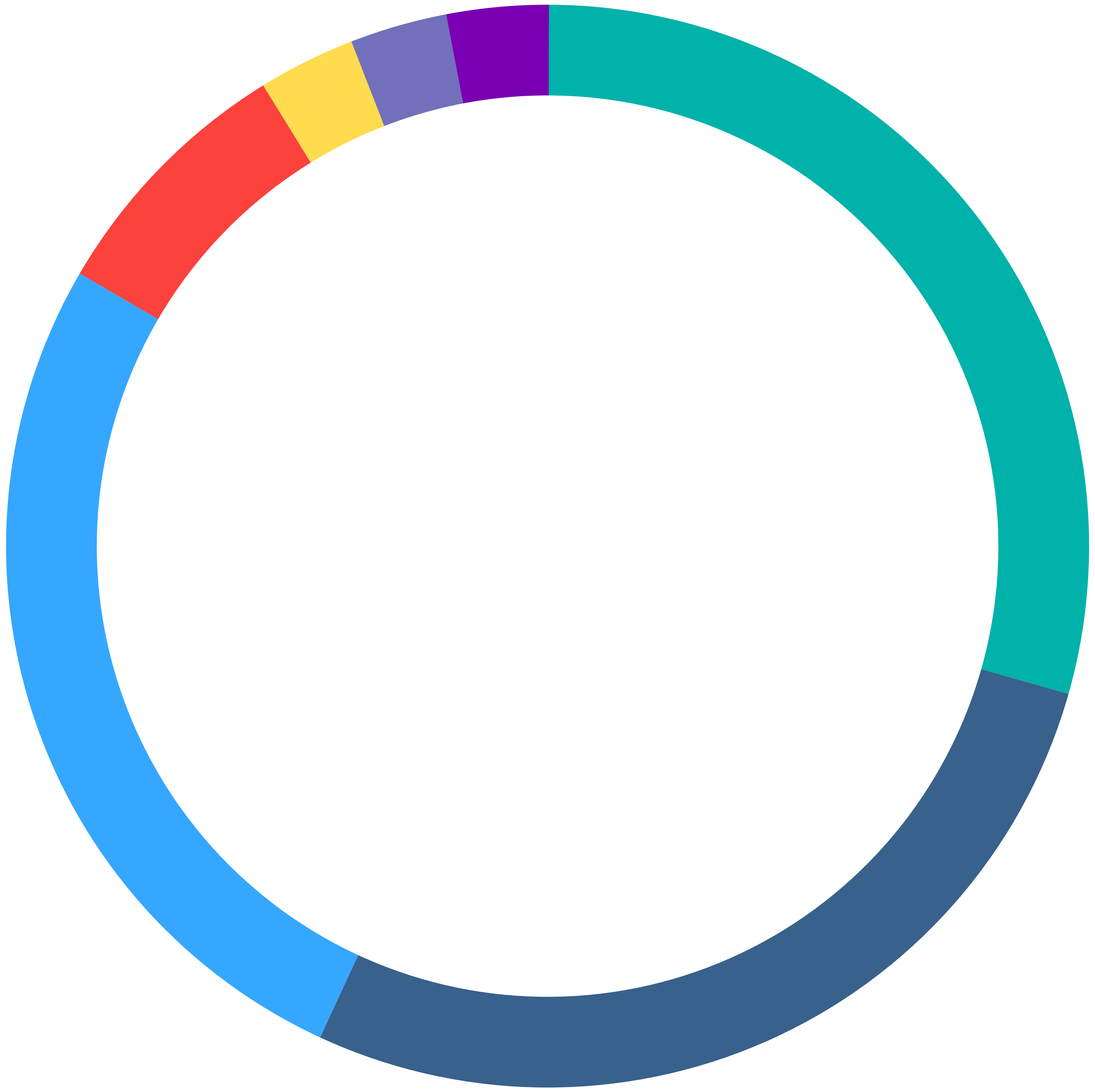
Do you use Python type hints in newly started projects?

● Yes	● Sometimes	● No
77.5%	19.6%	2.9%

Which approach tops the charts for implementing business objects? Python developers share their go-to solutions

In the world of Python development, the implementation of business objects is a crucial aspect that requires careful consideration. However, the results of our survey show that the majority of Python developers have a clear preference: **nearly 30% of respondents prefer to use the pydantic library, while just under 28% prefer to use built-in dataclasses. Only a slightly smaller number of 26.5% prefer to use regular Python classes.**

For businesses looking to employ Python developers, it's important to note that top experts should have a solid understanding of the different options available for implementing business objects and have the ability to choose the right approach based on project requirements, personal preferences, and the specific needs of the team. This versatility and informed decision-making are key to delivering high-quality software.



Which library/approach do you prefer to implement business objects?

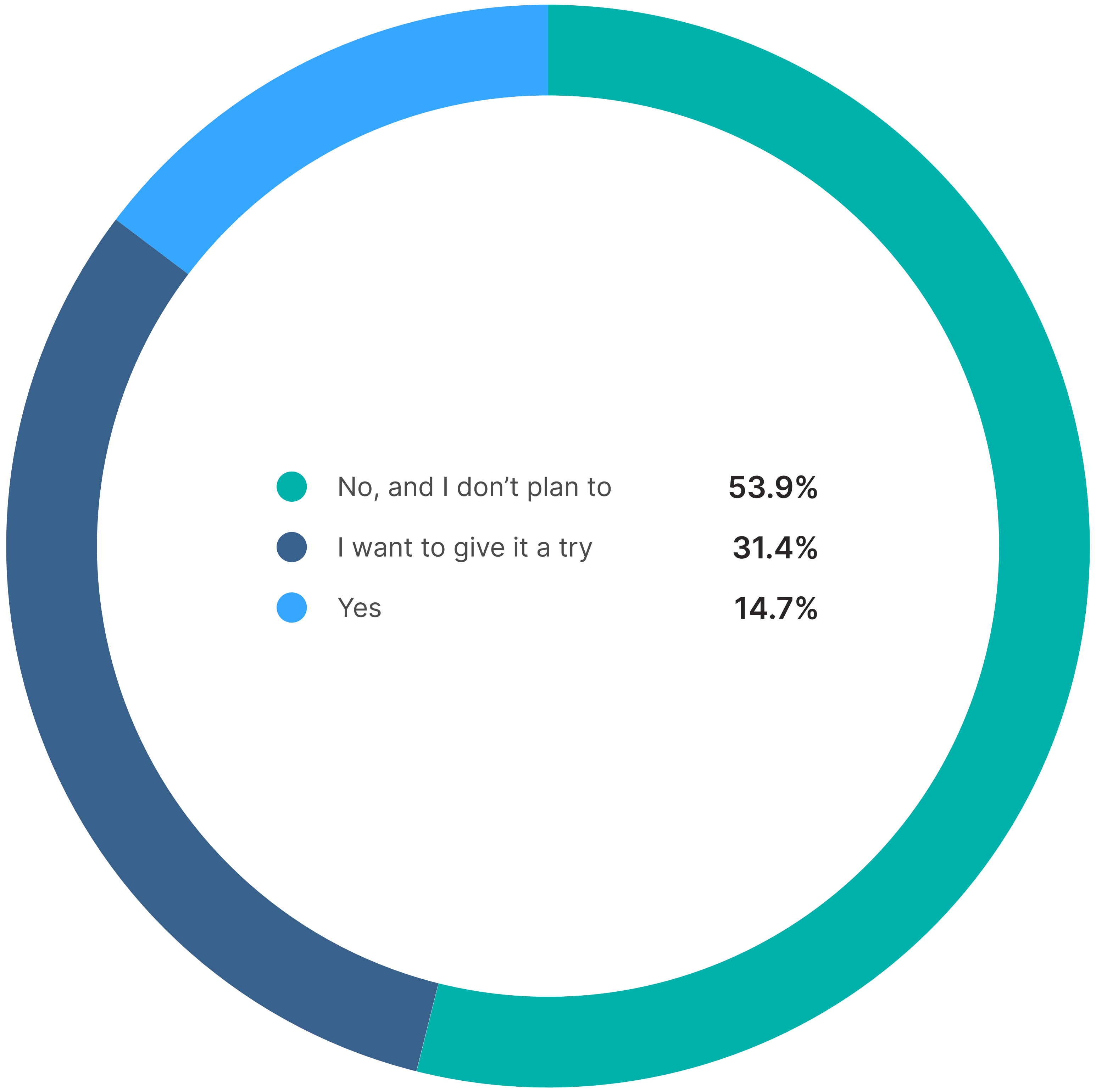
●	Pydantic	29.4%
●	Built-in dataclasses	27.5%
●	Regular Python classes	26.5%
●	When I join a project, I adopt the tools already in use	7.8%
●	Attrs	2.9%
●	Dicts	2.9%
●	Other	2.9%

Developers divided on adopting AI coding assistants

With advancements in technology, AI coding assistants are becoming increasingly popular among software developers. However, the answers reveal that at the time of our survey, only **14% of respondents were using an AI coding assistant, while 31.4% expressed interest in trying one out in the future. The majority, 53.9%, didn't use nor planned to use an AI coding assistant.**

It's possible that many developers are hesitant to use AI coding assistants because they prefer to rely on their own expertise and experience. Additionally, there may be concerns about the accuracy and reliability of such tools.

However, there are also advantages to using them, such as increased efficiency and productivity. So it's highly likely that when AI technology evolves and improves, more developers will adopt the use of AI coding assistants.



Are you using an AI coding assistant (e.g. GitHub Copilot, Tabnine)?*

● No, and I don't plan to

53.9%

● I want to give it a try

31.4%

● Yes

14.7%

*This question was asked before OpenAI introduced ChatGPT. If we asked it now, the answers might have been different.

Expert commentary

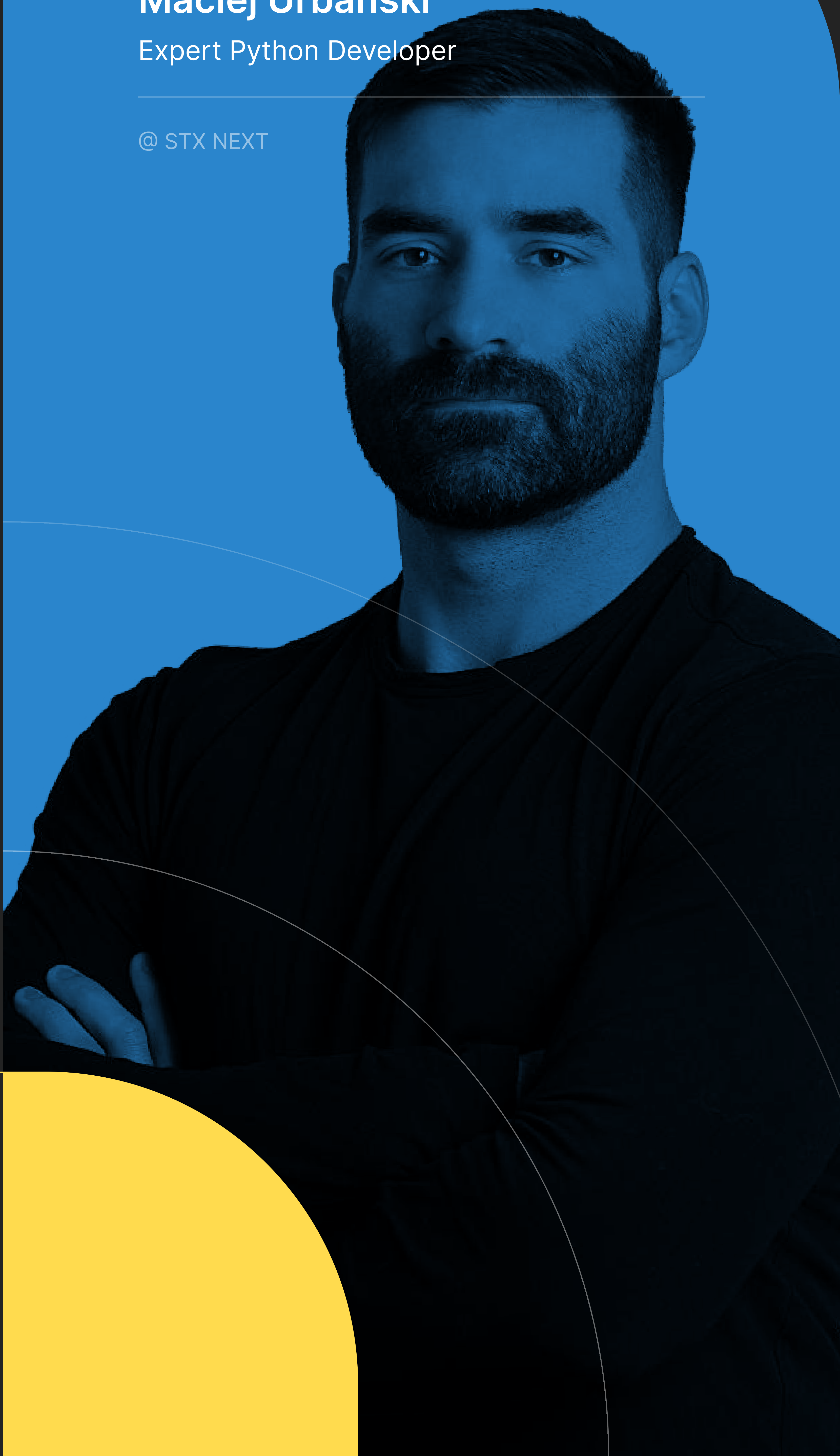
It's a running joke that programmers' job is mostly copying stuff from StackOverflow. And this is basically what these AI tools do, with all the good and bad that comes with it, but in milliseconds instead of minutes of searching, which adds up over time. The questions to ask are, "How much of a developer's work can AI do?" and "Is it safe?"—not "Is it helpful?"

However, developers don't give their trust easily. Github Copilot and similar tools have been under fire for allegedly selling something that's in public domain. The argument is that since it was trained on open-source data, it should be open source, as well. There's also a question of whether sending code of the project you're currently working on to a cloud provider is okay with the project owner.

My gut feeling is that since coding is all about reducing time needed to deliver features to market, using such tools won't be "optional" in the future.

Maciej Urbański
Expert Python Developer

@ STX NEXT



REPORT 05

What's NEXT?

The future of Python

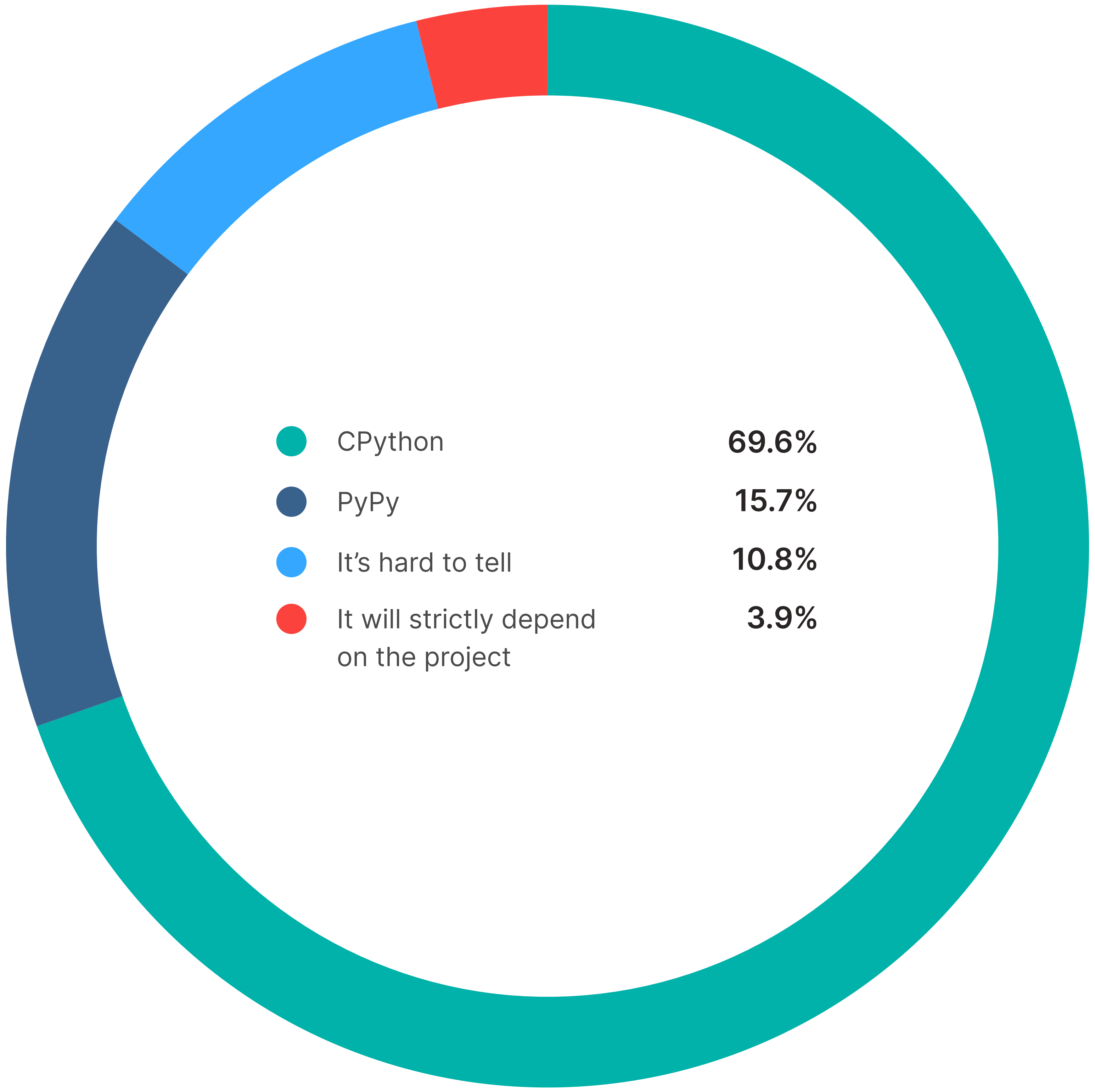
CPython dominates as the future's top Python implementation

When it comes to the future of Python, opinions may vary, but the data speaks for itself. **In our survey, 69.6% of respondents believe that CPython will be the best Python implementation in the future, while 15.7% think it will be PyPy. Even fewer (10.8%) feel it's hard to tell or it will strictly depend on the project (3.9%).**

It's clear that CPython has a strong following among the Python community and is considered to be the standard implementation of the language. With its robust feature set and well-established ecosystem, it's no wonder that so many developers see it as the top choice for future projects.

However, PyPy's ability to offer faster performance for certain use cases has also caught the attention of some developers.

As the Python landscape continues to evolve, it will be interesting to see how these trends develop and how developers will choose the best Python implementation for their projects in the future.



Which Python implementation will be the best choice in the future?

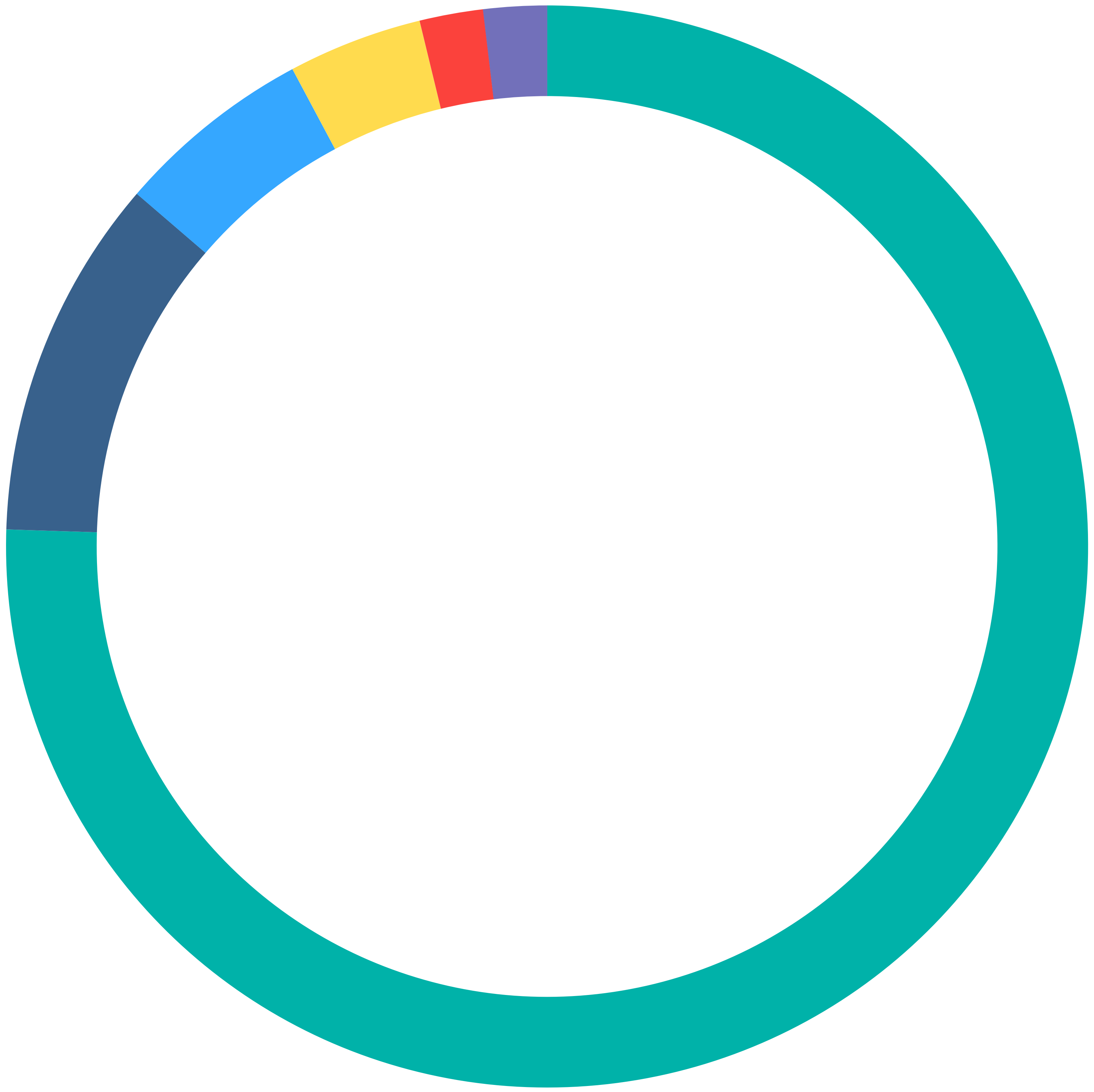
● CPython	● It will strictly depend on the project	● PyPy	● It's hard to tell
69.6%	3.9%	15.7%	10.8%

AWS is the #1 cloud platform for expert Python developers

The cloud platform landscape has become an essential part of modern software development. It’s revolutionizing the way businesses operate and deliver services to their customers as it provides unprecedented access to computing power, storage, and services critical for businesses to succeed in today’s digital economy.

Our survey results show that the top cloud platform of choice among Python developers is Amazon Web Services (AWS) with an overwhelming 75.5% of votes. Google Cloud Platform comes in second with 10.8% of the vote.

Having said that, with new technologies and innovations constantly emerging in the field, it remains to be seen what the future holds for cloud computing and which platform will reign supreme.



Which cloud platform will be the go-to choice?

●	AWS	75.5%
●	Google Cloud Platform	10.8%
●	It’s hard to tell	5.9%
●	In most cases it will be depend on the client	4%
●	Azure	1.9%
●	Dedicated server if it’s possible	1.9%

Python developers predict Trello and Zoom will fall out of favor

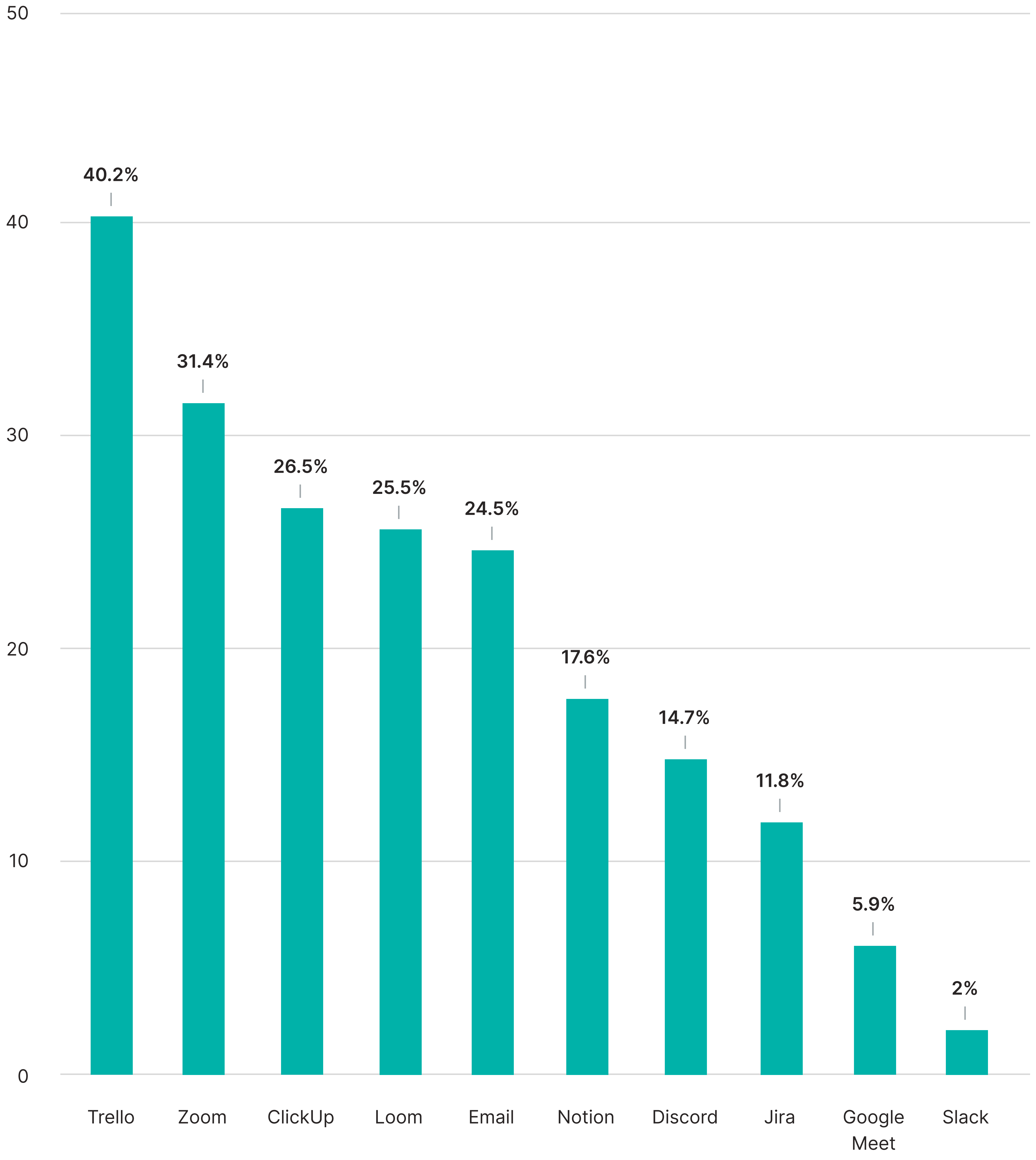
We asked developers about their thoughts on the future of various synchronous or asynchronous communication tools. Specifically, we asked which of them are bound to become outdated soon.

The most votes went to **Trello, with 40.2% of developers predicting it will soon become outdated.** Zoom was a close second, with 31.4% of votes. Other tools like ClickUp (26.5%), Loom (25.5%), and email (24.5%) also received a significant number of votes.

It's important to note that tools like **Slack, Google Meet, and Jira received relatively low votes, suggesting that they're still considered relevant and valuable tools**. If a developer uses such solutions, it means that they're using the tools that are considered the most effective, efficient, and up-to-date for communication and collaboration within a project.

Which synchronous/asynchronous communication tools will soon be out of date?

(select max. 3 answers)



Expert commentary

As developers, we like to be lazy. We opt for tools that we know, that are proven to work, and that new team members can quickly master.

Standardization is happening in many areas, including the methods and tools of communication or in the cloud platforms we use. This standardization helps us create cheaper and faster, reliable solutions. It allows us to focus on what is important.

Less useful tools are being abandoned. Natural selection is under way. I just wish that the practical solutions from the abandoned tools would be implemented in the ones that dominate the market.

However, I would like to see new tools and platforms appear in this ecosystem once in a while, to put pressure on the established ones. Otherwise, it will be difficult to make progress.

The tools will get better and better. Our work more and more efficient.
But email communication will stay forever.

Marek Bryling
Expert Software Engineer

@ STX NEXT

AI’s impact on software development revealed

Our survey asked developers about their thoughts on the extent to which AI will automate different aspects of the software development process, and the results were intriguing.

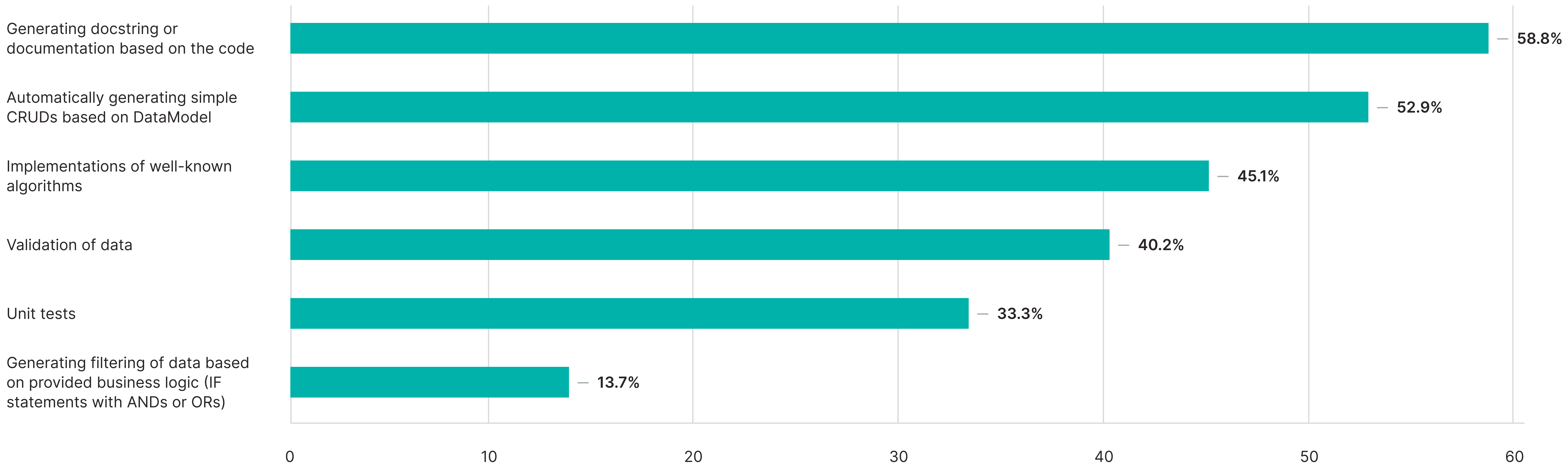
The top vote-getter is **“Generating docstring or documentation based on the code,” with 58.8%** of developers believing it will be automated by AI. **“Automatically generating simple CRUDs based on a data model” isn’t far behind, with 52.9%** of votes.

Implementation of well-known algorithms (45.1%), validation of data (40.2%), and unit tests (33.3%) also received a lot of votes. However, a more complex task such as **generating filtering of data based on business logic received relatively low votes, with only 13.7%** of developers predicting it will be automated by AI.

These results highlight the potential for AI to automate certain parts of the development process, but also the limitations it currently faces.

How much development (coding etc.) do you think will be automated through AI?

(select max. 3 answers)

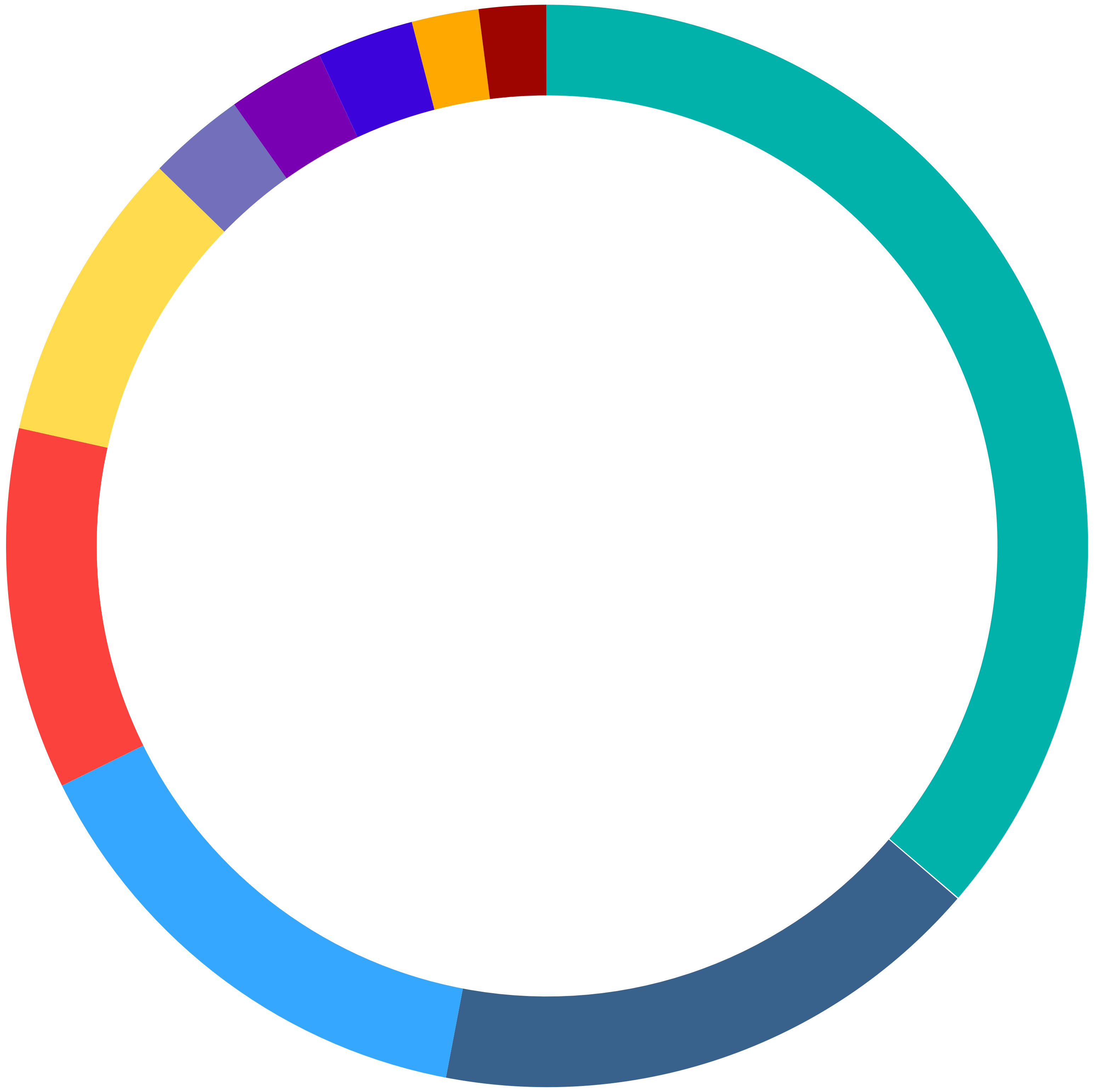


Future discussions about Python will likely focus on AI, performance, and GIL removal—but there's also lots of uncertainty

Predicting the future is tricky, especially when it comes to determining the next hot topic in the world of Python. **A substantial 36.3% of our survey respondents confessed to feeling curious about the next big thing in Python but uncertain about what that thing might be.**

However, despite the uncertainty, a few potential hot topics emerged from the responses. **Almost 1 in 5 developers (16.7%) believe that artificial intelligence, machine learning, data engineering, and data science** will be the next big thing in Python. Another **14.7% of developers are betting the focus will be on performance.** Of the remaining respondents, 8.8% think that GIL removal will be the next big thing, while smaller percentages believe that it will be PyScript (2.9%), typing (2.9%), security (2.9%), asyncio (2.0%), or FastAPI (2.0%).

So, while the future of Python remains a mystery, one thing is for sure: the developers who use Python are eager to see what's next. Whether it's AI and machine learning, performance improvements, or something entirely new, the Python community is poised for exciting developments in the coming years.



What will be next Python-related hot topic?

●	It's difficult to say, but I'm curious	36.3%
●	ML/AI/DE/DS	16.7%
●	Performance	14.7%
●	Other	10.8%
●	GIL removal	8.8%
●	PyScript	2.9%
●	Security	2.9%
●	Typing	2.9%
●	Asyncio	2%
●	FastAPI	2%

Expert commentary

The Python ecosystem is constantly evolving, and the survey results reveal a few exciting trends worth keeping an eye on.

Firstly, the interest in ML and AI is skyrocketing. Developers show a keen desire to harness the power of artificial intelligence and new, easy-to-use libraries to train, monitor and deploy machine learning models.

Developers are also looking for ways to make their code run faster, which is why performance optimization is another hot topic.

Another area that’s starting to gain traction is the removal of GIL in Python, which has long been a source of frustration for many developers. Removal of GIL can lead to a significant boost in the performance of multi-threaded Python applications.

Overall, the Python ecosystem is a hive of activity, and it’s a perfect time to be a part of it. With many exciting developments on the horizon, the future of Python looks bright.

Krzysztof Sopyła

Head of Machine Learning
and Data Engineering

@ STX NEXT

Nearly a third of Pythonistas expect performance improvements in the future

Everybody knows that "prediction is difficult, especially when it comes to the future." Is it the same for the evolution of programming languages, particularly Python? The results of our survey suggest that it might be. **The second most common response (26.5%) to the question of what changes are expected in Python and its ecosystem was, "It's difficult to say, but I'm curious."** This answer is not surprising in such a rapidly developing industry like software.

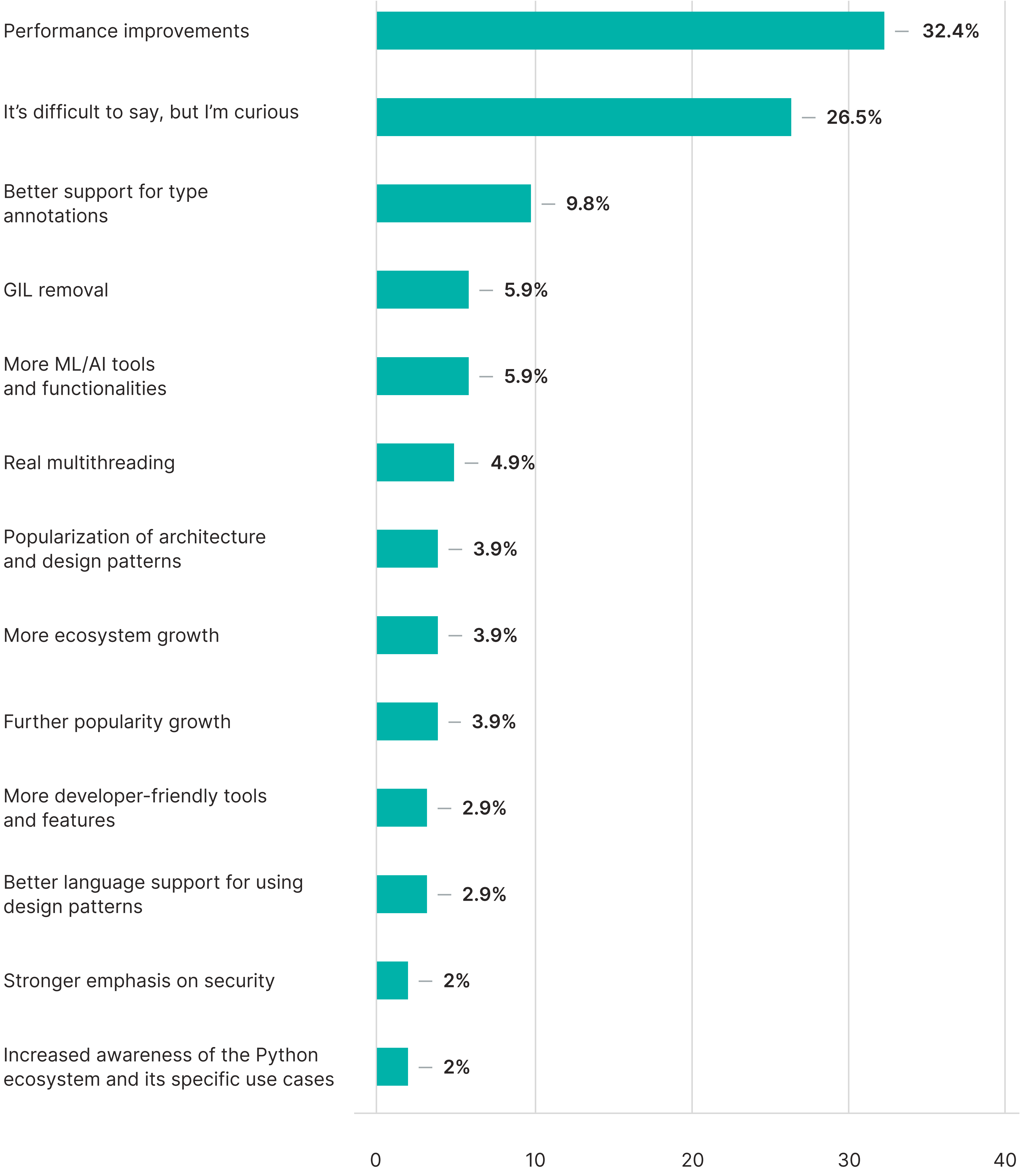
That being said, **most of our programmers (33%) expect that future Python development will focus on performance improvements.**

Type annotations received a nod of support from 9.8% of those surveyed, while ML/AI tools and functionalities, GIL removal, and real multithreading were each noted by around 5–6% of participants.

But the future of Python isn't just about technical advancements—there are also expectations for the ecosystem to grow and become more accessible to developers. 4% of our respondents see further popularity growth, while others believe that architecture and design patterns will become more popular (3.9%) or that the language will better support these patterns (2.9%).

What do you expect to change in the area of Python in the future?

(select max. 3 answers)



Expert commentary

Performance has always been considered a weakness rather than a strength of Python. Despite this, the language and its ecosystem have evolved in ways that allow us to overcome the limitations of its interpreted nature.

For example, the use of C extensions, which is common in resource-intensive libraries, provides both performance and ease of use. However, these solutions don't work for every case. The volumes of data being handled by systems today are higher than ever, and the world is starting to recognize the cost of running hundreds of servers for both businesses and the environment. In this context, performance is increasingly important. Fortunately, both the direction of Python's development and the results of our survey results suggest that our community is aware of this.

The speed-up in Python 3.11 is particularly noteworthy. Compared to its predecessor, the newest version is up to 60% faster, with an average improvement of 25%. These past changes indicate that this is just the next step in Python's journey toward improvement. We now have a powerful yet flexible system for type annotations that has already bridged the gap between Python and statically typed languages. It's also important that library and community adoption are keeping pace with this evolution. Providing type hints is becoming standard practice in both commercial projects and open-source libraries.

It's difficult to predict how any technology will change in the coming years. Nonetheless, I'm sure that future changes in Python will exceed our expectations.

Mikołaj Lewandowski
Expert Python Developer

@ STX NEXT

Expert commentary

The current global tech landscape paints a very clear picture: Python has established itself as a critical language for businesses and organizations of all sizes, from startups to enterprises.

The demand for Python development skills is constantly growing, driven by the increasing importance of data science, machine learning, and artificial intelligence. Businesses are now recognizing the power of these technologies, and are investing in teams and resources to develop their capabilities. What's more, the ease of use and versatility of Python make it an ideal choice for these complex, data-intensive projects.

Overall, I'm confident that Python will remain a dominant player in the global business landscape, and that its importance will only continue to grow. Businesses that invest in Python and build their development capabilities will be well-positioned for success in the years to come.

Ronald Binkofski

CEO

@ STX NEXT

About STX Next

Launched in 2005, STX Next has grown into Europe's leading software development consulting company specializing in Python.

With a passion for delivering the best results, we're dedicated to helping our clients achieve their goals by providing innovative technology and exceptional service. We work with businesses and enterprises on custom-made, top-quality digital products, supporting them every step of the way.

STX Next boasts a team of 400+ developers with extensive experience in **Python, JavaScript, .NET, and React Native.** We use this expertise to create bespoke solutions that meet our clients' unique needs. Whether it's a web application, a mobile app, or a data processing system—we have the skills to deliver.

Our developers are supported by a network of **Scrum Masters, Product Owners, DevOps Engineers, Machine Learning and Data Engineering Specialists, Service Delivery Managers, and Product Designers.**

Due to the company's dynamic expansion in the last 12 months, including opening a branch office in Mérida, Mexico in 2022, we now have an active presence in South America, North America, and Europe—particularly the US, the UK, and the DACH region.

To partner up with STX Next on your next project, visit stxnext.com or reach out to us at business@stxnext.com!

550+

professionals on board

300+

clients served

18+

years of market experience

200+

Python developers

3.5+

years' average partnership

6.5

years' average experience of our Python developers

400+

developers

200+

projects

TOP
MACHINE
LEARNING COMPANY
Clutch
2023
TOP
NODEJS
DEVELOPERS
Clutch
2023
TOP
PYTHON &
DJANGO DEVELOPERS
Clutch
2023
TOP
SOFTWARE
DEVELOPERS
Clutch
2023
TOP
SOFTWARE
DEVELOPERS
Clutch
FINANCIAL SERVICES
2023
TOP
WEB
DEVELOPERS
Clutch
2023

STX NEXT

REVSYS

codeanywhere

Python Academy
Training & Consulting