



Python Types Intro

```
def get_full_name(first_name, last_name):  
    full_name = first_name.title() + " " + last_name.title()  
    return full_name  
  
print(get_full_name("john", "doe"))
```

```
1 def get_full_name(first_name, last_name):  
2 | full_name = first_name. You, a few seconds ago • Uncommitted changes
```

- def
- first_name
- full_name
- get_full_name
- last_name

Add types

```
first_name, last_name
```

```
first_name: str, last_name: str
```

type hints

```
def get_full_name(first_name: str, last_name: str):  
    full_name = first_name.title() + " " + last_name.title()  
    return full_name  
  
print(get_full_name("john", "doe"))
```

این تایپ پیش فرض نیست!

```
first_name="john", last_name="doe"
```

: مثل = نیست.

```
1 def get_full_name(first_name: str, last_name: str):
```

```
2     full_name = first_name.
```

You, a few seconds ago • Uncommitted changes

- ★ format
- ★ join
- ★ split
- ★ encode
- capitalize
- casefold
- center
- count
- endswith
- expandtabs
- find
- format map

`str.format(self, *args, **kwargs)` ×

`S.format(args, *kwargs) -> str`

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').


```
1 def get_full_name(first_name: str, last_name: str):
```

```
2     full_name = first_name.
```

You, a few seconds ago • Uncommitted changes

- rfind
- rindex
- rjust
- rpartition
- rsplit
- rstrip
- splitlines
- startswith
- strip
- swapcase
- title**
- translate

`str.title(self)` ×

S.title() -> str

Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.

فواید دیگر؟

```
def get_name_with_age(name: str, age: int):  
    name_with_age = name + " is this old: " + age  
    return name_with_age
```

```
1 def get_name_with_age(name: str, age: int):
2
3     [mypy] Unsupported operand types for + ("str" and "int")
4     [error]
5     name_with_age = name + " is this old: " + age
6     return name_with_age
7
```

```
def get_name_with_age(name: str, age: int):  
    name_with_age = name + " is this old: " + str(age)  
    return name_with_age
```

می‌توان همه‌ی تایپ‌های پایتون را اضافه کرد.

- `int`
- `float`
- `bool`
- `bytes`

```
def get_items(item_a: str, item_b: int, item_c: float, item_d: bool, item_e: bytes):  
    return item_a, item_b, item_c, item_d, item_d, item_e
```

Generic types with type parameters [Python 3.9+](#)

```
def process_items(items: list[str]):  
    for item in items:  
        print(item)
```

```
1 from typing import List
2
3 def process_items(items: List[str]):
4     for item in items:
5         print(item.)
6
```

- capitalize
- casefold
- center
- count
- encode
- endswith
- expandtabs
- find
- format
- format_map
- index
- isalnum

`str.capitalize(self)` ×

`S.capitalize()` -> str

Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case.

```
def process_items(items_t: tuple[int, int, str], items_s: set[bytes]):  
    return items_t, items_s
```



```
def process_items(prices: dict[str, float]):  
    for item_name, item_price in prices.items():  
        print(item_name)  
        print(item_price)
```

Union

```
def process_item(item: int | str):  
    print(item)
```

int or a str

Possibly None

```
from typing import Optional

def say_hi(name: Optional[str] = None):
    if name is not None:
        print(f"Hey {name}!")
    else:
        print("Hello World")
```

Classes as types

```
class Person:  
    def __init__(self, name: str):  
        self.name = name  
  
def get_person_name(one_person: Person):  
    return one_person.name
```

```
1 class Person:
2     def __init__(self, name: str):
3         self.name = name
4
5
6 def get_person_name(one_person: Person):
7     return one_person.
8
```

- ★ name
- next
- __bases__
- __class__
- __delattr__
- __dir__
- __doc__
- __eq__
- __format__
- __ge__
- __getattribute__
- __gt__

str

×

Pydantic models

کتابخانه‌ای برای ارزیابی داده‌ها در پایتون

<https://docs.pydantic.dev/>

```
from datetime import datetime
```

```
from pydantic import BaseModel
```

```
class User(BaseModel):
```

```
    id: int
```

```
    name: str = "John Doe"
```

```
    signup_ts: datetime | None = None
```

```
    friends: list[int] = []
```

```
external_data = {
```

```
    "id": "123",
```

```
    "signup_ts": "2017-06-01 12:22",
```

```
    "friends": [1, "2", b"3"],
```

```
}
```

```
user = User(**external_data)
```

```
print(user)
```

```
# > User id=123 name='John Doe' signup_ts=datetime.datetime(2017, 6, 1, 12, 22) friends=[1, 2, 3]
```

```
print(user.id)
```

```
# > 123
```

Type Hints with Metadata Annotations



```
from typing import Annotated
```

```
def say_hello(name: Annotated[str, "this is just metadata"]) -> str:  
    return f"Hello {name}"
```

خود پایتون کاری با حاشیه متادیتا نداره منتها جایی هست که بعدا می‌توان ازش برای کنترل رفتار اپلیکیشن استفاده کرد.

Concurrency and async / await

نسخه‌های مدرن پایتون از **asynchronous** (کد ناهمزمان) با استفاده از چیزی به نام «کوروتین» با سینتکس **async** و **await** پشتیبانی می‌کنند.

کد ناهمزمان فقط به این معنی است که زبان  راهی دارد که به کامپیوتر / برنامه بگوید که در نقطه ای از کد، باید منتظر چیز دیگری باشد تا در جای دیگری تمام شود. 

فرض کنید کد با یک `slow_file` مواجه شده است.

تا زمانی که این `slow_file` کارش تمام شود برنامه (کد) به کارهای دیگرش می‌رسد.

بعد هر موقع کار `slow_file` به انجام رسید به ادامه‌ی کار روی آن می‌پردازد.

مثل آشپزی!

معمولا منظور از `slow_file` قسمت‌های I/O می‌باشد که عموما از کارهای محاسباتی با CPU و RAM کندتر است.

مثال‌ها:

- داده های کلاینت که از طریق شبکه ارسال می شود
- داده های ارسال شده توسط برنامه شما برای دریافت توسط کلاینت از طریق شبکه
- محتویات یک فایل در دیسک توسط سیستم خوانده شود و به برنامه شما داده شود
- محتویاتی که برنامه شما به سیستم داده تا روی دیسک نوشته شود
- یک عملیات API از راه دور
- یک عملیات پایگاه داده برای اتمام
- یک کوئری پایگاه داده برای برگرداندن نتایج
- و غیره

چرا Asynchronous؟

به این خاطر که برنامه ما با عملیات کند سینک نیست و منتظر نمیشینه که عملیات کند تموم بشه

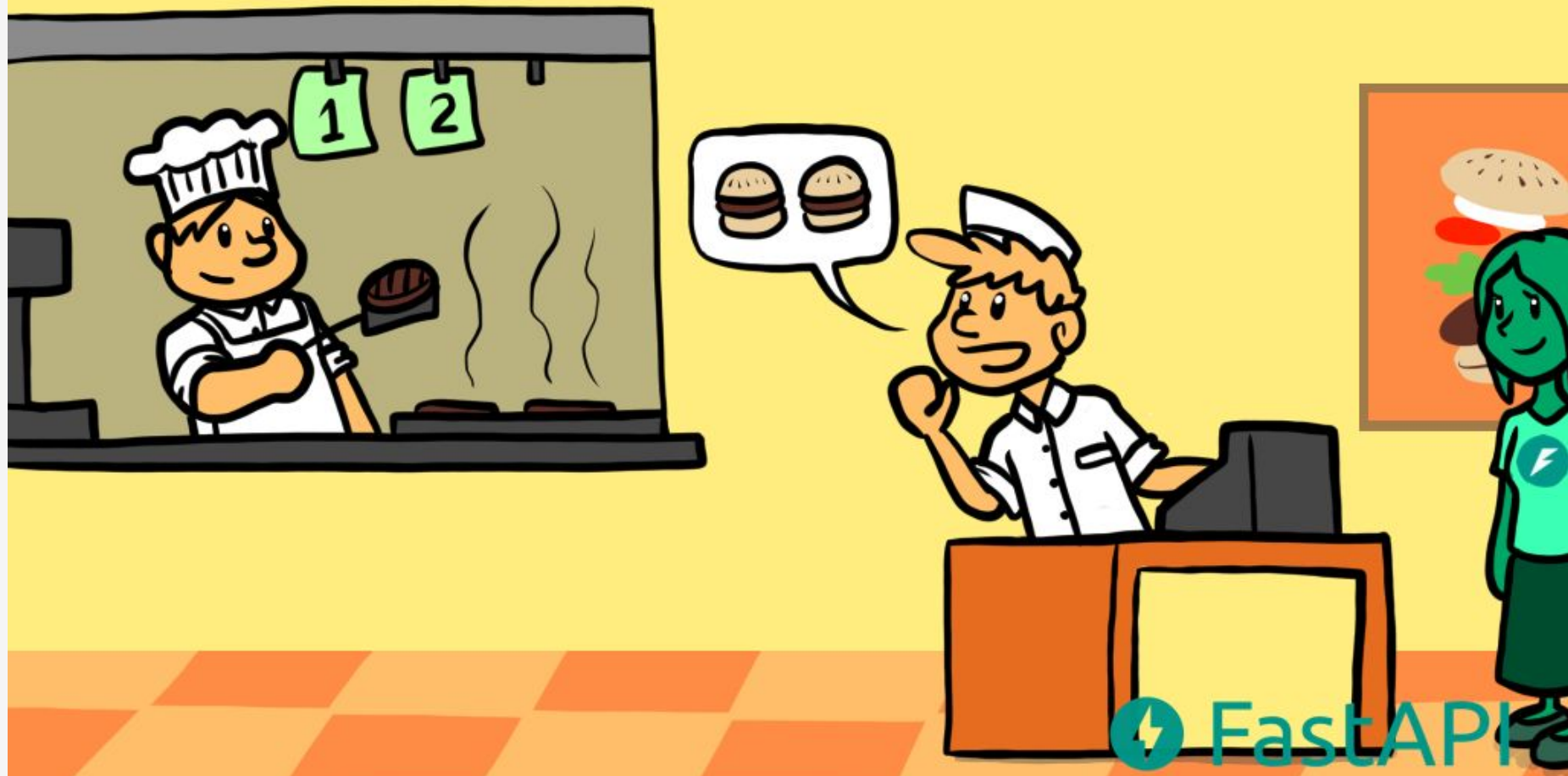
Synchronous != Asynchronous

Synchronous = sequential

Concurrency and Burgers



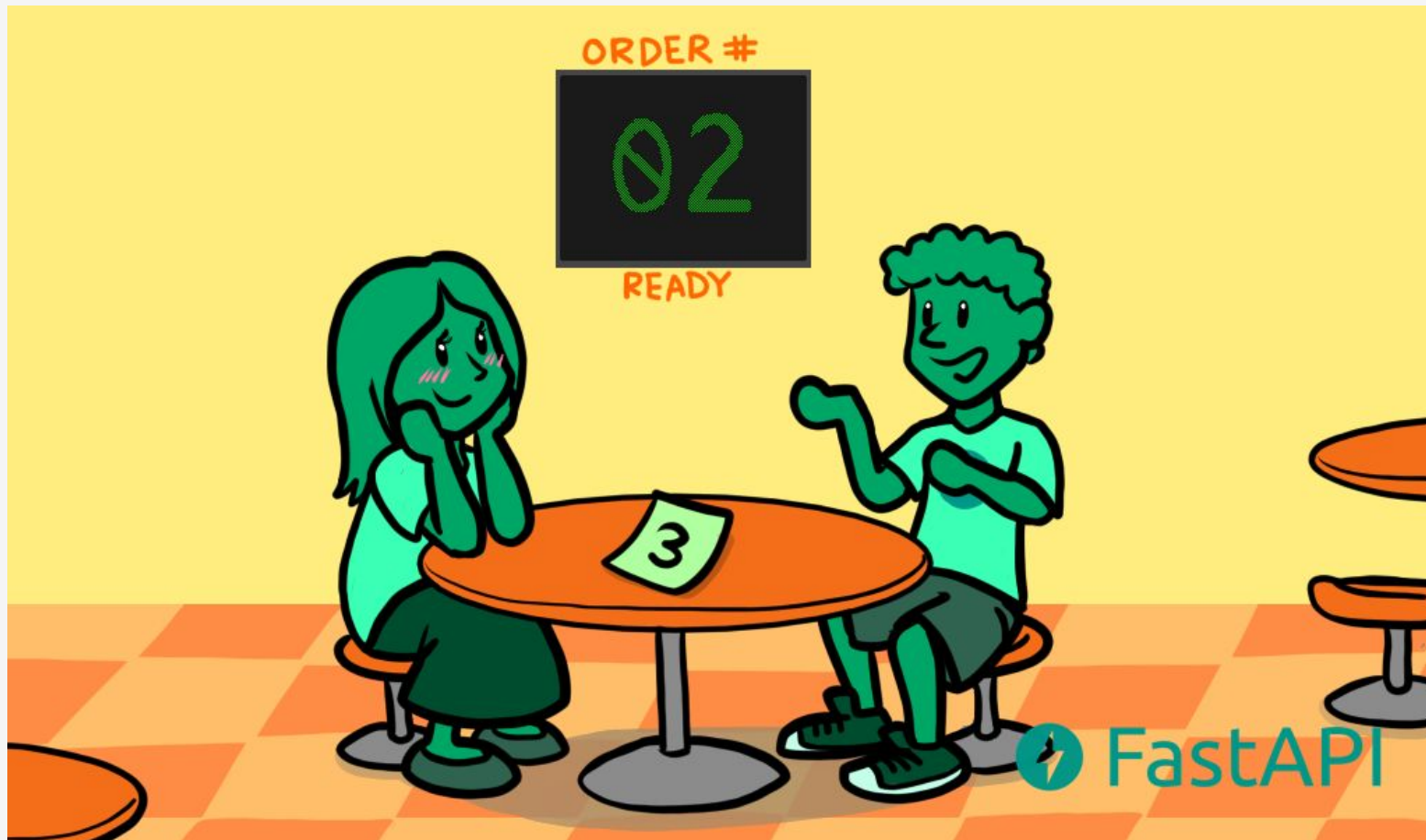




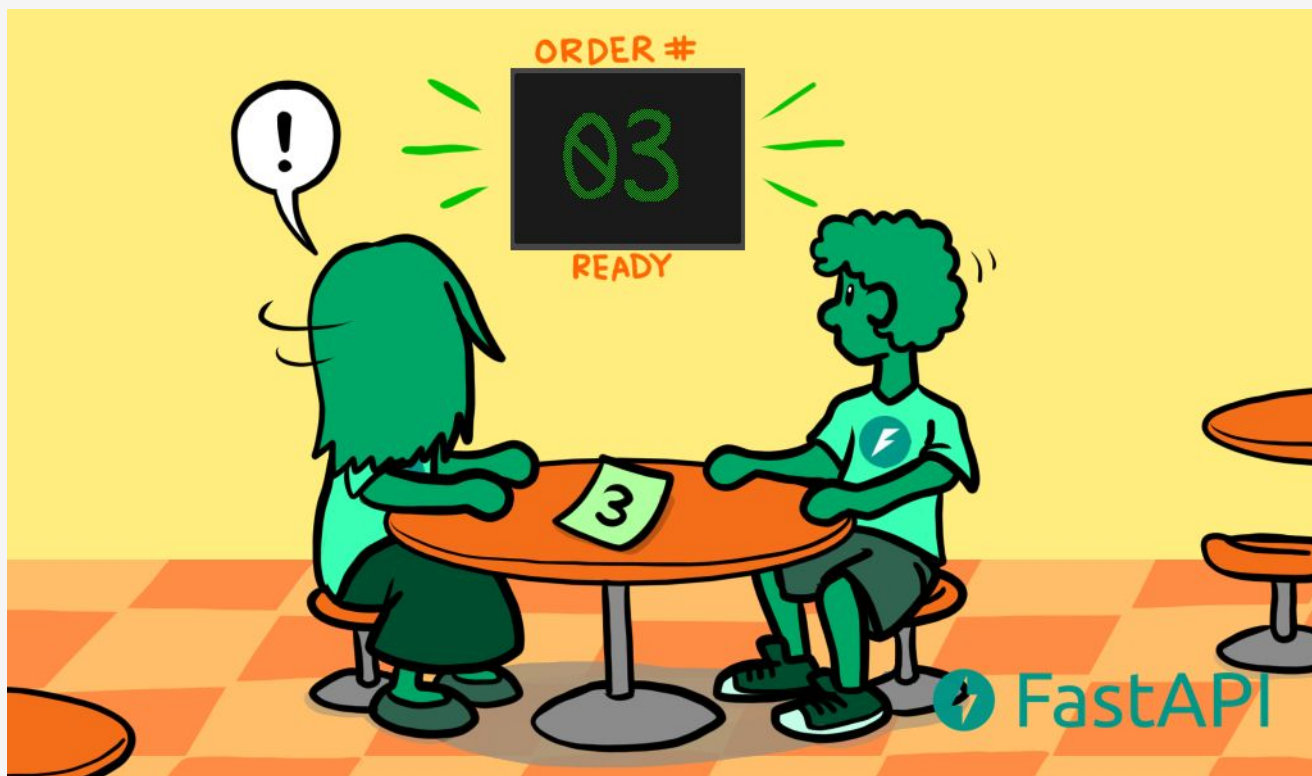
با اینکه آشپز در حال کار روی برگ‌های دیگه‌ای است، اما به او گفتند که دوتا برگ دیگه می‌خواهیم!



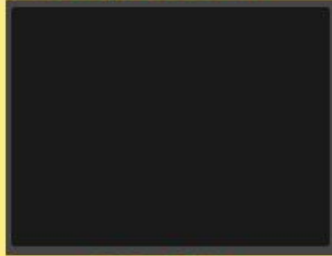
با داشتن شماره سفارش دو مشتری می‌توانند به جای انتظار صحبت کنند.



حال که مشغول صحبت هستند هر از چندگاهی نگاهی به شماره سفارش‌های آماده می‌اندازند تا اینکه شماره سفارش خودشان ظاهر شود.



ORDER #



READY



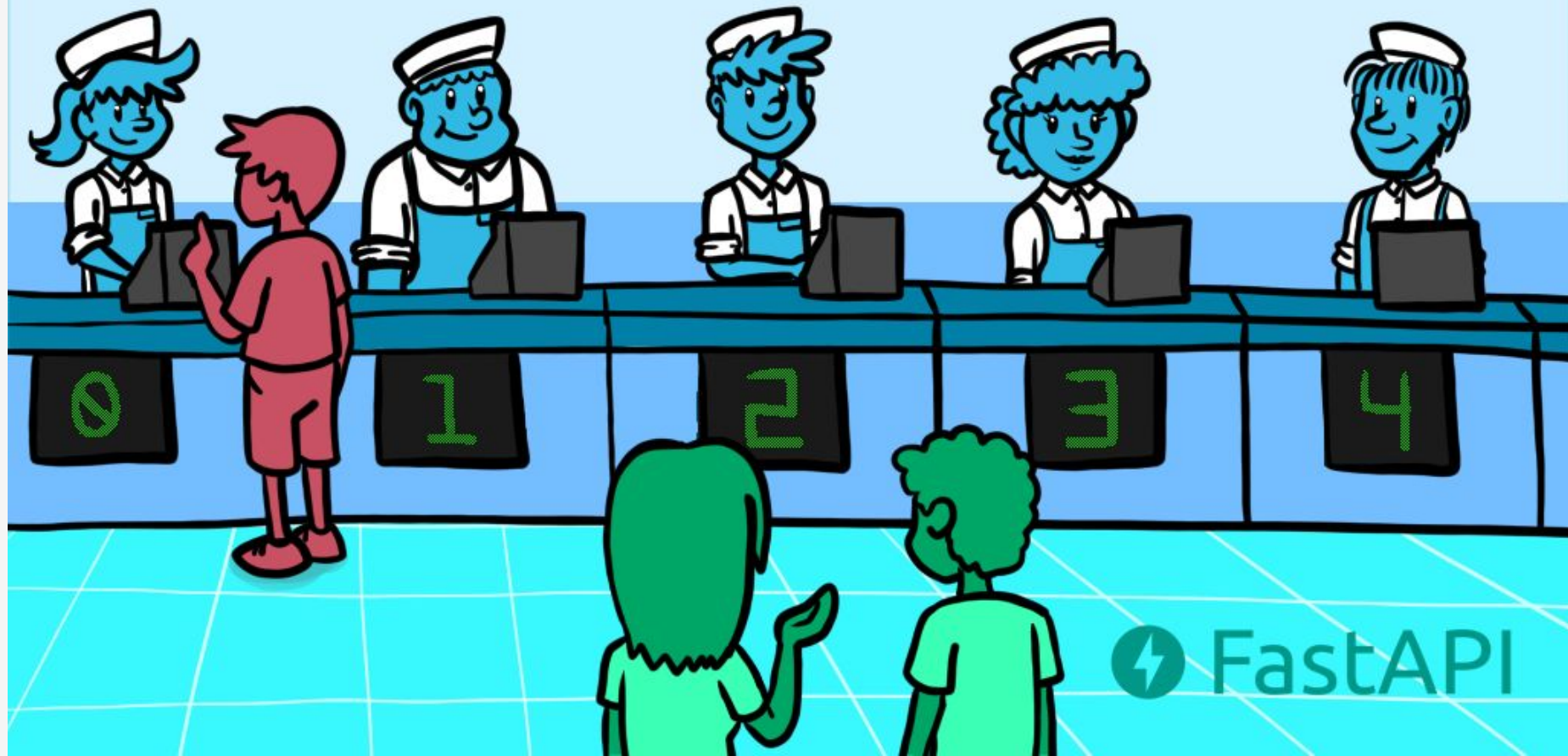
FastAPI

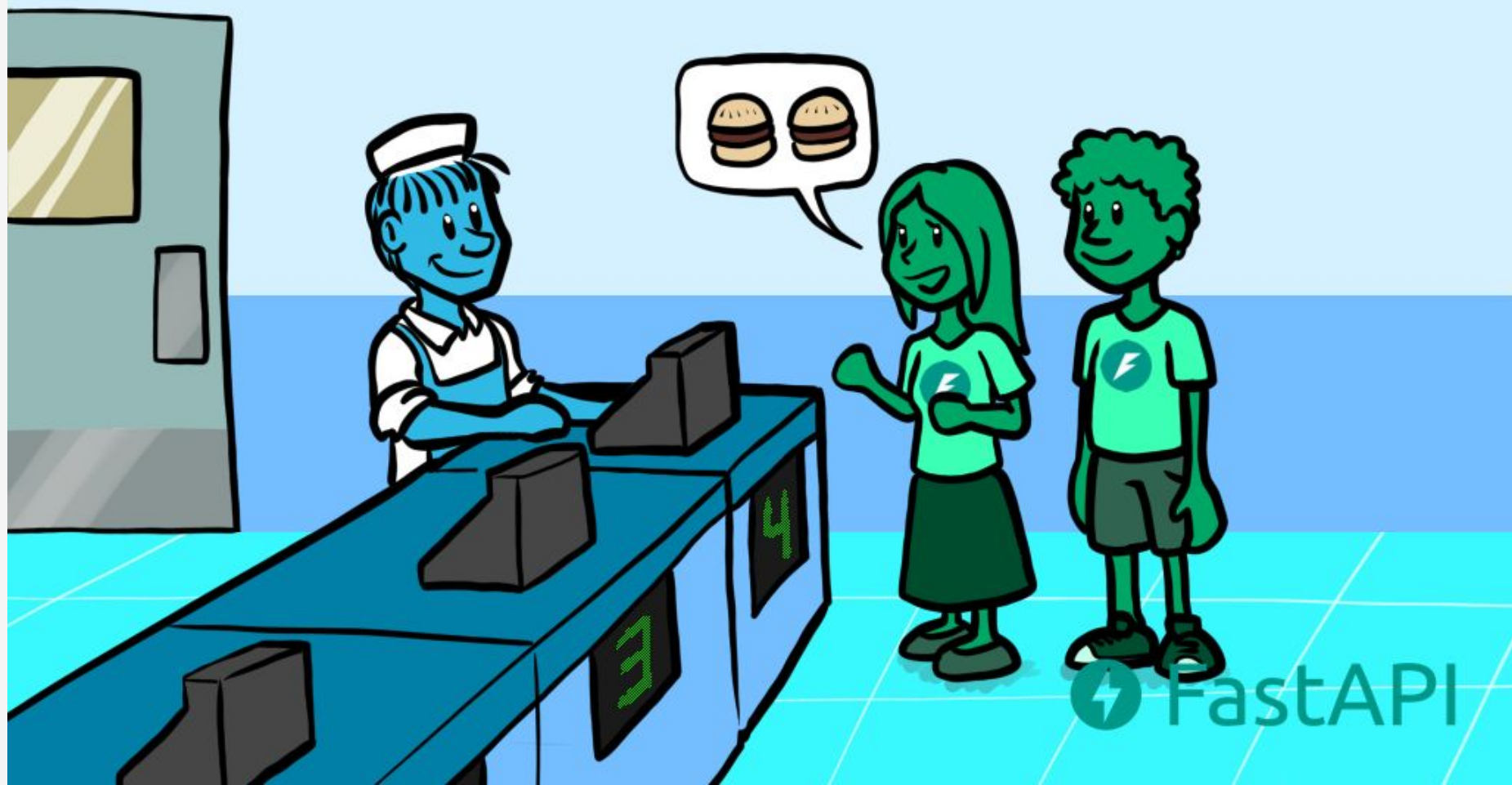
در این داستان:

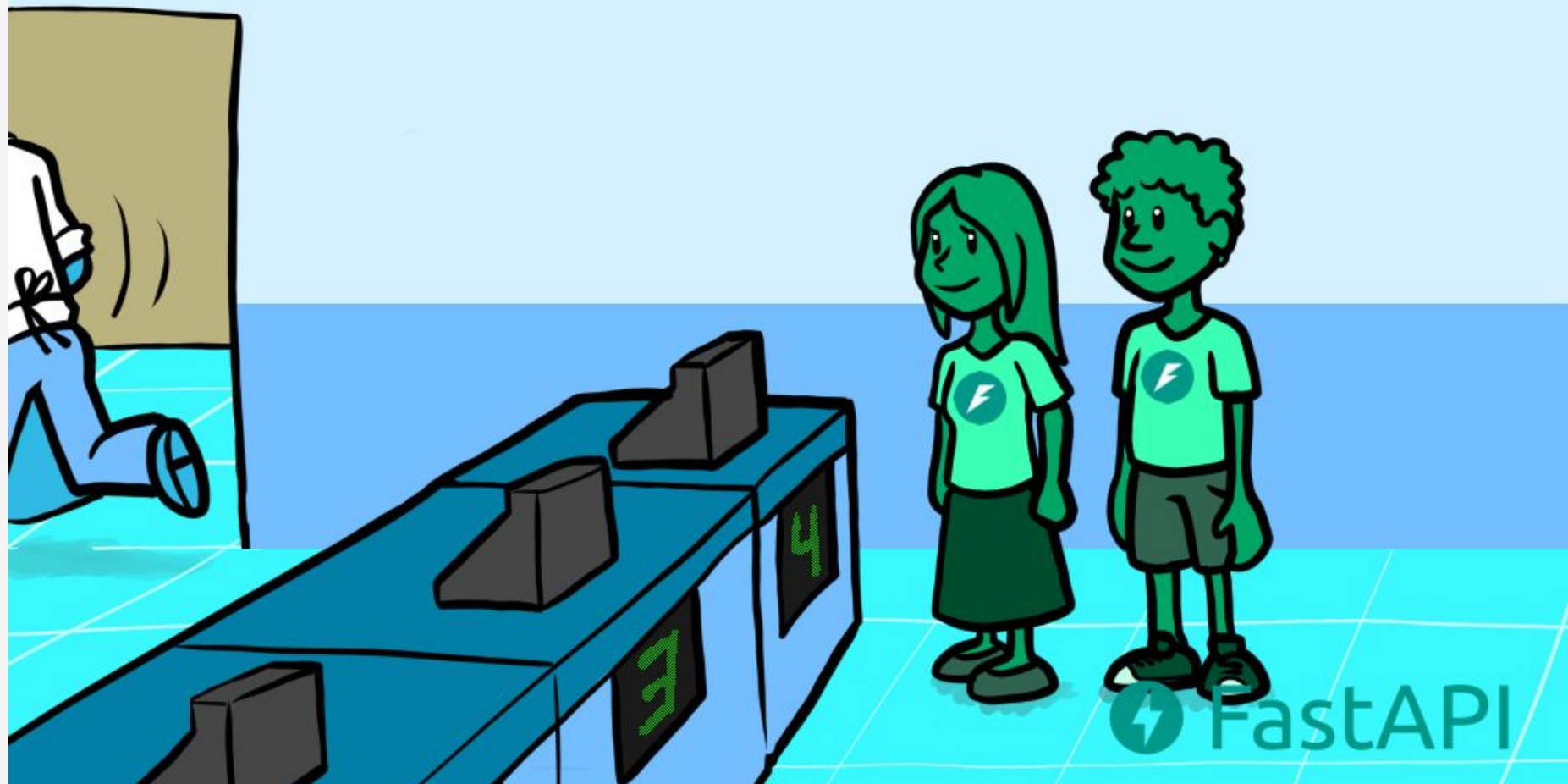
فرض کنید که شما برنامه هستید، هنگامی که در صف پیشخوان هستید:

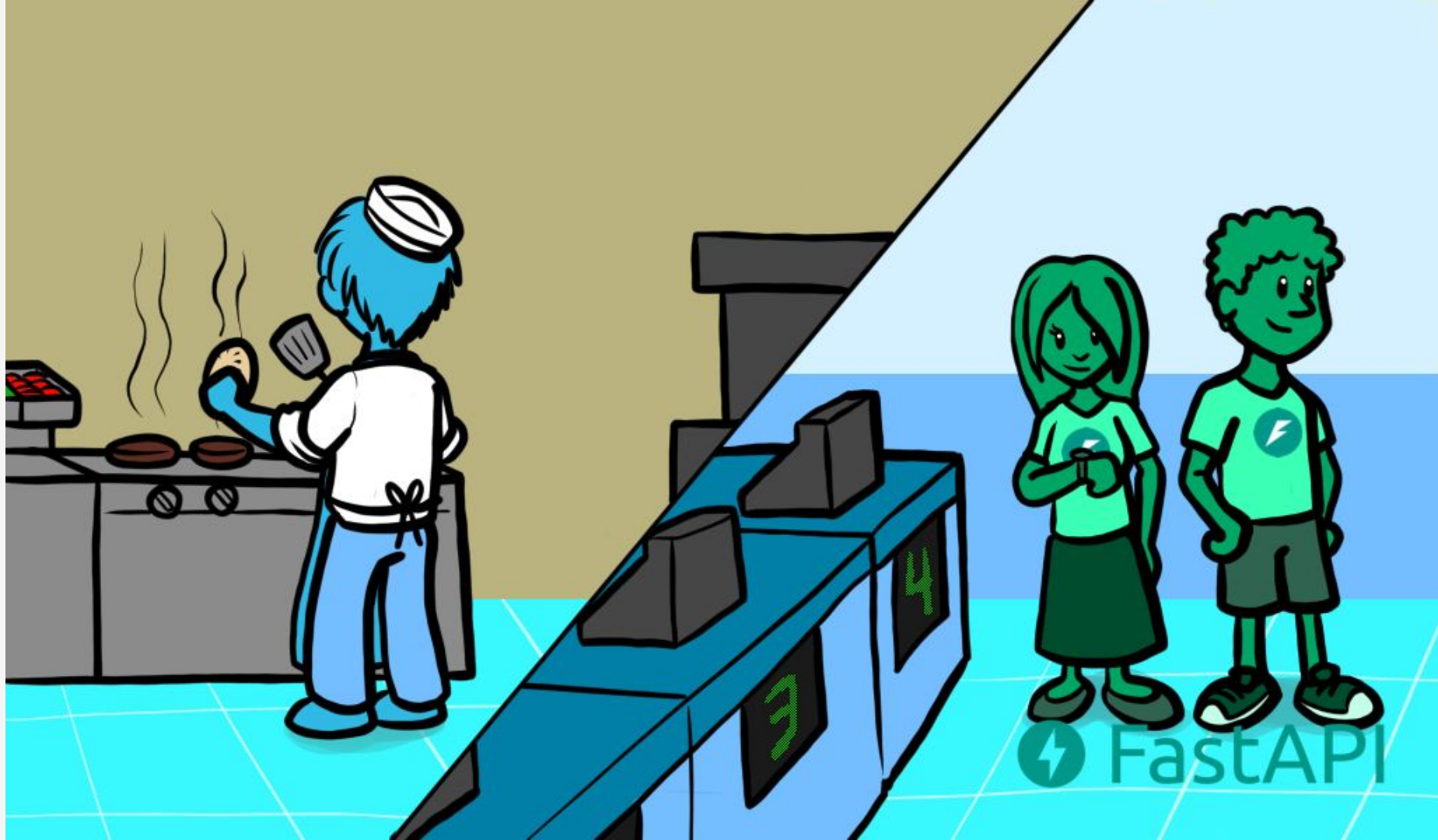
- منتظر می‌شوید نوبت شما شود و موقعی که نوبت شما شد همه‌ی کارهای لازم را انجام می‌دهید (چک کردن منو - آیتم‌ها - ثبت سفارش - پرداخت)
- بعد از آن کار شما با پیشخوان pause می‌شود! (چون هنوز سفارش را تحویل نگرفته اید)
- تا زمان تحویل سفارش مشغول صحبت می‌شوید.
- **موقع تحویل سفارش منتظر می‌شوید صحبت طرف مقابل تمام شود (انجام کامل کارها و محاسبات) بعد از آن سفارش را تحویل می‌گیرید.**
- **حال که برگر را تحویل گرفتید این تسک تمام شده است و تسک بعدی خوردن برگر است.**

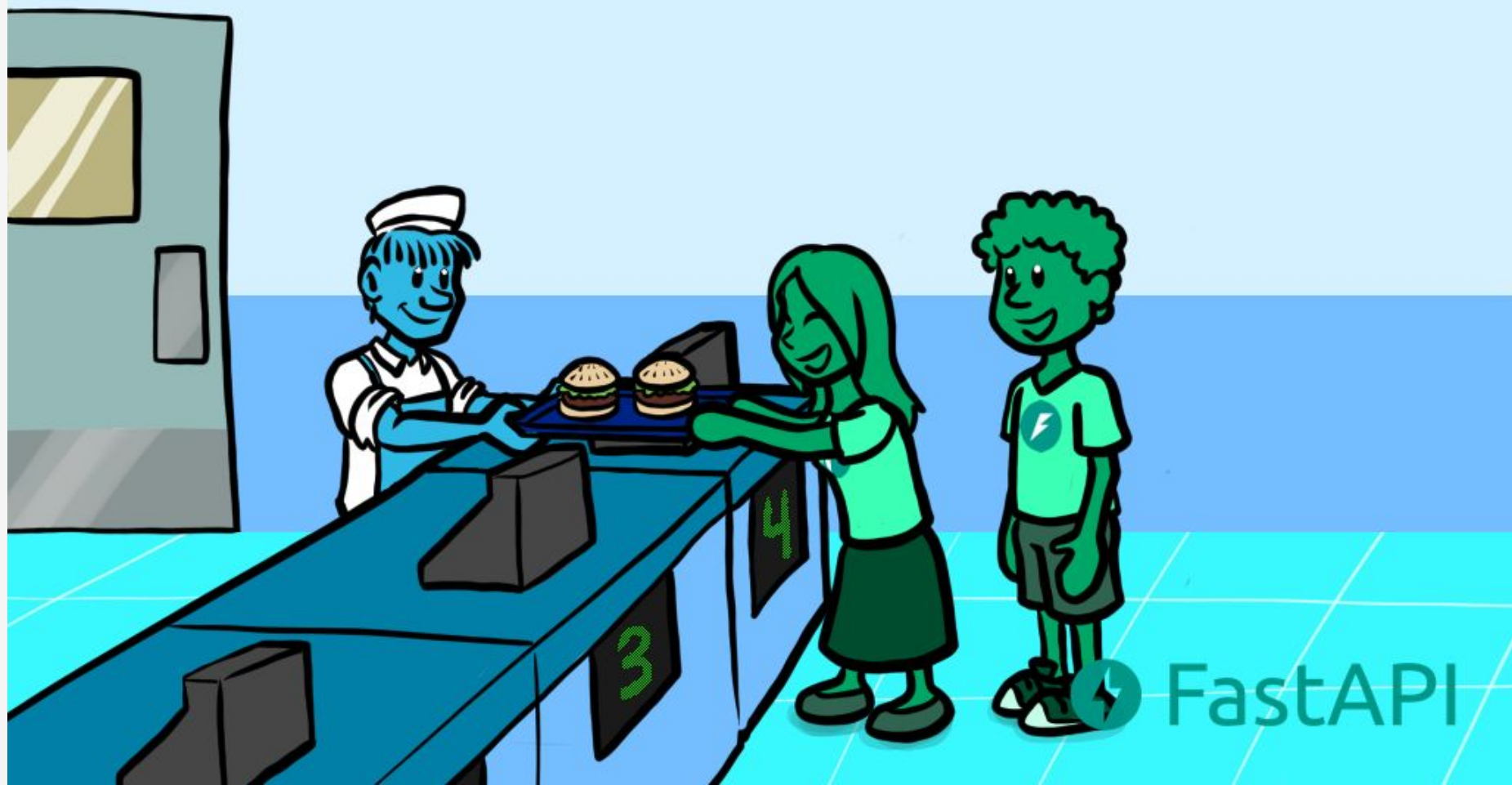
Parallel Burgers

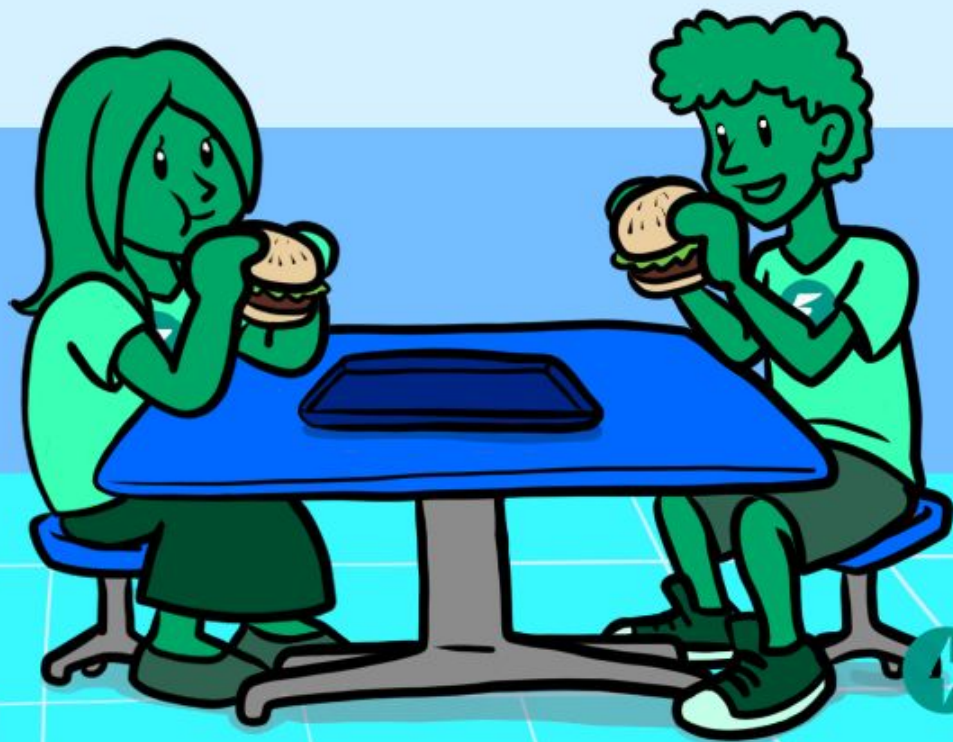












FastAPI

در این حالت شما با آشپز و پیشخوان سینک بودید و فقط منتظر بودید تمام شود.

async and await

```
burgers = await get_burgers (2)
```

این به پایتون می‌گوید برو کارهای دیگری انجام بده تا این قسمت تمام شود.

البته باید در تابعی که **async** دارد صدا زده شود.

```
async def get_burgers(number: int):  
    # Do some asynchronous stuff to create the burgers  
    return burgers
```

```
# This is not asynchronous  
def get_sequential_burgers(number: int):  
    # Do some sequential stuff to create the burgers  
    return burgers
```

```
# This won't work, because get_burgers was defined with: async def  
burgers = get_burgers(2)
```

این کد کار نمی‌کند، چون باید منتظر (await) شود.

هر تابعی که `async` دارد باید توسط `await` صدا زده شود که خودش
باید داخل تابع با `async` باشد!



FastAPI

این مساله را هندل خواهد کرد

Coroutines

به چیزی که از تابع `async def` برمی گردد می گویند.

در نهایت این تعریف مثل تابع است اما قابلیت `pause` دارد. و به کل پروسه شروع توقف و پایتان این نوع توابع `coroutines` گفته می شود.

Virtual Environments

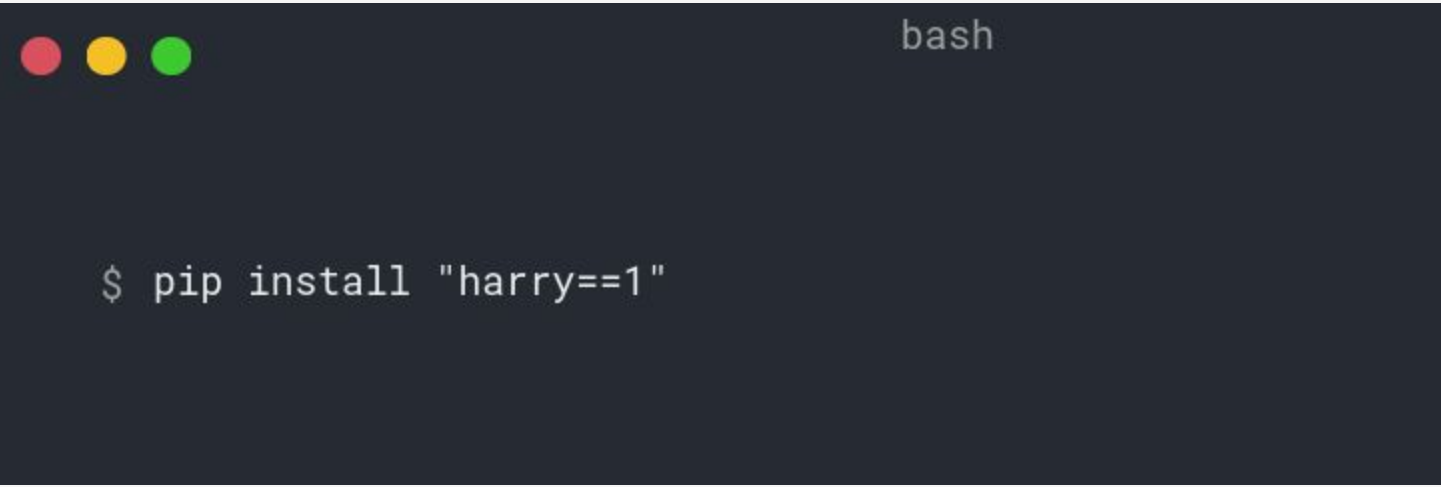
```
python -m pip install --upgrade pip
```

.gitignore

```
pip install -r requirements.txt
```


philosophers-stone —requires→ harry v1

prisoner-of-azkaban —requires→ harry v3



bash

```
$ pip install "harry==1"
```

philosophers-stone project

philosophers-stone

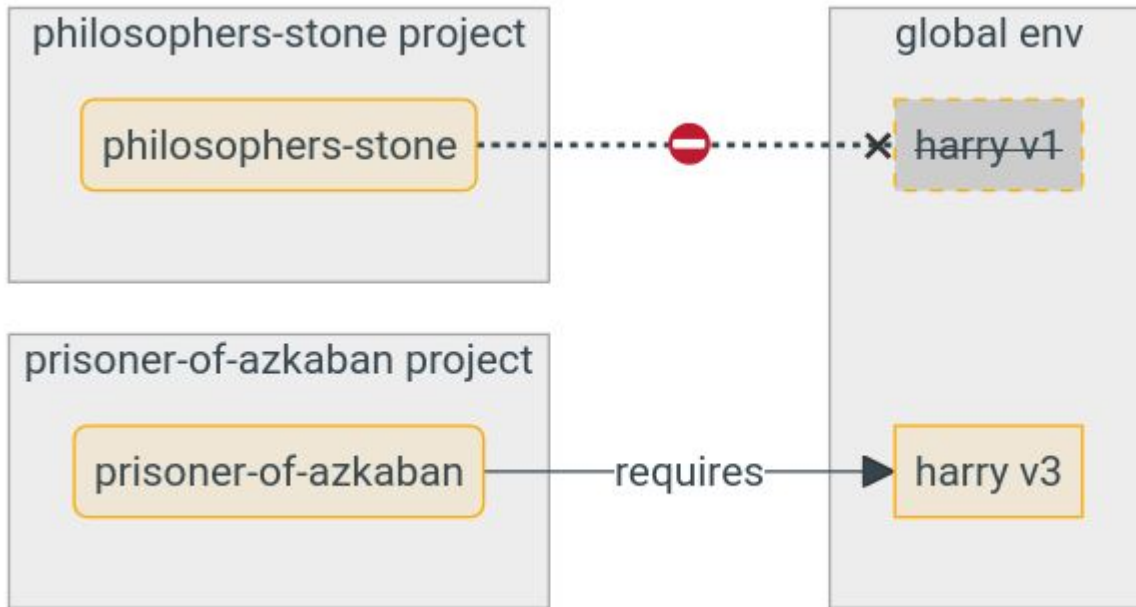
requires

global env

harry v1



```
graph LR; subgraph "philosophers-stone project"; PS[philosophers-stone]; end; subgraph "global env"; HV[harry v1]; end; PS -- requires --> HV;
```

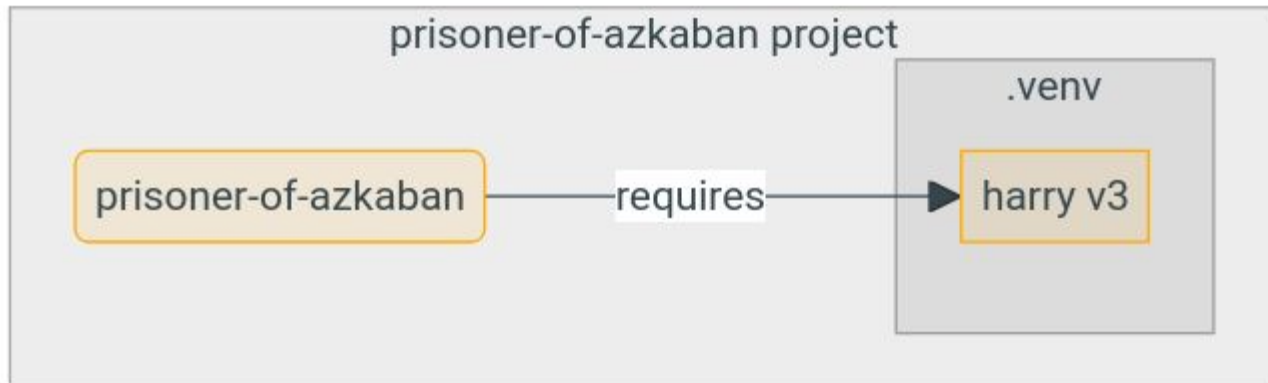
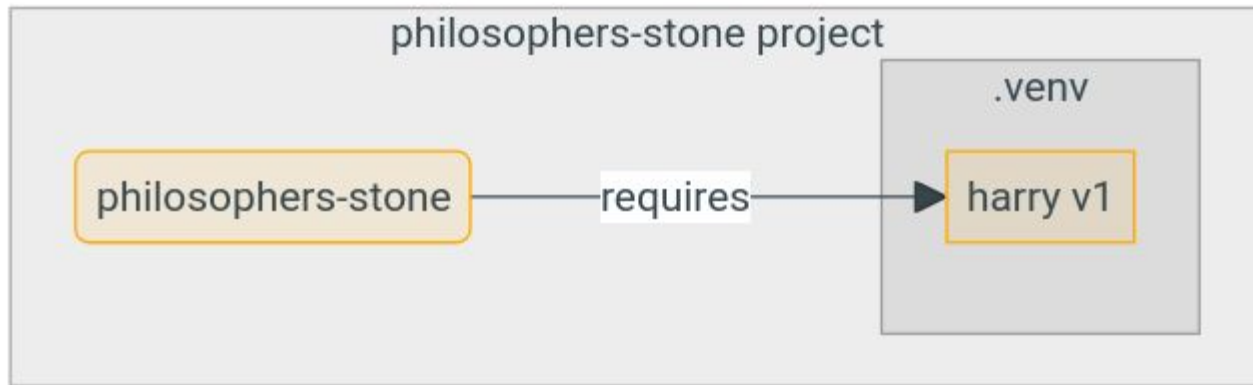


پکیج‌ها کجا نصب می‌شوند؟

```
bash
```

Don't run this now, it's just an example 🤓

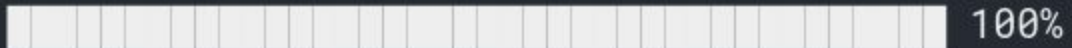
```
$ pip install "fastapi[standard]"
```



```
$ .venv\Scripts\Activate.ps1
```

نصب FastAPI

```
$ pip install "fastapi[standard]"
```




```
from fastapi import FastAPI
```

```
app = FastAPI()    -> یک نمونه ایجاد می‌کند
```

```
@app.get("/")
```

```
async def root():
```

```
    return {"message": "Hello World"}
```

```
$ fastapi dev main.py
```


```
INFO      Using path main.py
```

```
INFO      Resolved absolute path /home/user/code/awesomeapp/main.py
```

```
INFO      Searching for package file structure from directories with  
__init__.py files
```

```
INFO      Importing from /home/user/code/awesomeapp
```

Python module file

 `main.py`

<http://127.0.0.1:8000>

```
{"message": "Hello World"}
```

حالت تعاملی

<http://127.0.0.1:8000/docs>

Fast API - Swagger UI

127.0.0.1:8000/docs

Fast API

0.1.9 OAS3

/openapi.json

default

⌵

GET

/items/{item_id}

Read Item Get

Parameters

Try it out

Name	Description
item_id ★ required	
integer	
(path)	
q	
string	
(query)	

Responses

Code	Description	Links
200	<div>Successful Response</div> <div>application/json</div> <div>Controls Accept header.</div>	No links
422	<div>Validation Error</div> <div>application/json</div>	No links

Example Value | Schema

```
{  "detail": [    {      "loc": [        "string"      ]    }  ]}
```

```
from fastapi import FastAPI
```

`app = FastAPI()` → یک نمونه ایجاد می‌کند

`@app.get("/")` → آدرس‌ها

```
async def root():
```

```
    return {"message": "Hello World"}
```

```
https://example.com/items/foo
```

...the path would be:

```
/items/foo
```

"endpoint" or a "route".

Operation

- POST
- GET
- PUT
- DELETE

Operation

- POST
- GET
- PUT
- DELETE

در پروتکل HTTP می‌توان به یک **path** با یک یا چند نوع از متدهای بالا ارتباط برقرار کرد.

- **POST:** برای ایجاد دیتا
- **GET:** خواندن دیتا
- **PUT:** آپدیت دیتا
- **DELETE:** حذف دیتا.

در OpenAPI متدهای HTTP را Operation می‌گوییم.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

• موارد دیگر

- @app.post()
- @app.put()
- @app.delete()

می‌توان اینگونه نیز تعریف کرد.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def root():
    return {"message": "Hello World"}
```

```
return {"message": "Hello World"}
```

می‌توان `dict` یا `list` یا مقادیر واحدی مانند `int`, `str` برگرداند.

بنابراین کاری که تا الان انجام دادیم:

- Import `FastAPI`.
- `app`
- path operation decorator `@app.get("/")`.
- path operation function `def root():`
- `fastapi dev`.

Path Parameters

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id):
    return {"item_id": item_id}
```


Path parameters with types

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}
```

در این حالت `item_id` نوع `int` خواهد بود.