Wojciech Zwoliński, Małgorzata Szeler, Brian Ndamira

Most Danych

Praktyczny Poradnik Obsługi API m.st. Warszawa dla Początkujących

HYDRA
science foundation

projekt finansuje
miasto stołeczne
Warszawa

Przedmowa

Niniejszy poradnik został opracowany w ramach projektu **„Data Bridge: Warszawa – Tworzenie materiałów edukacyjnych z obsługi API m.st. Warszawa”** realizowanego przez **Hydra Science Foundation**.

Poradnik jest przeznaczony dla osób bez wykształcenia informatycznego, zachęcając je do korzystania z danych miejskich w codziennym życiu. Dzięki praktycznym przykładom i przystępnym instrukcjom użytkownicy mogą rozwijać umiejętności pracy z danymi niezależnie od poziomu zaawansowania. Jego celem jest zwiększenie świadomości oraz ułatwienie dostępu do otwartych danych udostępnianych przez miasto Warszawa. Poradnik powstał we współpracy z mieszkańcami – odbiorcy aktywnie uczestniczyli w procesie jego tworzenia. W ramach projektu zorganizowano konsultacje, podczas których uczestnicy zgłaszali uwagi i sygnalizowali potrzeby, dzięki czemu poradnik został dostosowany do ich oczekiwań.

Projekt Data Bridge oraz niniejszy poradnik zostały opracowane przez Fundację Hydra Science Foundation, która zajmuje się łączeniem ekspertów w celu tworzenia społecznie istotnych rozwiązań i prowadzenia dyskursu społecznego na temat nowych technologii oraz cyfryzacji.

Dziękujemy wszystkim osobom, które w trakcie realizacji projektu przyczyniły się do powstania tego poradnika. Wyrażamy wdzięczność uczestnikom spotkań za ich uwagi oraz pomysły, które wzbogaciły treść poradnika. Szczególne podziękowania kierujemy do przedstawicieli m.st. Warszawy z Biura Informatyki, w szczególności do Pana Roberta Kunickiego i Pani Marty Trzaskowskiej, za ich cenne sugestie dotyczące kształtu i zawartości tego opracowania.

Spis treści

1 Wstęp **1**

1.1 Motywacja . . . 1

1.2 Czym jest API? . . . 2

1.3 API na przykładach . . . 2

1.3.1 Połączenie drukarki z komputerem . . . 2

1.3.2 Jakdojade . . . 3

1.3.3 Warszawska Platforma IoT . . . 4

2 Podziały API **5**

2.1 API sieciowe a API lokalne . . . 5

2.2 API Prywatne a API Publiczne . . . 7

2.3 Architektury API . . . 9

2.3.1 REST API . . . 9

2.3.2 RPC API . . . 9

2.3.3 SOAP API . . . 10

2.3.4 GraphQL . . . 10

3 Omówienie działania komunikacji HTTP: Zapytanie i Odpowiedź **10**

3.1 Metody HTTP . . . 11

3.2 Adres URL . . . 12

3.2.1 URL w REST API . . . 13

3.2.2 URL w RPC API . . . 13

3.3 Wysyłanie zapytania . . . 13

3.4 Odpowiedź serwera . . . 14

3.5 Autoryzacja użytkownika API . . . 16

4 Strona api.um.warszawa.pl **17**

4.1 CKAN API . . . 18

4.2 Jak korzystać ze strony . . . 18

4.3 Specyfika innych dostępnych baz . . . 19

4.4 Generowanie klucza API . . . 20

5 Przykład korzystania z API za pośrednictwem przeglądarki **21**

6 Python i API w Pythonie **23**

6.1 Instalacja Pythona, IDE, Konfiguracja środowiska . . . 24

6.2 Wprowadzenie do Pythona . . . 29

6.2.1 Obsługa Jupyter Notebook . . . 29

6.3 Twoje Pierwsze Kroki w Pythonie . . . 31

6.3.1 Kilka słów: Czym Jest Kernel? . . . 41

6.4 Spis narzędzi używanych w przykładach . . . 43

6.4.1 Matplotlib . . . 43
6.4.2 Pandas . . . 48
6.4.3 Requests . . . 54
6.5 Przykłady użycia bibliotek z API Warszawy . . . 58
6.5.1 Wczytywanie klucza . . . 58
6.5.2 Pobieranie danych . . . 58
6.5.3 Eksportowanie danych do formatu CSV . . . 60
6.5.4 Importowanie danych JSON do pandas DataFrame . . . 61
6.5.5 Analiza i Wizualizacja danych . . . 63

7 Praktyczne użycia API **69**
7.1 Konsultacje społeczne - wykorzystanie . . . 69
7.1.1 Opis Zadania . . . 69
7.1.2 Co to znaczy dla NGO . . . 69
7.1.3 Cel i rezultaty zadania . . . 69
7.1.4 Analiza problemu . . . 70
7.2 Parkingi miejskie - Scraping Danych . . . 74
7.2.1 Opis Zadania . . . 74
7.2.2 Cel i rezultaty zadania . . . 74
7.2.3 Analiza problemu . . . 74
7.3 Parkingi miejskie - Wizualizacja Danych . . . 78
7.3.1 Opis Zadania . . . 78
7.3.2 Co to znaczy dla NGO . . . 78
7.3.3 Cel i rezultaty zadania . . . 78
7.3.4 Analiza problemu . . . 78
7.4 ZTM stats . . . 82
7.4.1 Opis Zadania . . . 82
7.4.2 Cel i rezultaty zadania . . . 82

1 Wstęp

1.1 Motywacja

W trakcie rozwoju informatyki powstawało coraz więcej stron internetowych, urządzeń i systemów, takich jak komputery osobiste, drukarki, telewizory czy różnego rodzaju czujniki, wyposażonych w swój własny, niezależny program, który ułatwia lub umożliwia ich obsługę czy automatyzuje zadania. Programy te były zoptymalizowane pod kątem swoich funkcji, operując na zestawie pojęć adekwatnych do swoich celów. W miarę ich popularyzacji naturalnie pojawiła się potrzeba komunikacji między niezależnymi, zupełnie różnymi programami, które jednak posiadały informacje wzajemnie im potrzebne. Przykładem tego może być program umożliwiający tworzenie grafiki, który działa w pamięci tymczasowej komputera, oraz menedżer plików kontrolujący zapis i odczyt na dysku twardym. Aby zapisać stworzoną przez użytkownika grafikę, te programy muszą się ze sobą skomunikować, wymieniając dane o obrazku.

Naiwnym sposobem rozwiązania tego problemu mogłoby być połączenie dwóch programów w taki sposób, aby miały wzajemny dostęp do wszystkich swoich danych. Jednak jest to rozwiązanie nadmiarowe i obciążające. Program graficzny nie potrzebuje informacji o technicznych szczegółach zapisu danych, takich jak system plików. Z drugiej strony menedżer plików nie potrzebuje informacji o tym, czy użytkownik obecnie korzysta z funkcji „pędzla” czy „ołówka”, ani o kolorze, którym maluje. Ponadto, przy scaleniu ze sobą dwóch programów, zawsze gdy chcielibyśmy zapisać lub otworzyć plik, komputer musiałby załadować program do rysowania i odwrotnie. Podczas tworzenia obrazów w tle musiałyby być oczekujące wszystkie funkcje związane z zarządzaniem danymi na dysku.

Naturalnie wydaje się konieczne stworzenie „pośrednika”, lub inaczej „protokółu komunikacji”, który pozwoli programom wymieniać między sobą niezbędne informacje bez znaczącego obciążania ich zbędnymi danymi. Tym pośrednikiem jest właśnie **API**.

1.2 Czym jest API?

API (Application Programming Interface), czyli **Interfejs Programowania Aplikacji**, to zestaw zasad i narzędzi, które umożliwiają różnym programom współpracę ze sobą. Kiedy jakaś aplikacja lub system udostępnia swoje API, pozwala to innym aplikacjom na łatwe korzystanie z jej funkcji lub danych, bez konieczności znajomości jej wewnętrznego działania.

Praktycznie oznacza to, że dzięki API jedna aplikacja może korzystać z funkcji lub zasobów innej aplikacji. Jest to bardzo ważne przy tworzeniu skomplikowanych i rozbudowanych systemów, gdzie wiele programów musi ze sobą współpracować.

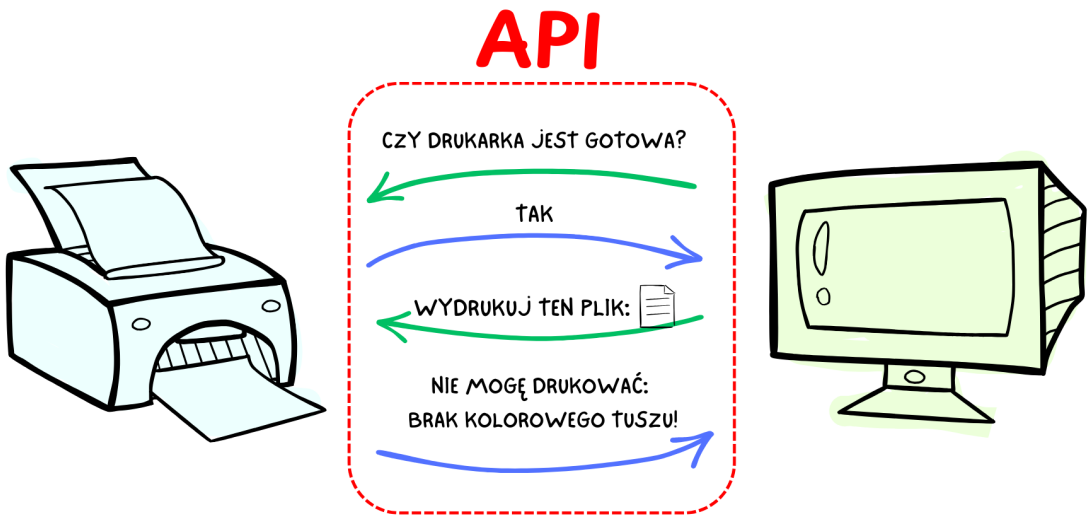
API działa jak pośrednik między użytkownikiem a dostawcą funkcji. Oznacza to, że programista korzystający z danego API nie musi wiedzieć, jak dokładnie działają poszczególne funkcje "pod maską". Może po prostu używać gotowych metod, narzędzi i operacji oferowanych przez API. Dzięki temu tworzenie aplikacji jest szybsze, a programy mogą łatwo integrować funkcje pochodzące z różnych źródeł.

1.3 API na przykładach

1.3.1 Połączenie drukarki z komputerem

API to nie tylko sposób komunikacji między programami działającymi na tym samym komputerze. Dzięki API odrębne aplikacje mogą współpracować, nawet jeśli działają na różnych urządzeniach, które nie są tradycyjnie uznawane za komputery. Przykładem z życia codziennego jest komunikacja między komputerem a drukarką.

Aby doszło do wydrukowania pliku, komputer musi upewnić się, czy drukarka jest obecna, a jeśli tak, przesłać jej plik do wydrukowania. Następnie drukarka powiadamia komputer o sukcesie drukowania lub jego niepowodzeniu.

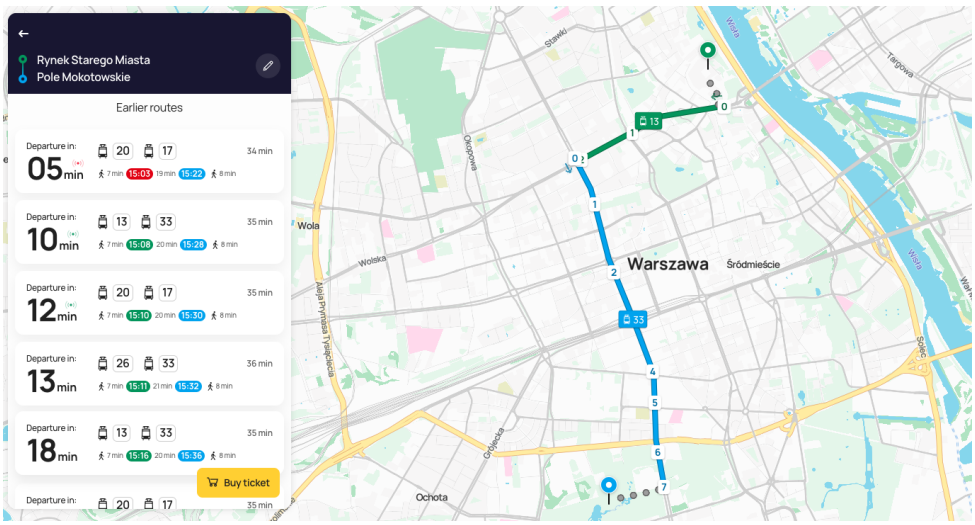


Rysunek 1: Komunikacja między komputerem a drukarką za pomocą API

1.3.2 Jakdojade

Obecnie bardzo popularna i powszechnie używana aplikacja Jakdojade, pokazująca różne metody przemieszczania się za pomocą komunikacji miejskiej w czasie rzeczywistym, również opiera się na komunikacji poprzez API. Aby pokazać użytkownikowi, jak dotrzeć do celu autobusem czy tramwajem, aplikacja musi mieć dostęp do aktualnych danych o rozkładach jazdy oraz o miejscu, w jakim znajdują się w danej chwili interesujące nas środki transportu. W tym celu łączy się z bazami danych udostępniającymi takie informacje, jak ta udostępniana przez dane miasto lub ZTM, i pobiera jedynie te informacje, które są niezbędne dla wybranej trasy w danym czasie. Aplikacja korzysta z plików typu GTFS oferowanych przez ZTM posiadających między innymi dane na temat przystanków, rozkładów jazdy oraz tras pojazdów transportu publicznego. Tego typu informacje nie są zbyt często aktualizowane, więc wygodnie jest je przechowywać w odpowiednio dostosowanej formie w wewnętrznej bazie firmy. Z drugiej strony dane na temat położenia autobusów czy tramwajów w danym czasie, udostępniane przez miasto, muszą być regularnie uaktualniane. Zarówno komunikacja z wewnętrzną bazą danych zawierającymi stosunkowo stałe informacje, jak i komunikacja z serwerami miejskimi, przebiega za pośrednictwem API.

Najpierw, wybierając interesujące nas miasto, w którym chcemy podróżować, aplikacja otrzymuje informację, z którą bazą danych będzie się komunikować. Gdy wpisujemy punkt początkowy i końcowy naszej podróży, aplikacja wysyła zapytania na temat tego, jaki autobus ma w swoim rozkładzie przystanki najbliższe naszemu punktowi startowemu i celowi, lub w przypadku połączeń wieloautobusowych, który autobus jedzie w stronę



Rysunek 2: Zrzut ekranu pochodzi ze strony internetowej Jakdojade (https://jakdojade.pl/warszawa, dostęp: 09.12.2024).

naszego celu i kolejny autobus, dla którego punkt startowy jest miejscem wysiadki z poprzedniego. W taki sposób aplikacja pobiera tylko wycinek danych adekwatnych do naszej podróży. Jeśli wybierzemy już interesujące nas połączenie, możemy zauważyć, że aplikacja często podaje, gdzie aktualnie znajduje się autobus. Do tego również komunikuje się z API konkretnej bazy, która udostępnia takie dane. Dzięki temu dostajemy informacje o ewentualnych opóźnieniach w rozkładzie.

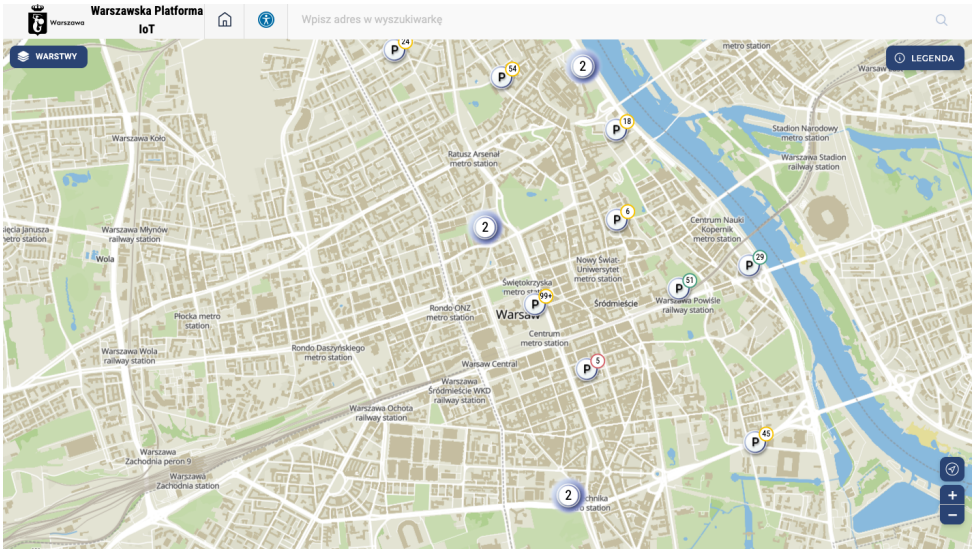
Gdyby nie istniało API pozwalające na komunikację z bazami danych, aplikacja musiałaby przechowywać pełne bazy danych wszystkich rozkładów jazdy w całej Polsce. To oznaczałoby, że przy każdym uruchomieniu aplikacji użytkownik musiałby czekać na załadowanie tych baz, nawet jeśli planuje przejechać jedynie daną trasę. Jak można się domyślać, taki sposób działania znacznie spowolniłby aplikację, a jej instalacja wymagałaby dużej ilości wolnej pamięci w urządzeniu. Co więcej, jakiekolwiek zmiany w tej bazie (takie jak zmiana rozkładu jazdy lub informacja o opóźnieniu) nie byłyby dostępne aplikacji i jej użytkownikowi.

Właśnie dlatego API umożliwiające dynamiczną komunikację, odbywającą się dopiero przy określeniu trasy, jest niezbędne do wygodnego korzystania z aplikacji.

1.3.3 Warszawska Platforma IoT

Warszawska Platforma IoT (Internet of Things) prezentuje dane pochodzące głównie z sensorów umieszczonych w przestrzeni miasta w sposób przyjazny dla obywatela. Znajdują się na niej dane aktualizowane w czasie zbliżonym do rzeczywistego, obejmujące między innymi informacje na temat jakości powietrza, pogody, wolnych miejsc parkingowych, lokalizacji toalet czy lokalizacji defibrylatorów.

Analiza danych z platform IoT może mieć szerokie zastosowania w różnych dziedzinach. Na przykład dane o jakości powietrza mogą być wykorzystywane zarówno do planowania aktywności na świeżym powietrzu, jak i do badań nad długoterminowymi zmia-



Rysunek 3: Zrzut ekranu Warszawskiej Platformy IoT (https://iot.warszawa.pl/mapa?filter=parking, dostęp: 09.12.2024).

nami środowiskowymi oraz oceną skuteczności polityk ekologicznych. Informacje te mogą również wspierać rozwój algorytmów sztucznej inteligencji, umożliwiając przewidywanie poziomów zanieczyszczeń czy wzorców ruchu drogowego. Z kolei dane o natężeniu ruchu w mieście mogą być przydatne firmom w planowaniu lokalizacji reklam i punktów sprzedaży, a także w optymalizacji i zarządzaniu transportem publicznym poprzez identyfikację obszarów o największym obciążeniu pasażerskim. Aby wszystkie te zastosowania były możliwe, niezbędny jest dostęp do danych w ustrukturyzowanej formie, co zapewniają interfejsy API.

2 Podziały API

API można podzielić na różne rodzaje, w zależności od ich funkcji i sposobu działania. Istnieją podziały na API lokalne i sieciowe, publiczne i prywatne, a także według stylu architektury, jak REST, SOAP czy RPC. Każdy z tych podziałów odpowiada na różne potrzeby i wymagania systemów, z którymi API współpracują, oraz na sposób, w jaki dane są przesyłane i przetwarzane. Dzięki temu API mogą być lepiej dopasowane do potrzeb programistów.

2.1 API sieciowe a API lokalne

API może działać zarówno lokalnie, na jednym urządzeniu, jak i sieciowo, umożliwiając komunikację między różnymi systemami przez Internet. Oba podejścia mają swoje zastosowania i różnice, które wpływają na sposób ich działania oraz obszary wykorzystania.

API sieciowe API sieciowe, znane również jako web API, stanowią fundament współczesnych aplikacji internetowych, umożliwiając komunikację i integrację między różnorod-

nymi systemami poprzez sieć, najczęściej za pośrednictwem protokołów takich jak HTTP czy HTTPS. Dzięki API sieciowym, deweloperzy mogą łatwo łączyć swoje aplikacje z zewnętrznymi serwisami, co otwiera drzwi do szerokiego zakresu funkcjonalności bez konieczności tworzenia wszystkiego od podstaw. Przykładowo, Google Maps API pozwala na integrację map w aplikacjach mobilnych i webowych, dostarczając jednocześnie aktualizowane dane geograficzne, natomiast Twitter API umożliwia dostęp do funkcji platformy społecznościowej, takich jak publikowanie tweetów czy pobieranie danych o użytkownikach. API sieciowe są zazwyczaj projektowane w architekturze REST (Representational State Transfer), GraphQL czy SOAP (Simple Object Access Protocol), które definiują sposoby wymiany danych oraz operacji możliwych do wykonania. Ich kluczową zaletą jest skalowalność oraz możliwość obsługi dużej liczby żądań jednocześnie, co jest niezbędne w dynamicznie rozwijających się środowiskach internetowych. Jednakże, ze względu na swoją naturę, API sieciowe muszą być odpowiednio zabezpieczone przed potencjalnymi zagrożeniami związanymi z dostępem przez Internet, co wymaga stosowania zaawansowanych metod autoryzacji i uwierzytelniania.

API lokalne API lokalne, z kolei, są przeznaczone do komunikacji i integracji aplikacji działających na tym samym urządzeniu lub w ramach tego samego systemu operacyjnego. Umożliwiają one dostęp do zasobów systemowych oraz funkcji sprzętowych, co jest niezbędne do tworzenia aplikacji o wysokiej wydajności i głębokiej integracji z systemem operacyjnym. Przykłady takich API obejmują Windows API, które dostarcza zestaw funkcji pozwalających na interakcję z systemem operacyjnym Windows, czy UNIX syscalls, które umożliwiają aplikacjom dostęp do podstawowych usług systemowych w systemach Unix-like. API lokalne często wykorzystują biblioteki DLL (Dynamic Link Libraries) lub inne mechanizmy dynamicznego ładowania kodu, co pozwala na efektywne zarządzanie zasobami i modularność aplikacji. Dzięki API lokalnym, programiści mogą tworzyć zaawansowane aplikacje desktopowe, które wymagają bezpośredniego dostępu do sprzętu, takie jak edytory grafiki, gry czy narzędzia do zarządzania systemem. Kluczową zaletą API lokalnych jest ich wysoka wydajność oraz możliwość optymalizacji pod kątem specyficznych potrzeb aplikacji, jednakże ich użycie jest zazwyczaj ograniczone do konkretnego systemu operacyjnego lub środowiska, co może utrudniać przenoszenie aplikacji między różnymi platformami.

Porównanie i zastosowania Zarówno API sieciowe, jak i lokalne odgrywają kluczową rolę w ekosystemie oprogramowania, jednak różnią się one pod względem zastosowań oraz środowiska, w którym operują. API sieciowe są idealnym rozwiązaniem dla aplikacji rozproszonych, które wymagają komunikacji z serwerami zdalnymi oraz integracji z usługami dostępnymi w Internecie. Przykłady obejmują aplikacje mobilne korzystające z chmury, systemy e-commerce czy platformy społecznościowe. Z drugiej strony, API lokalne są niezbędne w aplikacjach wymagających bezpośredniego dostępu do zasobów systemowych, takich jak oprogramowanie do edycji multimediów, gry komputerowe czy narzędzia do

zarządzania systemem. W praktyce, wiele nowoczesnych aplikacji wykorzystuje oba typy API, łącząc ich zalety w celu stworzenia kompleksowych i wydajnych rozwiązań. Na przykład, aplikacja mobilna może używać API sieciowych do synchronizacji danych z serwerem w chmurze, jednocześnie korzystając z API lokalnych do optymalizacji działania na urządzeniu użytkownika. Wybór odpowiedniego rodzaju API zależy od specyficznych potrzeb projektu, wymagań dotyczących wydajności oraz oczekiwań użytkowników końcowych.

Bezpieczeństwo i Wydajność Bezpieczeństwo stanowi kluczowy aspekt zarówno dla API sieciowych, jak i lokalnych, jednak podejścia do jego zapewnienia różnią się w zależności od środowiska. API sieciowe muszą być chronione przed atakami zewnętrznymi, takimi jak ataki DDoS, próby nieautoryzowanego dostępu czy wycieki danych. W tym celu stosuje się różnorodne mechanizmy zabezpieczeń, takie jak szyfrowanie danych, autoryzacja na poziomie tokenów czy firewalle aplikacyjne. API lokalne, choć mniej narażone na zagrożenia zewnętrzne, również wymagają zabezpieczeń, zwłaszcza w kontekście ochrony danych przechowywanych lokalnie oraz zapewnienia, że tylko uprawnione aplikacje mają dostęp do określonych zasobów systemowych. W zakresie wydajności, API lokalne zazwyczaj oferują szybszy dostęp do zasobów, ponieważ operacje odbywają się bezpośrednio na urządzeniu, eliminując opóźnienia związane z komunikacją sieciową. API sieciowe mogą natomiast być podatne na opóźnienia wynikające z przepustowości sieci oraz obciążenia serwerów, co wymaga optymalizacji zarówno po stronie klienta, jak i serwera, aby zapewnić płynne i responsywne działanie aplikacji.

Zarówno API sieciowe, jak i lokalne są nieodzownymi narzędziami w arsenale dewelopera, umożliwiającymi tworzenie zaawansowanych, skalowalnych i wydajnych aplikacji. Wybór odpowiedniego rodzaju API zależy od specyfiki projektu, jego wymagań oraz oczekiwań użytkowników, a ich umiejętne wykorzystanie może znacząco wpłynąć na sukces końcowego produktu.

Rysunek 4: Porównanie API sieciowego i lokalnego.

2.2 API Prywatne a API Publiczne

Innym podziałem API ze względu na ich widoczność i dostępność dla programistów jest podział na API publiczne i prywatne.

API Publiczne API publiczne jest udostępniane szerokiemu kręgowi użytkowników lub zewnętrznych aplikacji. Jego głównym celem jest umożliwienie integracji i współpracy z zewnętrznymi systemami. Firmy i organizacje tworzą publiczne API, aby umożliwić programistom lub partnerom dostęp do swoich danych, funkcji lub usług. Publiczne API zazwyczaj wymaga minimalnej autoryzacji, np. poprzez klucz API, który pozwala monitorować i kontrolować zużycie, ale nie jest wymagane szczegółowe sprawdzanie tożsamości użytkownika.

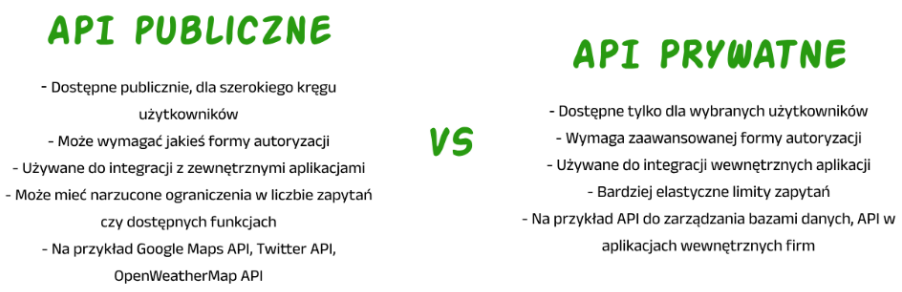
Zalety API publicznego

- **Szeroka dostępność**: Jest udostępniane publicznie, co pozwala na łatwą integrację z aplikacjami zewnętrznymi.
- **Zwiększenie zasięgu**: Umożliwia firmom dotarcie do szerszego kręgu użytkowników i partnerów.
- **Promowanie innowacji**: Pozwala zewnętrznym deweloperom na tworzenie aplikacji, które korzystają z dostępnych danych lub usług.

API Prywatne API prywatne jest dostępne tylko dla określonego kręgu użytkowników lub systemów, zazwyczaj w ramach jednej organizacji lub z wybranymi partnerami. Celem prywatnego API jest umożliwienie wewnętrznej komunikacji między różnymi systemami lub aplikacjami w obrębie jednej firmy. API prywatne jest bardziej zabezpieczone, ponieważ dostęp do niego jest ściśle kontrolowany, często za pomocą autoryzacji na poziomie użytkownika lub aplikacji. Zastosowanie takiego API jest bardziej zorientowane na wewnętrzne procesy, a dostęp jest ograniczony tylko do uprawnionych osób. Przykładem zastosowania prywatnego API może być komunikacja między systemami wewnętrznymi aplikacji a serwerem przechowującym dane, na przykład loginy i hasła użytkowników.

Zalety API prywatnego

- **Większe bezpieczeństwo**: Dostęp jest ograniczony, a komunikacja często odbywa się w zamkniętym środowisku.
- **Lepsza kontrola nad dostępem**: Dostęp do zasobów i usług jest ściśle regulowany i monitorowany.
- **Optymalizacja dla wewnętrznych procesów**: API prywatne służy do integracji systemów wewnętrznych organizacji, co poprawia efektywność operacyjną.



Rysunek 5: Porównanie API prywatnego i publicznego.

2.3 Architektury API

Architektury API definiują w jaki sposób komunikacje między programami są realizowane, obejmując m.in. metody wywołań, formaty danych, protokoły oraz strukturę zapytań i odpowiedzi. W zależności od wybranej architektury, API może stosować różne podejścia do obsługi zapytań, zarówno pod względem komunikacji, jak i efektywności. Istnieje kilka standardowych, powszechnie stosowanych architektur API wykorzystywanych w wszelkiego rodzaju aplikacjach. Z najpopularniejszych są to:

2.3.1 REST API

REST API (Representational State Transfer) to styl architektury, który opiera się na standardowych metodach HTTP, takich jak GET, POST, PUT i DELETE, do komunikacji między klientem a serwerem. REST jest lekki, elastyczny i łatwy w implementacji, co czyni go jednym z najczęściej używanych podejść w budowaniu interfejsów API. W architekturze REST każdy zasób, taki jak użytkownik, produkt czy zamówienie, jest identyfikowany za pomocą unikalnego adresu URL. Zapytania do serwera są realizowane przez wywołania HTTP, a dane przesyłane są zazwyczaj w formacie JSON lub XML. Jedną z głównych zalet REST API jest to, że jest bezstanowy, co oznacza, że każde zapytanie od klienta jest niezależne, a serwer nie przechowuje żadnych informacji o wcześniejszych zapytaniach. Ze względu na swoją prostotę, szeroką kompatybilność i łatwość integracji, REST API jest dominującym wyborem w aplikacjach internetowych, mobilnych i mikroserwisach.

2.3.2 RPC API

RPC API (Remote Procedure Call API) to styl architektury, który pozwala klientowi wywoływać funkcje lub procedury na serwerze, tak jakby były one lokalne. Klient wysyła zapytanie zawierające nazwę metody i parametry, a serwer przetwarza je i zwraca wynik. RPC API jest bardziej efektywne w systemach wymagających zaawansowanych

operacji i wysokiej wydajności, takich jak komunikacja między mikrousługami. W przeciwieństwie do REST API, które działa na zasobach identyfikowanych przez URL i używa standardowych metod HTTP (GET, POST, PUT, DELETE), RPC skupia się na wywoływaniu funkcji z określonymi parametrami. RPC może być mniej elastyczne w pracy z zasobami, ale jest bardziej odpowiednie dla scenariuszy, w których wymagana jest szybka wymiana danych i złożona logika.

2.3.3 SOAP API

SOAP API to protokół oparty na XML, który wymaga określonej, sztywnej struktury wiadomości. Dzięki temu zapewnia wysoki poziom bezpieczeństwa, transakcyjności i niezawodności, co czyni go idealnym rozwiązaniem w aplikacjach wymagających silnego zabezpieczenia, takich jak systemy bankowe czy korporacyjne. W przeciwieństwie do REST i RPC, SOAP jest bardziej skomplikowany i mniej elastyczny, ale zapewnia większe możliwości kontroli nad bezpieczeństwem i integralnością danych.

2.3.4 GraphQL

GraphQL to język zapytań opracowany przez Facebook, który umożliwia klientowi precyzyjne określenie, jakie dane chce otrzymać, eliminując problem nadmiarowości danych i redukując liczbę zapytań. W porównaniu do REST, GraphQL jest bardziej elastyczny, ponieważ pozwala na pobranie tylko tych danych, które są rzeczywiście potrzebne, w jednym zapytaniu. Jest to szczególnie przydatne w aplikacjach mobilnych i webowych, które muszą działać szybko i efektywnie. Jednak w przeciwieństwie do REST i RPC, GraphQL wymaga bardziej skomplikowanej konfiguracji na serwerze i jest trudniejszy w implementacji, zwłaszcza przy dużych i złożonych systemach.

Oprócz omówionych powyżej architektur istnieje wiele innych podejść do projektowania API, a także różne warianty i hybrydy tych rozwiązań dobierane w zależności od konkretnych potrzeb deweloperów.

3 Omówienie działania komunikacji HTTP: Zapytanie i Odpowiedź

Aby korzystać z API sieciowych, należy zrozumieć, jakie metody komunikacji są dostępne. API sieciowe, takie jak REST i RPC API, często opierają się na protokole HTTP, który umożliwia wymianę danych pomiędzy klientem a serwerem. Klient wysyła żądanie do serwera za pomocą konkretnych metod HTTP (np. GET, POST, PUT, DELETE w przypadku REST, czy specjalnych wywołań w RPC API), a serwer przetwarza to żądanie i zwraca odpowiedź. Odpowiedź zazwyczaj jest dostarczana w formie pliku w formacie JSON lub XML, chociaż formaty te mogą się różnić w zależności od architektury. W obu

przypadkach komunikacja odbywa się na zasobach, które są identyfikowane przez unikalne adresy URL (w przypadku REST API) lub adresy funkcji/procedur (w przypadku RPC API).

3.1 Metody HTTP

GET Metoda GET służy do pobierania zasobów z serwera. Jest to jedna z najczęściej używanych metod w REST API, gdyż pozwala na dostęp do zasobów, takich jak strony internetowe, pliki, dane użytkowników czy inne informacje przechowywane na serwerze. Przy użyciu metody GET nie zmienia się stan zasobu, czyli nie modyfikuje się danych, a jedynie je pobiera. Na przykład wysyłając zapytanie GET /users, gdzie users jest bazą danych trzymającą wszystkich użytkowników strony możemy dostać wszystkie rekordy z tej bazy danych.

POST POST to metoda wykorzystywana do tworzenia nowych zasobów na serwerze. W przeciwieństwie do metody GET, POST wysyła dane do serwera w celu zapisania ich, co zmienia stan aplikacji lub systemu. W kontekście REST API metoda POST jest powszechnie używana do tworzenia nowych rekordów, na przykład dodawania użytkowników do bazy danych. Żądanie POST zawiera dane w ciele zapytania (najczęściej w formacie JSON), które opisują nowy zasób. Na przykład, przy żądaniu POST /users można wysłać dane użytkownika, takie jak imię i e-mail, aby dodać go do systemu. Z kolei w przypadku RPC API, POST jest wykorzystywane do wywoływania dostępnych funkcji, a dane w ciele zapytania zawierają argumenty tych funkcji.

W kontekście REST API, metoda POST może być używana do przesyłania danych, które wpływają na ustawienia pamięci podręcznej (**cache**), podobnie jak w przypadku GET, ale z dodatkowymi opcjami konfiguracji. Gdy cache jest aktywne i wielokrotnie wysyłamy zapytanie GET, odpowiedź jest zapisywana w pamięci podręcznej po pierwszym przetworzeniu zapytania. Przy kolejnych identycznych zapytaniach GET, serwer zwraca tę samą odpowiedź z pamięci podręcznej, co przyspiesza dostęp do danych. Jednak w przypadku dynamicznie zmieniających się danych, cache może spowodować, że użytkownik nie otrzyma najnowszych informacji, ponieważ odpowiedź została zapisana w pamięci podręcznej po pierwszym wywołaniu GET, a nowe dane nie zostaną uwzględnione.

W odróżnieniu od metody GET, metoda POST może być używana do wysyłania specyficznych informacji, które modyfikują sposób przechowywania lub wygasania danych w cache, bez konieczności tworzenia nowych zasobów. Przykładowo, użytkownik może przesłać żądanie POST zawierające parametry, takie jak czas przechowywania danych w pamięci podręcznej czy włączenie/wyłączenie cachowania, co pozwala na dynamiczną konfigurację bez konieczności zmiany danych, a jedynie ich ustawień.

PUT PUT jest metodą, która służy do pełnej aktualizacji istniejącego zasobu. Kiedy klient wysyła zapytanie PUT, oczekuje, że serwer zaktualizuje zasób, który już istnieje. PUT zastępuje cały zasób nowymi danymi, które zostały przesłane w ciele żądania. Na przykład, zapytanie PUT /users/1 z danymi użytkownika może zastąpić wszystkie dotychczasowe dane użytkownika o ID 1 nowymi informacjami (np. imieniem, e-mailem, adresem).

PATCH PATCH to metoda wykorzystywana do częściowej aktualizacji zasobu. W przeciwieństwie do PUT, który wymaga przesłania pełnych danych, PATCH pozwala na modyfikację tylko wybranych pól w istniejącym zasobie. Jest to przydatne, gdy chcemy zaktualizować tylko jeden lub kilka atrybutów zasobu, nie zmieniając całego obiektu. Na przykład, zapytanie PATCH /users/1 z danymi dotyczącymi tylko e-maila użytkownika pozwala na zmianę wyłącznie tego atrybutu bez potrzeby przesyłania całych danych użytkownika.

DELETE DELETE to metoda, której używa się do usuwania zasobów z serwera. W przypadku zapytania DELETE serwer oczekuje, że dany zasób zostanie trwale usunięty. Na przykład, żądanie DELETE /users/1 usunie użytkownika o ID 1 z bazy danych. Po wykonaniu tej operacji zasób jest usuwany, a odpowiedź może zawierać jedynie potwierdzenie wykonania operacji lub kod statusu informujący o powodzeniu lub niepowodzeniu operacji.

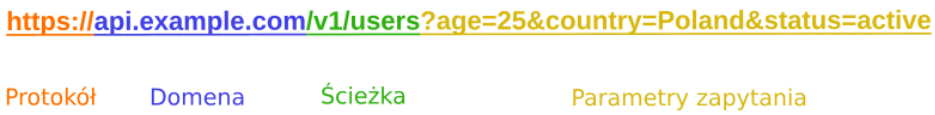
3.2 Adres URL

Aby wykonać odpowiednie zapytanie, nie wystarczy jedynie określenie metody. W przypadku REST API konieczne jest także zdefiniowanie zasobu, z którym chcemy wejść w interakcję, a w przypadku RPC API - adresu funkcji, która ma operować na zasobach. Jest to określane przy pomocy **adresu URL (Uniform Resource Locator)**, który ma swoją standardową budowę. Może składać się z kilku elementów, z których standardowymi są:

- **Protokół**: Określa, jakiego typu połączenie jest używane do komunikacji z zasobem (w przypadku REST i RPC API - http lub https).
- **Domena**: Jest to część adresu, która wskazuje na serwer lub usługę, na której zasób jest przechowywany.
- **Ścieżka**: Wskazuje konkretny zasób w obrębie serwera (na przykład na konkretną bazę danych czy miejsce wywołania funkcji).
- **Parametry zapytania**: Mogą zawierać dodatkowe informacje potrzebne do przetworzenia zapytania, takie jak filtry czy identyfikatory. Mogą one być opcjonalne lub obowiązkowe. Najczęściej informację o dostępnych parametrach zapytania znajdują się w dokumentacji danego API. Standardowo podawane są po znaku zapytania.

3.2.1 URL w REST API

W przypadku REST API URL podaje adres do zasobu. Zasób ten może obejmować szeroki zbiór danych, na przykład całą bazę danych, ale również pojedynczy wpis w tej bazie. Każdy zasób posiada unikalny identyfikator, który w połączeniu ze ścieżką dostępu tworzy adres URL. Rysunek 6 przedstawia przykładowy adres URL do zasobu.

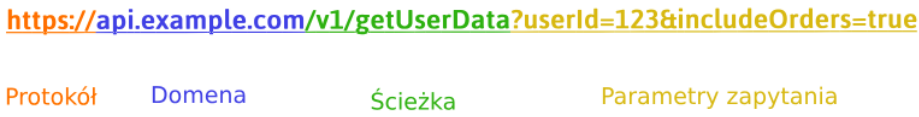


Rysunek 6: Przykładowy URL w REST API.

Powyższy URL wskazuje na użytkowników bazy users, dostępnej przez domenę api.example.com, którzy mają 25 lat, są z Polski i mają aktywny status.

3.2.2 URL w RPC API

W przypadku RPC API, które komunikuje się z bazą poprzez funkcje, adres URL wygląda podobnie, ale podaje ścieżkę do serwera i funkcję, jaka ma zostać wywołana. Funkcje są zdefiniowane przez konkretne API, a informacje o nich możemy znaleźć w dokumentacji. Rysunek 7 przedstawia przykładowy adres URL do funkcji.



Rysunek 7: Przykładowy URL w RPC API.

W powyższym przykładzie ścieżka wskazuje na funcję getUserData, a parametry zapytania są argumentami tej funkcji. Po wysłaniu powyższego URLa jako zapytania dostaniemy w odpowiedzi dane o użytkowniku o id 123 z dodatkowymi informacjami o jego zamówieniach (dzięki parametrowi includeOrders=true).

3.3 Wysyłanie zapytania

Adres URL często wystarczy do wysłania żądania GET, jednak inne metody HTTP jak na przykład POST potrzebują więcej informacji, w tym przypadku informacje o obiekcie, jaki . Takie informacje przekazywane są w ciele żądania.

Adres URL często wystarczy do wysłania żądania GET, jednak inne metody HTTP, takie jak POST, wymagają dodatkowych informacji, które są przekazywane w **ciele żą-**

dania. W przypadku POST są to dane dotyczące obiektu, który chcemy dodać lub zmodyfikować na serwerze. Ciało żądania zawiera te dane w określonym formacie, najczęściej w formacie JSON lub XML.

Listing 1: Przykładowe zapytanie POST z ciałem funkcji.

```
 POST /api/users HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "first_name": "Jan",
  "last_name": "Kowalski",
  "email": "jan.kowalski@example.com",
  "phone": "+48 123 456 789",
  "address": {
    "street": "ul. Pieciomorgowa 12",
    "city": "Warszawa",
    "postcode": "01-234"
  }
}
```

Dla innych metod HTTP w ciele żądania należy podać:

- **PUT** – w ciele żądania należy podać pełne dane, które mają zaktualizować dany rekord w bazie. Zwykle jest to pełna wersja obiektu, a nie tylko zmienione pola.
- **PATCH** – w ciele żądania należy podać tylko te dane, które mają zostać zmienione w konkretnym rekordzie bazy. Jest to metoda do częściowej aktualizacji zasobów.

W przypadku RPC API często parametry funkcji przesyła się właśnie w ciele żądania.

3.4 Odpowiedź serwera

Odpowiedź na zapytanie w API, zarówno w przypadku REST, jak i RPC, jest wynikiem przetworzenia żądania wysłanego przez klienta.

REST API W przypadku REST API, odpowiedź zwykle zawiera dane związane z żądanym zasobem, takie jak informacje o użytkowniku, produkcie czy zamówieniu. Odpowiedź jest zwracana w formacie JSON lub XML, w zależności od ustawień serwera, i zawiera informacje o statusie przetwarzania zapytania, zazwyczaj w postaci kodu HTTP (np. 200 OK dla powodzenia, 404 Not Found, 500 Internal Server Error). W odpowiedzi często przesyłane są także dodatkowe informacje o zasobach, takie jak linki do powiązanych zasobów lub informacje o następnych stronach wyników w przypadku paginacji.

Listing 2: Przykładowa odpowiedź REST API dla metody GET.

```
{
```

```
    "status": "success",
    "data": {
        "user_id": 123,
        "first_name": "Jan",
        "last_name": "Kowalski",
        "email": "jan.kowalski@example.com",
        "phone": "+48␣123␣456␣789",
        "address": {
            "street": "ul.␣Pieciomorgowa␣12",
            "city": "Warszawa",
            "postcode": "01-234"
        },
        "registered_at": "2022-05-15T12:34:56Z"
    },
    "message": "Dane␣uzytkownika␣zostaly␣pobrane␣pomyslnie."
}
```

RPC API W RPC API odpowiedź jest zbliżona, ale często bardziej zależna od wywołanej funkcji. Zamiast zwracać dane zasobów w sposób ogólny, jak ma to miejsce w REST, RPC API może zwrócić bardziej specyficzne dane wynikowe, które zależą od wykonywanej operacji, np. wyniki obliczeń, status wykonania procedury lub wynik wywołania określonej funkcji. Tak jak w przypadku REST, odpowiedzi są najczęściej dostarczane w formacie JSON lub XML, ale mogą również mieć inne struktury, zależnie od zastosowanej specyfikacji. W przypadku RPC, odpowiedź może zawierać zarówno dane, jak i informacje o błędach związanych z wykonaniem danej procedury lub funkcji.

Listing 3: Przykładowa odpowiedź RPC API dla wywołania funkcji getUserData.

```
{
    "jsonrpc": "2.0",
    "method": "getuserdata",
    "params": {
        "user_id": 123,
        "includeorders": true
    },
    "result": {
        "status": "success",
        "data": {
            "user_id": 123,
            "first_name": "Jan",
            "last_name": "Kowalski",
            "email": "jan.kowalski@example.com",
            "phone": "+48␣123␣456␣789",
            "address": {
                "street": "ul.␣Pieciomorgowa␣12",
                "city": "Warszawa",
                "postcode": "01-234"
            },
```

```
            "registered_at": "2022-05-15T12:34:56Z",
            "orders": [
                {
                    "order_id": 1,
                    "order_date": "2024-12-01T10:30:00Z",
                    "total_amount": 199.99,
                    "items": [
                        {
                            "item_id": 101,
                            "name": "Laptop",
                            "quantity": 1,
                            "price": 199.99
                        }
                    ]
                }
            ]
        },
        "message": "Dane␣uzytkownika␣z␣zamowieniami␣zostaly␣pobrane␣
           pomyslnie."
    },
    "id": 1
}
```


W przypadku RPC API w odpowiedzi otrzymujemy informacje o wywołanej funkcji (pole "method") oraz przekazanych parametrach (pole "params"). Pole "result"zawiera dane zwrócone przez wykonaną funkcję. Funkcja getUserData działa podobnie do metody GET w REST API, jednak w przeciwieństwie do niej, może pobierać dane z wielu różnych źródeł (np. z wielu baz danych) i łączyć je w jedną odpowiedź.

3.5 Autoryzacja użytkownika API

Autoryzacja w API to proces weryfikacji tożsamości użytkownika lub aplikacji, który pozwala na kontrolowanie dostępu do zasobów i usług udostępnianych przez API. W przypadku API istnieje kilka metod autoryzacji, gdzie każda ma swoje zastosowanie w różnych scenariuszach i poziomach bezpieczeństwa. Przykładowymi metodami autoryzacji są:

- **Autoryzacja podstawowa** - czyli wysyłanie loginu i hasła w nagłówku zapytania HTTP. Dzięki temu zalogowani użytkownicy strony mają dostęp tylko do danych związanych z ich profilem, a nie do danych każdego użytkownika strony.

- **Token nosiciela** - to popularna metoda autoryzacji, w której serwer API przyznaje klientowi token dostępu, zwykle po zalogowaniu lub autoryzacji użytkownika. Ten token jest następnie przesyłany w nagłówku zapytania HTTP jako "Bearer token". Token może mieć określony czas życia (np. godzina) i ograniczać ilość dostępnych zapytań.

Przykładem tej formy autoryzacji jest dostęp do nowych modeli chatu GPT, gdzie na dany czas dostępna jest ograniczona liczba zapytań.

- **API Key (Klucz API)** - metoda autoryzacji, w której użytkownik lub aplikacja otrzymuje unikalny klucz dostępu, który należy dołączyć do zapytania API (zwykle jako parametr w URL lub nagłówek). Klucz API jest przypisany do konta i może być wykorzystywany do ograniczania liczby zapytań lub monitorowania aktywności użytkownika.

- **Session Cookies (Ciastka sesyjne)** - Sesyjne ciasteczka są używane do przechowywania informacji o użytkowniku w czasie jego sesji w aplikacji webowej. Gdy użytkownik loguje się na stronie, serwer wysyła ciasteczko, które następnie jest przesyłane z każdym kolejnym zapytaniem HTTP. Sesyjne ciasteczka są wykorzystywane głównie w aplikacjach webowych, ale mogą być również używane do autoryzacji w API, szczególnie w aplikacjach, które działają w obrębie tej samej domeny.

4 Strona api.um.warszawa.pl

Strona api.um.warszawa.pl oferuje bogaty zbiór baz danych związanych z funkcjonowaniem Warszawy. Udostępnia publiczne API wraz z dokumentacją, umożliwiając łatwe wykorzystanie tych zasobów. Wśród dostępnych danych znajdują się m.in. informacje o aktualnym położeniu autobusów, które są aktualizowane co minutę, kalendarz wydarzeń w Warszawie, rozmieszczenie aptek, obiektów sportowych, noclegów, miejsc parkingowych i wiele więcej. Bazy danych na stronie są systematycznie aktualizowane, a z czasem pojawiają się nowe zasoby, co czyni je wartościowym źródłem informacji o mieście.

Rysunek 8: Strona startowa api.um.warszawa.pl (https://api.um.warszawa.pl, dostęp: 09.12.2024).

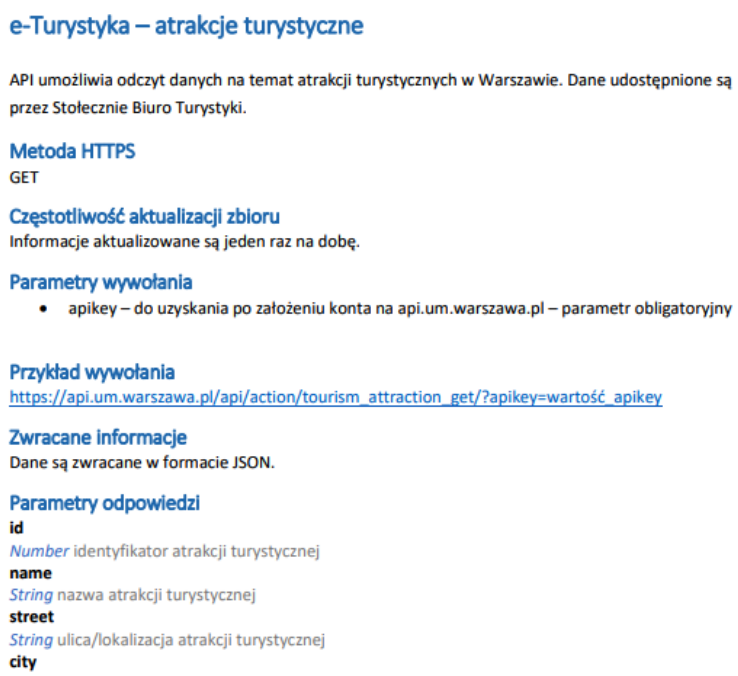
4.1 CKAN API

Bazy danych oferowane przez stronę są dostępne za pośrednictwem CKAN API. Jest to API sieciowe, które jest podobne działaniem do RPC API. Tak samo jak w przypadku RPC, CKAN API komunikuje się z serwerem poprzez wywoływanie konkretnych funkcji lub operacji, które serwer interpretuje i wykonuje.

4.2 Jak korzystać ze strony

Po wejściu na stronę i kliknięciu w przycisk "Dostępne dane"przedstawione zostają kategorie dostępnych baz danych. Po kliknięciu w interesującą nas kategorię możemy wybrać konkretną bazę danych. Dla każdej bazy dostępna jest dokumentacja umożliwiająca wysyłanie żądań do tej bazy.

Przykład dokumentacji bazy "Atrakcje turystyczne"



e-Turystyka – atrakcje turystyczne

API umożliwia odczyt danych na temat atrakcji turystycznych w Warszawie. Dane udostępnione są przez Stołecznie Biuro Turystyki.

Metoda HTTPS
GET

Częstotliwość aktualizacji zbioru
Informacje aktualizowane są jeden raz na dobę.

Parametry wywołania
- apikey – do uzyskania po założeniu konta na api.um.warszawa.pl – parametr obligatoryjny

Przykład wywołania
https://api.um.warszawa.pl/api/action/tourism_attraction_get/?apikey=wartość_apikey

Zwracane informacje
Dane są zwracane w formacie JSON.

Parametry odpowiedzi
id
Number identyfikator atrakcji turystycznej
name
String nazwa atrakcji turystycznej
street
String ulica/lokalizacja atrakcji turystycznej
city

Rysunek 9: Początek dokumentacji bazy danych *Atrakcje turystyczne*.

- **Metoda HTTPS** - mówi za pośrednictwem jakich metod HTTP można komunikować się z bazą. W przypadku baz danych oferowanych przez Urząd Miasta Warszawy w większości przypadków jest to wyłącznie metoda GET, chociaż niektóre bazy oferują również metodę POST, w celu ustawienia Catche.

- **Częstotliwość aktualizacji zbioru** - co jaki czas baza jest aktualizowana. W tym przypadku jeśli chcemy mieć aktualne dane to należy pobierać dane raz na dobę.

- **Parametry wywołania** - jakich parametrów musimy (bądź możemy) użyć w parametrach zapytania przy przesyłaniu żądania. Dla Warszawskiej bazy danych bardzo często parametrem obligatoryjnym jest API key.

- **Przykład wywołania** - jest to przykładowe zapytanie, najczęściej w formie adresu URL. Dzięki przykładowi wywołania możemy poznać ścieżkę do funkcji.
- **Zwracane informacje** - informacja o formacie odpowiedzi (najczęściej JSON).
- **Parametry odpowiedzi** - określają, jakie informacje są zwracane przez serwer dla pojedynczego obiektu.

Przykład odpowiedzi


```
{
  "result": [
  {
    "id": 2183,
    "name": "Muzeum Warszawskiej Pragi",
    "street": "ul. Targowa 50/52, Praga Północ",
    "city": "Warszawa",
    "status": "otwarte",
    "tickets": "wstęp bezpłatny: czwartek\r\nAktualne informacje dostępne
na stronie internetowej: https://muzeumpragi.pl/",
    "hours": "wtorek: 9.00-17.00\r\nśroda: 9.00-17.00\r\nczwartek: 10.00-
18.00\r\npiątek: 9.00-17.00\r\nsobota - niedziela: 10.00-18.00",
    "amenities": {
      "hearing": null,
      "vision": null,
      "mobility": null,
      "polish_sign_language": null
    },
    "description": "Muzeum Pragi gromadzi pamiątki historyczne związane z
dziejami prawobrzeżnej Warszawy. Część z nich to przedmioty, które na rzecz
muzeum przekazali sami prażanie. Wewnątrz obejrzysz makietę XVIII-wiecznej
Pragi, która w tym okresie była oddzielnym miastem i wysłuchasz wywiadów z
najstarszymi mieszkańcami. Szczególną atrakcją są eksponaty i multimedia
związane z praskimi bazarami i targami. W oficynie jednego z budynków
muzeum zwróć uwagę na malowidła ścienne zdobiące dawny żydowski dom
modlitwy. Są one zabytkiem unikatowym w skali Warszawy i całego
Mazowsza. ",
    "photo": {
      "alt": null,
      "desc": "Muzeum Warszawskiej Pragi",
      "copyright": "Archiwum m.st. Warszawy",
      "thumb": "https://wawastorage.s3.eu-central-
1.amazonaws.com/photos/shares/thumbs/Muzeum-Warszawskiej-Pragifot-mst-
Warszawa-1.JPG",
      "author": null,
      "url": "https://wawastorage.s3.eu-central-
1.amazonaws.com/photos/shares/Muzeum-Warszawskiej-Pragifot-mst-Warszawa-
1.JPG"
    },
    "www": "https://muzeumpragi.pl",
    "category": "muzea",
    "additional_categories": [],
    "latlng": {
      "lat": "52.2516126",
      "lng": "21.0392634"
    }
  }
    ]
  }
```


Rysunek 10: Przykład odpowiedzi z dokumentacji bazy danych *Atrakcje turystyczne*

- **Przykład odpowiedzi** - pokazuje, jak wygląda odpowiedź na przykładowe zapytanie. Dzięki temu możemy zobaczyć strukturę zwracanych danych (np. że interesujące nas informacje znajdują się w polu "result"). Jest to przydatne, aby wiedzieć, jak odczytać dane i z nich korzystać.

4.3 Specyfika innych dostępnych baz

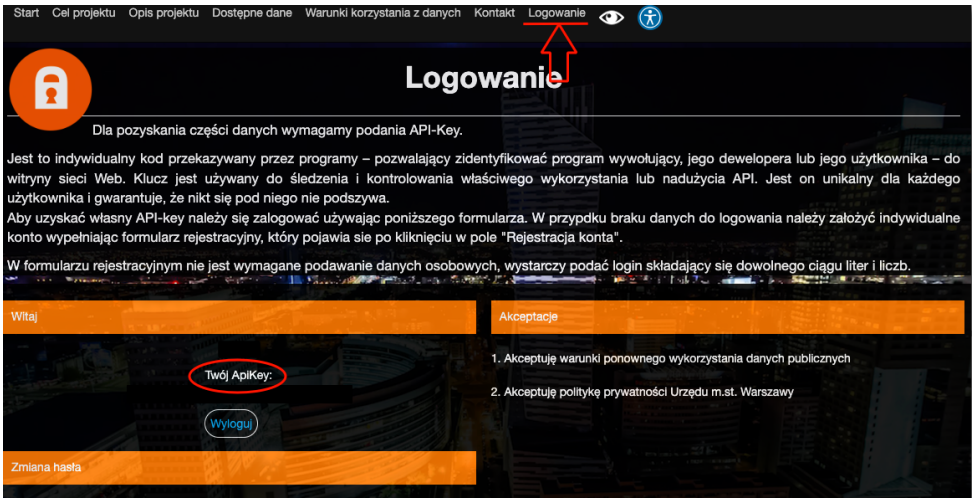
Strona api.um.warszawa.pl udostępnia również bardziej wyspecjalizowane bazy danych, które mogą wymagać szczegółowego omówienia:

- **Mapy wektorowe** – bazy danych zawierające obiekty przypisane do konkretnych współrzędnych geograficznych. Podczas pobierania danych z takich baz konieczne jest określenie obszaru geograficznego, z którego chcemy uzyskać informacje.

- **Bazy online** – dynamiczne bazy danych aktualizowane znacznie częściej niż standardowe (np. co minutę). W takich przypadkach często stosuje się metodę POST zamiast GET, aby umożliwić konfigurację ustawień pamięci podręcznej (cache) lub dostosowanie zapytania do specyficznych potrzeb użytkownika.

4.4 Generowanie klucza API

Przy niektórych bazach danych obligatoryjnym parametrem do wywołania GET jest klucz API. Klucz API do warszawskich baz danych można wygenerować, logując się na stronę. Po zalogowaniu klucz API jest niezmienny i znajduje się w zakładce "Logowanie". Przy wykonywaniu żądań należy go przekopiować do parametrów żądania w miejsce "apikey=".



Rysunek 11: Gdzie się znajduję klucz API (https://api.um.warszawa.pl, dostęp: 09.12.2024).

5 Przykład korzystania z API za pośrednictwem przeglądarki

Za pośrednictwem przeglądarki możemy przesłać zapytanie GET, które nie ma ciała żądania (jest samym adresem URL). Wystarczy wkleić URL w pasek przeglądarki. Na przykład można skorzystać z przykładów wywołania z dokumentacji.

W naszym przykładzie będziemy chcieli uzyskać dostęp do bazy "Wydarzenia kalendarz"z kategorii "Dane urzędowe", aby wypisać wszystkie wydarzenia w Warszawie.

Wydarzenia w Warszawie

API umożliwia odczyt zaplanowanych wydarzeń w Warszawie.

Identyfikator zasobu (resource_id)
resource_id = fbf95b7b-ff95-48bc-afda-a16e968cc9a6

Metoda HTTPS
GET

Częstotliwość aktualizacji zbioru
Informacje aktualizowane są kilka razy dziennie.

Parametry wywołania

- resource_id – identyfikator zasobu (uwaga, wartość id podana w dokumentacji może być przykładowa – poprawne identyfikatory zasobu przydzielone do poszczególnych zasobów zawiera portal api.um.warszawa.pl) – parametr obligatoryjny
- apikey – do uzyskania po założeniu konta na api.um.warszawa.pl - parametr obligatoryjny

Przykład wywołania
https://api.um.warszawa.pl/api/action/events_calendar?id=fbf95b7b-ff95-48bc-afda-a16e968cc9a6&apikey=wartość_apikey

Zwracane Informacje
Dane są zwracane w formacie JSON.

Rysunek 12: Dokumentacja bazy "Wydarzenia kalendarz".

Skorzystamy z przykładu wywołania. W miejsce apikey należy przekopiować wygenerowany klucz API.

Po przekopiowaniu linku w pasek przeglądarki w odpowiedzi dostajemy:


```
{
  "result": [
    {
      "category": [
        {
          "id": "34057",
          "name": "Kultura"
        }
      ],
      "language": "pl_PL",
      "lead": "Po␣sukcesie␣2.␣edycji␣Festiwalu␣Teatralnego␣Wolnosc,␣
         potwierdzajacym␣ogromne␣zainteresowanie␣teatrem␣wsrod␣mlodziezy
```

```
        ␣␣Muzeum␣Powstania␣Warszawskiego␣zaprasza␣do␣udzialu␣w␣3.␣
        edycji␣ogolnopolskiego␣festiwalu␣najlepszych␣spektakli␣
        tworzonych␣przez␣uczniow␣szkol␣ponadpodstawowych",
      "localization": [
        {
          "id": "33978",
          "name": "Wola"
        }
      ],
      "url": "https://kultura.um.warszawa.pl/-/1944-festiwal-wolnosc",
      "text": "\u003Cp\u003ETematem␣festiwalu jest␣walka␣o␣wolnosc␣
        w␣kontekstach␣zarowno␣historycznych,␣jak␣i␣wspolczesnych.\u003C
        /p\u003E\n\u003Cp\u003EDo␣28␣lutego␣2025␣roku␣nalezy␣nagrac␣
        spektakl␣i␣wyslc␣go␣do␣Muzeum.␣Najlepsze␣przedstawienia␣zostana
        ␣zaprezentowane␣8,␣9␣i␣10␣kwietnia␣2025␣roku␣w␣Muzeum␣Powstania
        ␣Warszawskiego.␣Zwycieski␣spektakl␣otrzyma␣nagrode␣finansowa␣w␣
        wysokosci␣5000␣zl␣oraz␣nagrody␣rzeczowe␣dla␣tworcow␣
        przedstawienia.\u003C/p\u003E",
      "image": {
        "highlighted": {
          "path": "http://um.warszawa.pl/documents/65745/120822017/wolno
            %C5%9B%C4%87+wprow.png"
        },
        "small": {
          "path": "http://um.warszawa.pl/documents/65745/120822017/wolno
            %C5%9B%C4%87+min.png"
        }
      },
      "title": "3.␣␣Festiwal␣Teatralny␣Wolnosc",
      "availableLanguages": [
        "pl_PL"
      ],
    ...
```


W taki sposób możemy mieć łatwy wgląd do bazy danych, jednak jeśli chcemy wykonywać bardziej złożone operacje, to musimy skorzystać z innych narzędzi, umożliwiających tworzenie i testowanie zapytań API, czy obróbkę i wizualizację otrzymanych danych.

6 Python i API w Pythonie

W kolejnej części poradnika nauczymy się, jak w praktyce korzystać z API. Konieczne jest do tego użycie języka programowania, ponieważ sami będziemy pisać programy, które będą się komunikować z różnymi interfejsami API. W tym poradniku postanowiliśmy użyć Pythona ze względu na jego cechy przydatne do tego celu:

- **Prosty i intuicyjny**: Python charakteryzuje się czytelną i zwięzłą składnią, co sprawia, że jest łatwy do nauki nawet dla osób początkujących w programowaniu. Intuicyjne podejście pozwala szybko zrozumieć podstawowe koncepcje i rozpocząć tworzenie własnych projektów.

- **Szeroka dostępność materiałów i pomocy naukowych**: Python posiada ogromną społeczność oraz bogatą bazę zasobów edukacyjnych, takich jak kursy online, poradniki, książki czy fora dyskusyjne. Dzięki temu łatwo znaleźć wsparcie i materiały do nauki, które pomogą w rozwiązywaniu napotkanych problemów.

- **Uniwersalny w zastosowaniach**: Python jest wykorzystywany w wielu dziedzinach, takich jak web development, analiza danych, sztuczna inteligencja, automatyzacja czy tworzenie aplikacji desktopowych. Ta wszechstronność umożliwia zastosowanie nabytych umiejętności w różnorodnych projektach i branżach.

- **Mnogość dostępnych bibliotek programistycznych**: Python oferuje bogaty ekosystem bibliotek programistycznych i frameworków, które znacznie ułatwiają pracę z API oraz inne zadania programistyczne. Biblioteki takie jak `requests` do obsługi HTTP, `NumPy` która dostarcza potężne narzędzia do obliczeń numerycznych i manipulacji dużymi zbiorami danych, czy `Pandas` do analizy danych, pozwalają na szybkie i efektywne tworzenie zaawansowanych rozwiązań bez konieczności pisania wszystkiego od podstaw.

- **Po poznaniu podstaw łatwo przejść do profesjonalnych zastosowań**: Opanowanie fundamentów Pythona otwiera drzwi do bardziej zaawansowanych i specjalistycznych zastosowań. Dzięki temu możliwe jest uczestnictwo w skomplikowanych projektach technologicznych oraz rozwój kariery w różnych sektorach IT, takich jak data science, machine learning czy inżynieria oprogramowania.

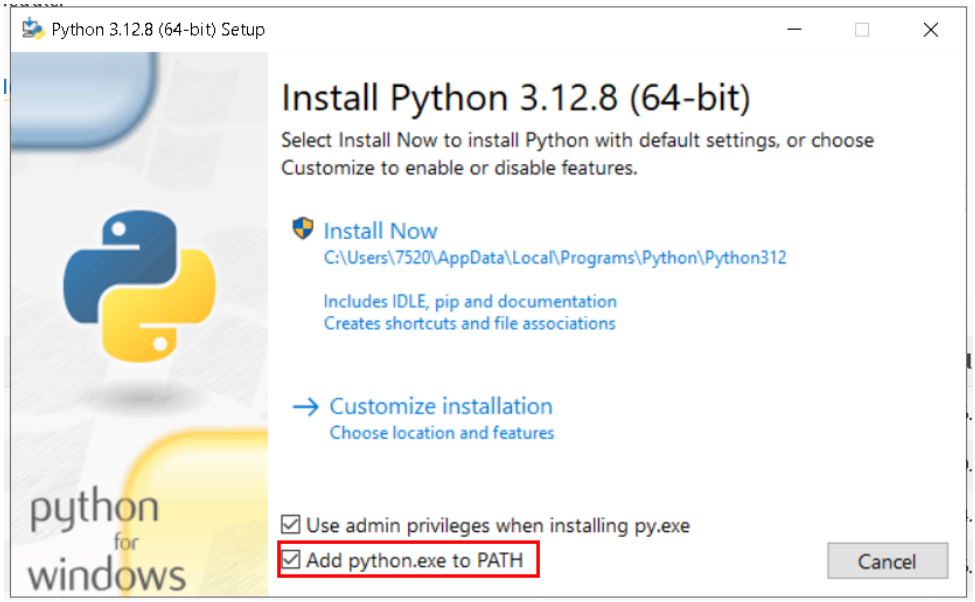
Wybór Pythona jako języka programowania w naszym poradniku zapewnia solidne podstawy oraz elastyczność potrzebną do efektywnego korzystania z API i dalszego rozwoju w świecie programowania.

6.1 Instalacja Pythona, IDE, Konfiguracja środowiska

W tej części poradnika zostanie omówiona przykładowa instalacja potrzebnych narzędzi na systemie **Windows (x64)**. Konieczne będą do tego uprawnienia administratora. Instalacja dla innych systemów przebiega analogicznie. Od uruchomienia Jupyter Notebook praca w różnych środowiskach się nie różni i procedura jest uniwersalna dla wszystkich.

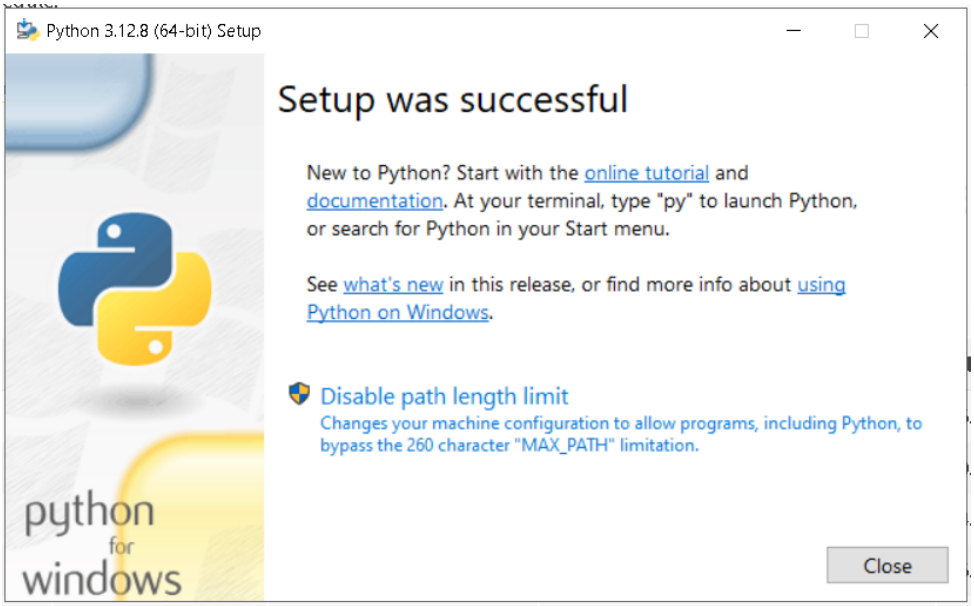
Instalacja Python Przejdź na oficjalną stronę Python:
https://www.python.org/downloads/.
Wybierz odpowiednią wersję (dla potrzeb tego poradnika korzystamy z Pythona w wersji **3.12.x**).
Aktualną wersją jest **3.12.8**, dostępna pod adresem:
https://www.python.org/downloads/release/python-3128/. Pobierz instalator właściwy dla swojego systemu operacyjnego.

Następnie uruchom pobrany plik `python-3.12.8-amd64.exe`, zobaczymy instalator (Rys. 13):



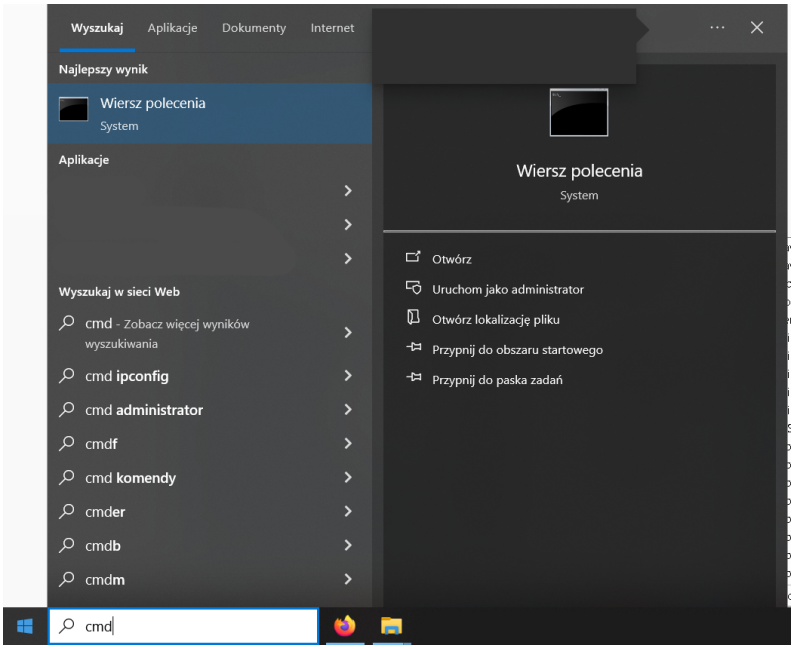
Rysunek 13: Instalator Pythona. Uwaga: zaznacz opcję *Add Python to PATH*, aby uniknąć problemów z dostępem do Pythona w terminalu.

Po zakończeniu instalacji powinniśmy zobaczyć komunikat o pomyślnym jej przebiegu (Rys. 14):



Rysunek 14: Instalator Pythona po pomyślnej instalacji.

Weryfikacja instalacji: Otwórz wiersz poleceń (w menu Start wpisz `cmd`) (widoczne na Rys. 15):

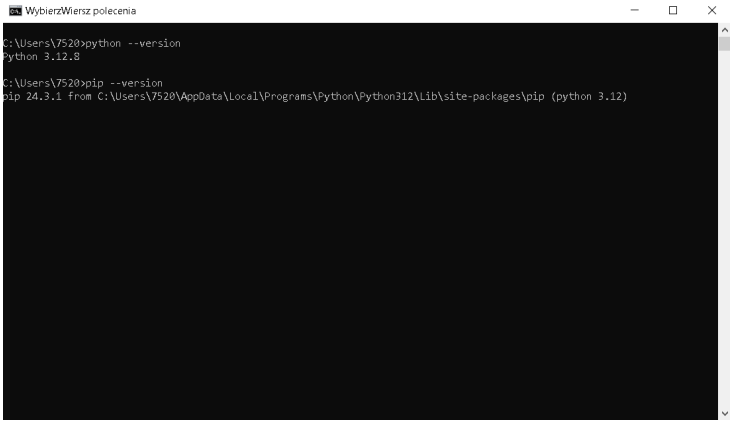


Rysunek 15: Uruchamianie wiersza poleceń w systemie Windows.

W wierszu poleceń wpisz:

```
python --version
pip --version
```

Jeśli wszystko przebiegło pomyślnie, zobaczysz informacje o zainstalowanej wersji Pythona i PIP-a (Rys. 16):



Rysunek 16: Weryfikacja instalacji Pythona i PIP w wierszu poleceń.

Instalacja bibliotek: Aby zainstalować niezbędne biblioteki, wpisz:

```
pip install numpy matplotlib pandas requests
```

Instalacja Jupyter Notebook Aby zainstalować Jupyter Notebook, wykonaj polecenie:

```
pip install notebook
```

Po zakończonej instalacji powinna pojawić się informacja o skutecznym zainstalowaniu wymaganych pakietów (Rys. 17):

Rysunek 17: Wiersz poleceń po zainstalowaniu Jupyter Notebook. Zawiera informacje o skutecznej instalacji i wyświetla raport z jej przebiegu.

Aby uruchomić Jupyter Notebook, wpisz:

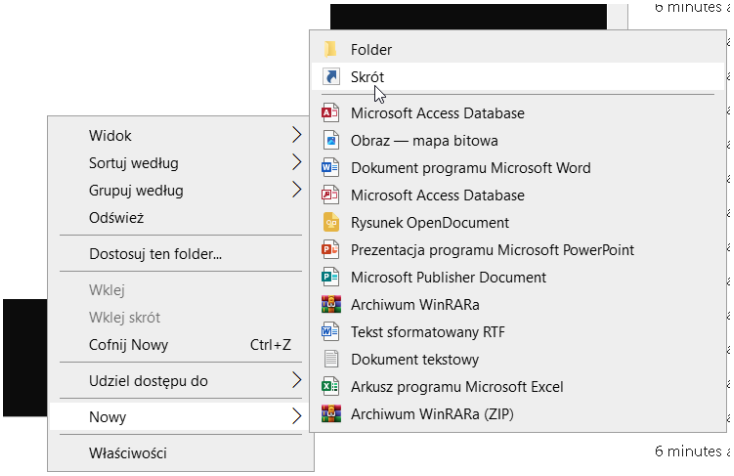
```
jupyter notebook
```

Rysunek 18: Wiersz poleceń - wyświetlenie ścieżki do Jupyter Notebook. Zaznacz i skopiuj tę ścieżkę (Enter).

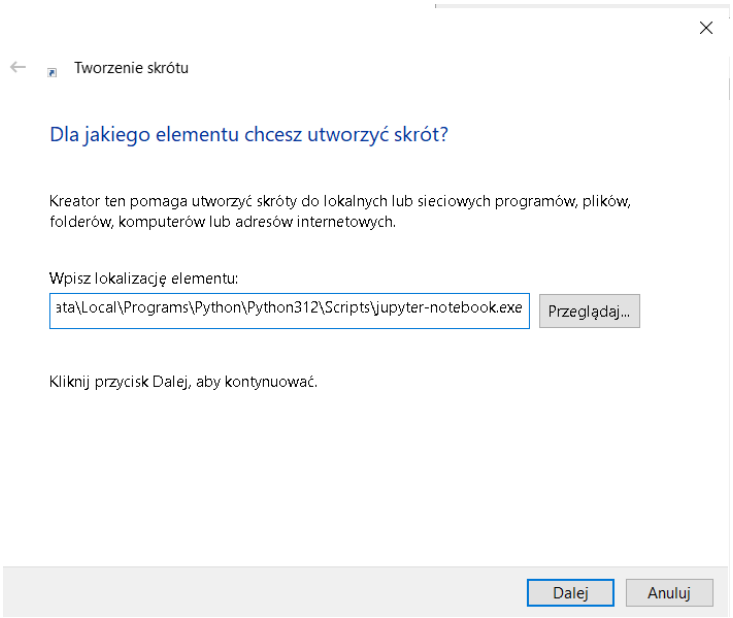
Tworzenie skrótu do Jupyter Notebook: Aby móc uruchamiać Jupyter Notebook bezpośrednio (bez użycia wiersza poleceń), najpierw sprawdź, gdzie znajduje się plik uruchamiający. W wierszu poleceń wykonaj komendę:

```
where jupyter-notebook
```

Wyświetli się ścieżka do pliku `jupyter-notebook.exe` (Rys. 18). Następnie na pulpicie lub w wybranym katalogu kliknij prawym przyciskiem myszy i wybierz: *Nowy -> Skrót*. Wklej skopiowaną ścieżkę w polu *Element docelowy* (Rys. 19).

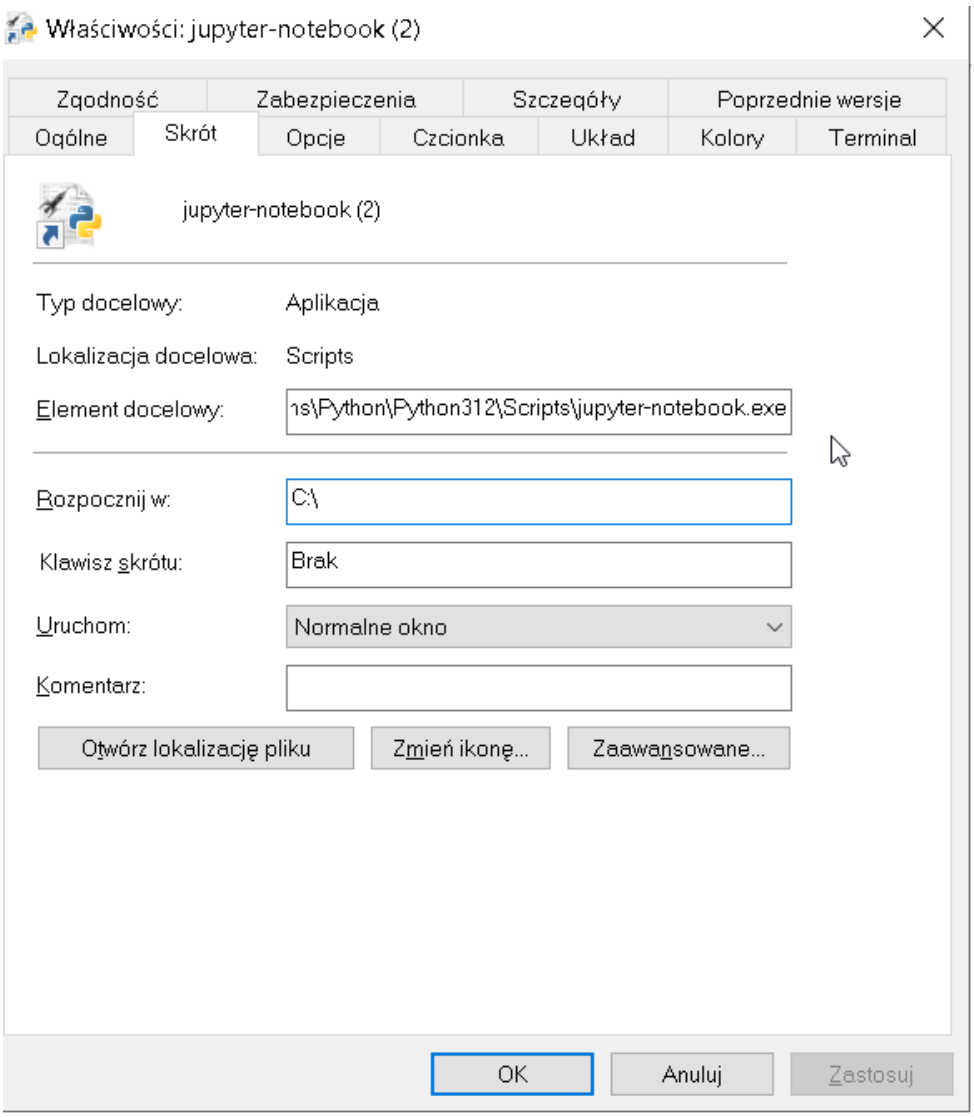


Rysunek 19: Tworzenie skrótu do Jupyter Notebook.



Rysunek 20: Okno tworzenia skrótu, należy tu wkleić skopiowaną wcześniej scieżkę do `jupyter-notebook.exe`.

Po utworzeniu skrótu, można zmienić katalog startowy, w którym będzie uruchamiany Jupyter Notebook. Aby to zrobić, kliknij prawym przyciskiem myszy na skrót (Rys. 21), wybierz *Właściwości* i w polu *Rozpocznij w:* wprowadź np. `C:\`, jeśli chcesz, by Notebook otwierał się w katalogu głównym dysku C.



Rysunek 21: Zmiana katalogu startowego w właściwościach skrótu.

6.2 Wprowadzenie do Pythona

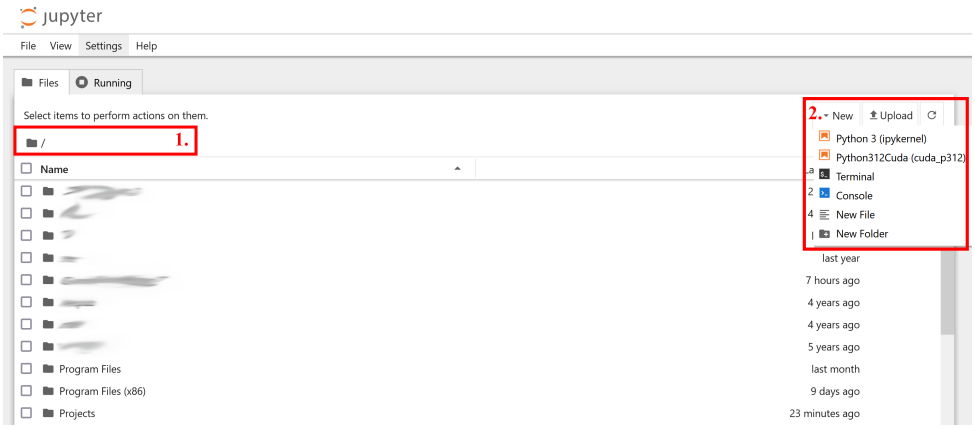
6.2.1 Obsługa Jupyter Notebook

Jupyter Notebook to otwarte, interaktywne środowisko obliczeniowe, które umożliwia tworzenie i udostępnianie dokumentów zawierających kod źródłowy, tekst, wykresy oraz wizualizacje danych.

Po uruchomieniu Jupyter Notebook pojawia się następujący widok:

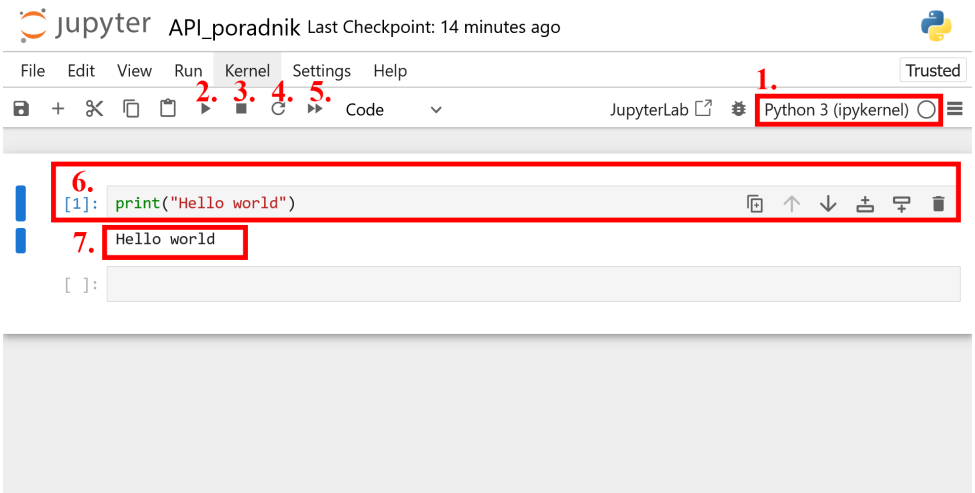
1. **Ścieżka katalogu** – widoczna jest ścieżka katalogu, w którym uruchomiono Jupyter Notebook. Domyślnie otwiera się w katalogu użytkownika.

2. **Nowy Notebook** – kliknięcie tej opcji pozwala na utworzenie nowego pliku, w którym można pisać i wykonywać kod.



Rysunek 22: Widok Jupyter Notebook po uruchomieniu.

Po utworzeniu nowego notatnika i przejściu do niego widzimy następujący widok:



Rysunek 23: Widok po otwarciu nowego notatnika w Jupyter Notebook.

1. **Kernel** – umożliwia wybór interpretera Pythona, w którym wykonywany jest napisany kod.

2. **Komórka kodu** – obszar, w którym piszemy kod. Po kliknięciu w komórkę widoczny jest migający kursor, co oznacza, że jest ona aktywna.

3. **Uruchamianie komórki** – ikona powodująca wykonanie kodu w aktywnej komórce. Skrót klawiaturowy: `Shift + Enter`.

4. **Przerwanie kodu** – ikona zatrzymująca wykonywanie aktualnie uruchomionego kodu.

5. **Restart kernela** – pozwala zrestartować środowisko obliczeniowe.

6. **Restart i uruchomienie wszystkich komórek** – restartuje kernel, a następnie wykonuje wszystkie komórki w kolejności.

7. **Wyniki wykonania** – po uruchomieniu komórki wynik działania programu wyświetla się poniżej, o ile komórka zwraca wynik.

6.3 Twoje Pierwsze Kroki w Pythonie

Zacznijmy od stworzenia naszego pierwszego programu w Pythonie, który wyświetli tekst "Witaj, świecie!".

W nowo otwartym notatniku, wpisz poniższy kod w pierwszej komórce:

Listing 4: Twój pierwszy program w Pythonie

```
print("Witaj, świecie!")
```

Naciśnij `Shift + Enter`, aby uruchomić komórkę. Powinieneś zobaczyć następujący wynik poniżej komórki:

```
Witaj, świecie!
```

Gratulacje! Właśnie napisałeś i uruchomiłeś swój pierwszy program w Pythonie.

Zmienne i Typy Danych Zmienne służą do przechowywania danych, które mogą być używane i modyfikowane w programie. W Pythonie nie musisz deklarować typu zmiennej przed jej użyciem – typ jest dynamicznie przypisywany.

Listing 5: Deklaracja zmiennych w Pythonie

```
liczba = 10          # Liczba całkowita (int)
tekst = "Python"     # Łańcuch znaków (str)
zmiennoprzecinkowa = 3.14  # Liczba zmiennoprzecinkowa (float)
```

Typy Danych

- **int:** Liczby całkowite, np. `5`, `-3`
- **float:** Liczby zmiennoprzecinkowe, np. `3.14`, `-0.001`
- **str:** Łańcuchy znaków, np. `"Hello"`, `'Python'`
- **bool:** Wartości logiczne, `True` lub `False`

Listing 6: Przykład zmiennych i ich wyświetlania

```
# Przykład zmiennych
wiek = 25
temperatura = 23.5
imie = "Anna"
jest_studentem = True
```

```
# Wyświetlanie zmiennych
print(wiek)                # 25
print(temperatura)         # 23.5
print(imie)                # Anna
print(jest_studentem)      # True
```

Typowanie Zmiennych Możesz sprawdzić typ zmiennej za pomocą funkcji `type()`.

Listing 7: Sprawdzanie typu zmiennych

```
print(type(wiek))            # <class 'int'>
print(type(temperatura))     # <class 'float'>
print(type(imie))            # <class 'str'>
print(type(jest_studentem))  # <class 'bool'>
```

Operatory Operatory służą do wykonywania operacji na zmiennych i wartościach.

Operatory Arytmetyczne

- **Dodawanie:** `+`
- **Odejmowanie:** `-`
- **Mnożenie:** `*`
- **Dzielenie:** `/`
- **Dzielenie całkowite:** `//`
- **Reszta z dzielenia:** `%`
- **Potęgowanie:** `**`

Listing 8: Operatory arytmetyczne w Pythonie

```
a = 10
b = 3

print(a + b)   # 13
print(a - b)   # 7
print(a * b)   # 30
print(a / b)   # 3.3333333333333335
print(a // b)  # 3
print(a % b)   # 1
print(a ** b)  # 1000
```

Operatory Porównania

- **Równe:** `==`
- **Różne:** `!=`
- **Większe:** `>`
- **Mniejsze:** `<`
- **Większe lub równe:** `>=`
- **Mniejsze lub równe:** `<=`

Listing 9: Operatory porównania w Pythonie

```
x = 5
y = 10

print(x == y)  # False
print(x != y)  # True
print(x > y)   # False
print(x < y)   # True
print(x >= y)  # False
print(x <= y)  # True
```

Operatory Logiczne

- **AND:** `and`
- **OR:** `or`
- **NOT:** `not`

Listing 10: Operatory logiczne w Pythonie

```
a = True
b = False

print(a and b)  # False
print(a or b)   # True
print(not a)    # False
```

Struktury Kontrolne Struktury kontrolne pozwalają na sterowanie przepływem programu.

Instrukcje Warunkowe Instrukcje warunkowe pozwalają na wykonywanie różnych bloków kodu w zależności od spełnienia określonych warunków.

Listing 11: Instrukcje warunkowe w Pythonie

```
wiek = 20

if wiek >= 18:  # Sprawdza, czy wiek jest większy lub równy 18
    print("Jesteś pełnoletni.")
elif wiek >= 16:  # Sprawdza, czy wiek jest większy lub równy 16
    print("Możesz ubiegać się o prawo jazdy.")
else:  # Wykonuje się, jeśli żaden z powyższych warunków nie jest spełniony
    print("Jesteś niepełnoletni.")
```

Pętle Pętle pozwalają na powtarzanie bloków kodu wielokrotnie.

Pętla `for` Pętla `for` jest używana do iteracji po sekwencjach (takich jak listy, krotki, słowniki, zestawy lub łańcuchy znaków).

Listing 12: Pętla for w Pythonie

```
# Przykład pętli for
owoce = ["jabłko", "banan", "czereśnia"]

for owoc in owoce:
    print(owoc)
```

Wyjście:

```
jabłko
banan
czereśnia
```

Pętla `while` Pętla `while` powtarza blok kodu tak długo, jak długo warunek jest prawdziwy.

Listing 13: Pętla while w Pythonie

```
# Przykład pętli while
licznik = 0

while licznik < 5:
    print("Licznik:", licznik)
    licznik += 1
```

Wyjście:

```
Licznik: 0
Licznik: 1
Licznik: 2
Licznik: 3
Licznik: 4
```

Przerwanie Pętli

- `break`: Przerywa pętlę.
- `continue`: Przerywa bieżącą iterację i przechodzi do następnej.

Listing 14: Przykład użycia break i continue

```python
# Przykład użycia break
for liczba in range(10):
    if liczba == 5:
        break
    print(liczba)

# Przykład użycia continue
for liczba in range(10):
    if liczba % 2 == 0: # Jeżeli liczba jest parzysta(reszta z dzielenia przez
    2 == 0) to przejdź dalej
        continue
    print(liczba)
```

Wyjście:

```
0
1
2
3
4

1
3
5
7
9
```

Funkcje Funkcje to bloki kodu, które można wielokrotnie wywoływać w programie. Pomagają one w organizacji kodu i jego ponownym wykorzystaniu.

Definiowanie Funkcji

Listing 15: Definiowanie funkcji w Pythonie

```
def przywitaj(imie):
    print(f"Witaj, {imie}!")
```

Wywoływanie Funkcji

Listing 16: Wywoływanie funkcji w Pythonie

```
przywitaj("Anna")  # Witaj, Anna!
przywitaj("Piotr")  # Witaj, Piotr!
```

Funkcje z Wartością Zwrotną

Listing 17: Funkcje z wartością zwrotną

```
def dodaj(a, b):
    return a + b

wynik = dodaj(5, 3)
print(wynik)  # 8
```

Funkcje z Argumentami Domyślnymi

Listing 18: Funkcje z argumentami domyślnymi

```
def powitanie(imie, powitanie="Cześć"):
    print(f"{powitanie}, {imie}!")

powitanie("Anna")                # Cześć, Anna!
powitanie("Piotr", "Witaj")      # Witaj, Piotr!
```

Struktury Danych Python oferuje różnorodne struktury danych, które umożliwiają przechowywanie i manipulację danymi w efektywny sposób.

Listy Listy są uporządkowanymi, zmiennymi sekwencjami elementów. Listy w Pythonie są indeksowane od 0, co oznacza, że pierwszy element listy ma indeks 0, drugi element ma indeks 1, i tak dalej.

Listing 19: Praca z listami w Pythonie

```
# Tworzenie listy
owoce = ["jabłko", "banan", "czereśnia"]

# Dostęp do elementów listy (indeksowanie od 0)
print(owoce[0])  # jabłko

# Dodawanie elementu
```

```
owoce.append("gruszka")
print(owoce)  # ["jabłko", "banan", "czereśnia", "gruszka"]

# Usuwanie elementu
owoce.remove("banan")
print(owoce)  # ["jabłko", "czereśnia", "gruszka"]
```

Krotki Krotki są uporządkowanymi, niezmiennymi sekwencjami elementów.

Listing 20: Praca z krotkami w Pythonie

```
# Tworzenie krotki
kolory = ("czerwony", "zielony", "niebieski")

# Dostęp do elementów krotki
print(kolory[1])  # zielony

# Próba modyfikacji krotki (spowoduje błąd)
# kolory[0] = "żółty"  # TypeError: 'tuple' object does not support item
    assignment
```

Słowniki Słowniki przechowują pary klucz-wartość, są nieuporządkowane i zmienne.

Listing 21: Praca ze słownikami w Pythonie

```
# Tworzenie słownika
student = {
    "imie": "Anna",
    "wiek": 22,
    "kurs": "Informatyka"
}

# Dostęp do wartości
print(student["imie"])  # Anna

# Dodawanie nowej pary klucz-wartość
student["rok"] = 3
print(student)  # {'imie': 'Anna', 'wiek': 22, 'kurs': 'Informatyka', 'rok': 3}

# Usuwanie pary klucz-wartość
del student["wiek"]
print(student)  # {'imie': 'Anna', 'kurs': 'Informatyka', 'rok': 3}
```

Zbiory Zbiory są nieuporządkowanymi, unikalnymi elementami.

Listing 22: Praca ze zbiorami w Pythonie

```
# Tworzenie zbioru
liczby = {1, 2, 3, 4, 5}

# Dodawanie elementu
liczby.add(6)
print(liczby)  # {1, 2, 3, 4, 5, 6}

# Usuwanie elementu
liczby.remove(3)
print(liczby)  # {1, 2, 4, 5, 6}

# Zbiory nie przechowują duplikatów
liczby.add(2)
print(liczby)  # {1, 2, 4, 5, 6}
```

Operacje Wejścia/Wyjścia Operacje wejścia/wyjścia pozwalają na interakcję z użytkownikiem oraz odczytywanie i zapisywanie danych.

Odczyt Danych od Użytkownika

Listing 23: Odczyt danych od użytkownika w Pythonie

```
# Odczyt danych od użytkownika
imie = input("Podaj swoje imię: ")
print(f"Witaj, {imie}!")
```

Po uruchomieniu tego programu, program będzie czekał, aż użytkownik poda swoje imię w wierszu poleceń.

Zapisywanie Danych do Pliku

Listing 24: Zapisywanie danych do pliku w Pythonie

```
# Zapisywanie danych do pliku
tekst = "To jest przykładowy tekst."

with open("plik.txt", "w") as plik:
    plik.write(tekst)
```

Po uruchomieniu tego programu, program będzie zapisywał dane do pliku o nazwie "plik.txt". "w"oznacza tryb zapisu (write). Otwiera plik do zapisu: Jeśli plik nie istnieje,

zostanie utworzony, jeśli plik istnieje, jego zawartość zostanie nadpisana.

Odczytywanie Danych z Pliku

Listing 25: Odczytywanie danych z pliku w Pythonie

```
# Odczytywanie danych z pliku
with open("plik.txt", "r") as plik:
    zawartosc = plik.read()
    print(zawartosc)
```

Praca z Bibliotekami Python posiada ogromną liczbę bibliotek, które rozszerzają jego funkcjonalność. Aby użyć biblioteki, należy ją najpierw zainstalować za pomocą `pip`, a następnie zaimportować w swoim programie.

Importowanie Biblioteki

Listing 26: Importowanie biblioteki NumPy w Pythonie

```
import numpy as np

# Tworzenie tablicy NumPy
tablica = np.array([1, 2, 3, 4, 5])
print(tablica)
```

Przykład z Biblioteką `math` Biblioteka `math` zawiera funkcje matematyczne.

Listing 27: Przykład użycia biblioteki math w Pythonie

```
import math

# Obliczanie pierwiastka kwadratowego
print(math.sqrt(16))  # 4.0

# Obliczanie sinusa kąta w radianach
print(math.sin(math.pi / 2))  # 1.0
```

Proste Projekty i Przykłady

Kalkulator Stwórz prosty kalkulator, który wykonuje podstawowe operacje arytmetyczne.

Listing 28: Prosty kalkulator w Pythonie

```
def dodaj(a, b):
```

```
    return a + b

def odejmij(a, b):
    return a - b

def mnoz(a, b):
    return a * b

def dziel(a, b):
    if b != 0:
        return a / b
    else:
        return "Nie można dzielić przez zero!"

print("Wybierz operację:")
print("1. Dodawanie")
print("2. Odejmowanie")
print("3. Mnożenie")
print("4. Dzielenie")

wybor = input("Wprowadź numer operacji (1/2/3/4): ")

a = float(input("Podaj pierwszą liczbę: "))
b = float(input("Podaj drugą liczbę: "))

if wybor == "1":
    print(f"Wynik: {dodaj(a, b)}")
elif wybor == "2":
    print(f"Wynik: {odejmij(a, b)}")
elif wybor == "3":
    print(f"Wynik: {mnoz(a, b)}")
elif wybor == "4":
    print(f"Wynik: {dziel(a, b)}")
else:
    print("Nieprawidłowy wybór!")
```

Liczby Fibonacciego Generuj ciąg liczb Fibonacciego do określonej liczby.

Listing 29: Generowanie liczb Fibonacciego w Pythonie

```
def fibonacci(n):
    a, b = 0, 1
    wynik = []
    while len(wynik) < n:
```

```
        wynik.append(a)
        a, b = b, a + b
    return wynik

liczba = int(input("Ile liczb Fibonacciego chcesz wygenerować? "))
print(fibonacci(liczba))
```

Co Dalej? Aby kontynuować naukę, rozważ następujące kroki:

1. **Praktyka:** Ćwicz, rozwiązując różnorodne zadania programistyczne. Platformy takie jak LeetCode, HackerRank, czy Codewars oferują wiele zadań do praktyki.
2. **Projekty:** Zacznij tworzyć własne projekty, które Cię interesują. Może to być strona internetowa, aplikacja do analizy danych, gra czy automatyzacja codziennych zadań.
3. **Społeczność:** Dołącz do społeczności programistów. Udzielaj się na forach, takich jak Stack Overflow, uczestnicz w meetupach, grupach na Reddit.
4. **Dokumentacja:** Naucz się korzystać z oficjalnej dokumentacji Pythona oraz bibliotek, które używasz.

Dodatkowe Zasoby

- **Oficjalna Dokumentacja Pythona:** python.org/doc/
- **Poradniki do Pythona:** w3schools.com/python/
- **Książki:**
  - *"Python Crash Course"* autorstwa Erica Matthesa
  - *"Automate the Boring Stuff with Python"* autorstwa Al Sweigarta
  - *"Learning Python"* autorstwa Marka Lutz

6.3.1 Kilka słów: Czym Jest Kernel?

Kernel to serce środowiska programistycznego, takiego jak Jupyter Notebook. Można go porównać do mózgu, który wykonuje Twoje polecenia. W kontekście Pythona, kernel interpretuje i wykonuje kod napisany w języku Python. **Jak Działa Kernel w Pythonie?**

- **Pisanie Kodów:** Kiedy piszesz kod w Jupyter Notebook, wpisujesz go w komórkach (ang. **cells**).
- **Wysyłanie Kodów do Kernela:** Po uruchomieniu komórki (na przykład naciskając `Shift + Enter`), kod jest wysyłany do kernela Pythona.

- **Wykonywanie Kodów:** Kernel interpretuje (czyta) Twój kod linia po linii i wykonuje go. To oznacza, że przeprowadza wszystkie operacje, które zdefiniowałeś w kodzie, takie jak obliczenia, manipulacje danymi czy wyświetlanie wyników.
- **Zwrot Wyników:** Po wykonaniu kodu, kernel zwraca wynik z powrotem do Jupyter Notebook, gdzie jest on wyświetlany bezpośrednio pod komórką z kodem.

Kernel Pythona ma dostęp do **bibliotek**, które są zbiorem gotowych funkcji, narzędzi i modułów, które ułatwiają wykonywanie różnych zadań. Dzięki tym bibliotekom, Kernel może wykonywać skomplikowane operacje bez konieczności pisania wszystkiego od podstaw. **Zmienne** są przechowywane tymczasowo w instancji Kernela. Oznacza to, że wszystkie zdefiniowane zmienne i ich wartości są utrzymywane w pamięci kernela podczas sesji. Jednak po zrestartowaniu kernela wszystkie zmienne zostają utracone, co wymaga ponownego zdefiniowania ich, jeśli chcesz się do nich odnieść. Jest to ważne do zrozumienia, ponieważ restart kernela może być konieczny w przypadku błędów lub zmian w kodzie, ale wymaga od użytkownika ponownego uruchomienia wcześniejszych komórek kodu, aby przywrócić stan środowiska pracy.

6.4 Spis narzędzi używanych w przykładach

6.4.1 Matplotlib

Matplotlib to jedna z najpopularniejszych bibliotek do tworzenia wykresów i wizualizacji danych w Pythonie. Jest niezwykle wszechstronna i pozwala na tworzenie różnorodnych wykresów, od prostych liniowych, po zaawansowane 3D. Ten poradnik pomoże Ci zacząć korzystać z Matplotlib w środowisku **Jupyter Notebook**.

Jeśli jeszcze nie masz zainstalowanej biblioteki Matplotlib, możesz to zrobić za pomocą `pip`. W Jupyter Notebooku możesz uruchomić komórkę z poniższym kodem:

Listing 30: Instalacja Matplotlib za pomocą pip

```
!pip install matplotlib
```

Uwaga: Dodanie wykrzyknika `!` pozwala na wykonanie komendy systemowej bezpośrednio w Jupyter Notebooku.

Podstawowe Importy Aby zacząć korzystać z Matplotlib, należy zaimportować odpowiednie moduły. Najczęściej używanym modułem jest `pyplot`, który dostarcza funkcje podobne do tych z MATLAB-a.

Listing 31: Importowanie biblioteki Matplotlib

```
import matplotlib.pyplot as plt
```

Dla lepszej integracji z Jupyter Notebookiem, warto dodać magiczną komendę `%matplotlib inline`, która umożliwia wyświetlanie wykresów bezpośrednio w notatniku.

Listing 32: Magiczna komenda dla Jupyter Notebook

```
%matplotlib inline
```

Tworzenie Prostych Wykresów

Wykres Liniowy Wykresy liniowe są podstawowym typem wykresów używanym do przedstawiania trendów w danych.

Listing 33: Tworzenie wykresu liniowego

```
# Dane
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

# Tworzenie wykresu liniowego
plt.plot(x, y)
```

```
# Wyświetlanie wykresu
plt.show()
```

Wykres Słupkowy Wykresy słupkowe są używane do porównywania różnych kategorii.

Listing 34: Tworzenie wykresu słupkowego

```
# Dane
kategorie = ["A", "B", "C", "D"]
wartosci = [10, 24, 36, 18]

# Tworzenie wykresu słupkowego
plt.bar(kategorie, wartosci, color="skyblue")

# Wyświetlanie wykresu
plt.show()
```

Wykres Punktowy Wykresy punktowe (scatter plots) są używane do przedstawiania zależności między dwiema zmiennymi.

Listing 35: Tworzenie wykresu punktowego

```
# Dane
x = [5, 7, 8, 7, 2, 17, 2, 9, 4, 11, 12, 9, 6]
y = [99, 86, 87, 88, 100, 86, 103, 87, 94, 78, 77, 85, 86]

# Tworzenie wykresu punktowego
plt.scatter(x, y, color="red")

# Wyświetlanie wykresu
plt.show()
```

Dostosowywanie Wykresów

Tytuł i Etykiety Dodanie tytułu oraz etykiet osi poprawia czytelność wykresu.

Listing 36: Dostosowanie tytułu i etykiet

```
# Dane
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

# Tworzenie wykresu liniowego z tytułem i etykietami
```

```
plt.plot(x, y, marker="o")
plt.title("Przykładowy Wykres Liniowy")
plt.xlabel("Oś X")
plt.ylabel("Oś Y")

# Wyświetlanie wykresu
plt.show()
```

Legenda Legenda pomaga w identyfikacji różnych serii danych na wykresie.

Listing 37: Dostosowanie legendy

```
# Dane
x = [1, 2, 3, 4, 5]
y1 = [2, 3, 5, 7, 11]
y2 = [1, 4, 6, 8, 10]

# Tworzenie wykresu z legendą
plt.plot(x, y1, label="Seria 1", marker="o")
plt.plot(x, y2, label="Seria 2", marker="x")
plt.title("Wykres z Legendą")
plt.xlabel("Oś X")
plt.ylabel("Oś Y")
plt.legend()

# Wyświetlanie wykresu
plt.show()
```

Stylizacja Matplotlib oferuje różne style wykresów, które można łatwo zastosować.

Listing 38: Stylizacja wykresu

```
# Ustawienie stylu
plt.style.use("ggplot")

# Dane
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

# Tworzenie wykresu liniowego ze stylizacją
plt.plot(x, y, marker="o")
plt.title("Wykres w Stylu ggplot")
plt.xlabel("Oś X")
plt.ylabel("Oś Y")
```

```
# Wyświetlanie wykresu
plt.show()
```

Zapis Wykresów do Pliku Możesz zapisać wykresy jako pliki graficzne (np. PNG, JPG, PDF) używając funkcji `savefig`.

Listing 39: Zapis wykresu do pliku

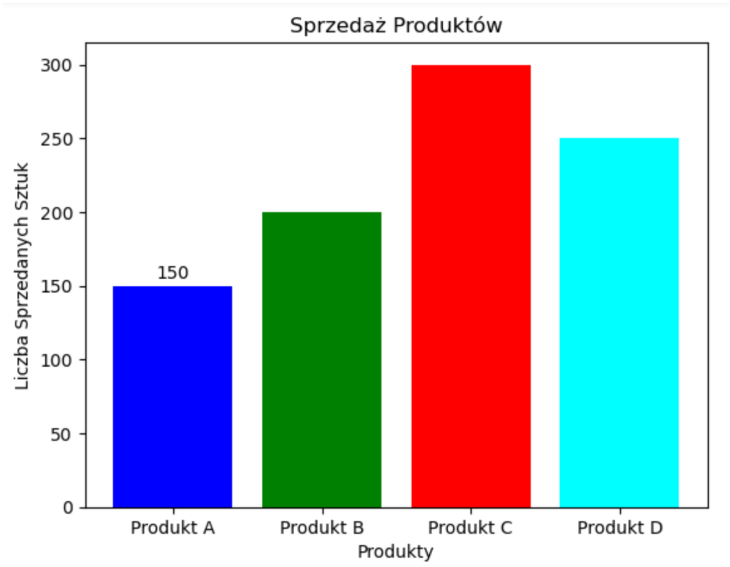
```
# Dane
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

# Tworzenie wykresu liniowego
plt.plot(x, y, marker="o")
plt.title("Wykres do Zapisu")
plt.xlabel("Oś X")
plt.ylabel("Oś Y")

# Zapisywanie wykresu do pliku
plt.savefig("wykres_liniowy.png")

# Wyświetlanie wykresu
plt.show()
```

Przykładowy Projekt Stworzymy wykres słupkowy przedstawiający sprzedaż różnych produktów.

Listing 40: Przykładowy wykres słupkowy

```
# Dane
produkty = ["Produkt A", "Produkt B", "Produkt C", "Produkt D"]
sprzedaz = [150, 200, 300, 250]

# Tworzenie wykresu słupkowego
plt.bar(produkty, sprzedaz, color=["blue", "green", "red", "cyan"])
plt.title("Sprzedaż Produktów")
plt.xlabel("Produkty")
plt.ylabel("Liczba Sprzedanych Sztuk")

# Dodanie etykiet wartości na słupkach
for i, v in enumerate(sprzedaz):
    plt.text(i, v + 5, str(v), ha="center")
```

```
# Wyświetlanie wykresu
plt.show()
```



Opis: Ilustracja sprzedaży różnych produktów przedstawiona na wykresie słupkowym.

Podsumowanie

Dodatkowe Zasoby Aby pogłębić swoją wiedzę na temat Matplotlib, rozważ zapoznanie się z poniższymi zasobami:

- **Oficjalna Dokumentacja Matplotlib:** matplotlib.org/stable/contents.html
- **Poradniki i Przykłady:** matplotlib.org/stable/tutorials/index.html
- **Książki:**
  - *"Python Data Science Handbook"* autorstwa Jake'a VanderPlasa
  - *"Matplotlib for Python Developers"* autorstwa Sandro Tosi
- **Społeczność:**
  - Stack Overflow - Matplotlib
  - Reddit - r/Python

6.4.2 Pandas

Pandas to jedna z najpopularniejszych bibliotek w Pythonie używana do analizy i manipulacji danymi. Umożliwia łatwe zarządzanie strukturami danych, takimi jak tabele (DataFrame) czy serie czasowe (Series). Ten krótki poradnik pomoże Ci zacząć korzystać z Pandas, nawet jeśli dopiero zaczynasz swoją przygodę z programowaniem.

Instalacja Pandas Zakładamy, że masz już zainstalowanego Jupyter Notebooka. Jeśli jeszcze nie masz `pandas`, możesz go zainstalować bezpośrednio z poziomu notebooka używając komendy `pip`:

```
!pip install pandas
```

Uwaga: Dodanie wykrzyknika `!` pozwala na wykonanie komendy systemowej bezpośrednio w Jupyter Notebooku.

Podstawowe Importy Na początku każdego notebooka zazwyczaj importujemy niezbędne biblioteki. Najczęściej używanym aliasem dla `pandas` jest `pd`:

```
import pandas as pd
```

Dodatkowo, często importuje się bibliotekę **NumPy** do operacji numerycznych:

```
import numpy as np
```

Podstawowe Struktury Danych `Pandas` oferuje dwie główne struktury danych:

Series **Series** to jednowymiarowa struktura danych, która może przechowywać dane różnych typów (np. liczby, ciągi znaków).

```
# Tworzenie Series z listy
data = [10, 20, 30, 40, 50]
series = pd.Series(data)
print(series)
```

Output:

```
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

Możemy również przypisać własne etykiety (indeksy):

```
series = pd.Series(data, index=["a", "b", "c", "d", "e"])
print(series)
```

Output:

```
a    10
b    20
c    30
d    40
e    50
dtype: int64
```

DataFrame **DataFrame** to dwuwymiarowa struktura danych, przypominająca tabelę, z wierszami i kolumnami.

Tworzenie DataFrame

Z Listy Możemy tworzyć `DataFrame` z listy list lub listy słowników.

```
# Tworzenie DataFrame z listy list
data = [
    ["Jan", 25, "Poznań"],
    ["Ewa", 30, "Łódź"],
    ["Tomasz", 22, "Wrocław"]
]
df = pd.DataFrame(data, columns=["Imię", "Wiek", "Miasto"])
print(df)
```

Output:

```
      Imię  Wiek   Miasto
0      Jan    25   Poznań
1      Ewa    30     Łódź
2  Tomasz    22  Wrocław
```

Z Słownika Tworzenie `DataFrame` ze słownika jest często bardziej przejrzyste.

```
# Tworzenie DataFrame ze słownika
data = {
    "Imię": ["Anna", "Piotr", "Maria"],
    "Wiek": [28, 34, 29],
    "Miasto": ["Warszawa", "Kraków", "Gdańsk"]
}
df = pd.DataFrame(data)
```

```
print(df)
```

Output:

```
   Imię  Wiek    Miasto
0  Anna    28  Warszawa
1 Piotr    34     Kraków
2 Maria    29     Gdańsk
```

Odczytywanie i Zapisywanie Danych Praca z rzeczywistymi danymi często polega na ich odczytywaniu z plików i zapisywaniu wyników do plików.

Odczyt z CSV `Pandas` umożliwia łatwe wczytywanie danych z plików CSV.

```
# Odczyt danych z pliku CSV
df = pd.read_csv("sciezka/do/twojego_pliku.csv")
print(df.head())  # Wyświetlenie pierwszych 5 wierszy
```

Uwaga: Upewnij się, że ścieżka do pliku CSV jest poprawna. Możesz również użyć ścieżki względnej, jeśli plik znajduje się w tym samym katalogu co notebook.

Zapis do CSV Po przeprowadzeniu analizy często chcemy zapisać wyniki do pliku.

```
# Zapis DataFrame do pliku CSV
df.to_csv("wynik.csv", index=False)
```

Parametr `index=False` powoduje, że indeksy wierszy nie zostaną zapisane do pliku CSV.

Podstawowe Operacje `Pandas` oferuje szeroki wachlarz funkcji do manipulacji i analizy danych.

Wyświetlanie Danych

- **Pierwsze kilka wierszy:**

```
print(df.head())  # Domyślnie 5 pierwszych wierszy
```

- **Ostatnie kilka wierszy:**

```
print(df.tail())  # Domyślnie 5 ostatnich wierszy
```

- **Informacje o DataFrame:**

```
print(df.info())
```

- **Statystyki opisowe:**

```
print(df.describe())
```

Selektowanie Kolumn i Wierszy

- **Wybieranie jednej kolumny:**

```
imiona = df["Imię"]
print(imiona)
```

- **Wybieranie wielu kolumn:**

```
dane_osobowe = df[["Imię", "Wiek"]]
print(dane_osobowe)
```

- **Wybieranie wierszy za pomocą indeksu:**

```
pierwszy_wiersz = df.iloc[0]
print(pierwszy_wiersz)
```

- **Wybieranie wierszy na podstawie warunku:**

```
wiek_powyzej_30 = df[df["Wiek"] > 30]
print(wiek_powyzej_30)
```

Filtrowanie Danych Filtrowanie pozwala na wybieranie danych spełniających określone kryteria.

```
# Filtracja osób z miasta "Warszawa"
warszawa = df[df["Miasto"] == "Warszawa"]
print(warszawa)
```

Podstawowe Statystyki `Pandas` umożliwia szybkie obliczanie różnych statystyk.

```
# Średnia wieku
srednia_wiek = df["Wiek"].mean()
print(f"Średni wiek: {srednia_wiek}")

# Maksymalny wiek
maks_wiek = df["Wiek"].max()
print(f"Maksymalny wiek: {maks_wiek}")

# Minimalny wiek
min_wiek = df["Wiek"].min()
print(f"Minimalny wiek: {min_wiek}")
```

Obsługa Brakujących Danych Często w danych mogą występować brakujące wartości (`NaN`). `Pandas` oferuje narzędzia do ich obsługi.

- **Sprawdzenie brakujących danych:**

```
print(df.isnull().sum())
```

- **Usuwanie wierszy z brakującymi danymi:**

```
df_cleaned = df.dropna()
```

- **Uzupełnianie brakujących danych:**

```
# Uzupełnianie brakujących wartości w kolumnie "Wiek" średnią wartością
srednia_wiek = df["Wiek"].mean()
df["Wiek"] = df["Wiek"].fillna(srednia_wiek)
```

Podsumowanie

Dodatkowe Zasoby Aby pogłębić swoją wiedzę na temat `pandas` i analizy danych, warto skorzystać z poniższych źródeł:

- **Oficjalna Dokumentacja Pandas:** https://pandas.pydata.org/
- **Książki:**
  - *Python for Data Analysis* autorstwa Wes McKinney

- **Społeczność:**
  - Stack Overflow - Tag pandas
  - Forum pandas GitHub
- **Poradniki i Artykuły:**
  - Real Python - Pandas Tutorials
  - Towards Data Science - Pandas

6.4.3 Requests

Instalacja Requests Zakładamy, że masz już uruchomionego Jupyter Notebooka. Jeśli jeszcze nie masz zainstalowanej biblioteki `Requests`, możesz to zrobić bezpośrednio z poziomu notebooka używając komendy `pip`:

```
!pip install requests
```

Uwaga: Wykrzyknik `!` pozwala na wykonanie komendy systemowej bezpośrednio w Jupyter Notebooku.

Podstawowe Importy Na początku każdego notebooka zazwyczaj importujemy niezbędne biblioteki. Importujemy `Requests` bez aliasu, ale możesz użyć aliasu, jeśli wolisz.

```
import requests
```

Dodatkowo, warto importować bibliotekę **json**, jeśli planujesz pracować z danymi w formacie JSON.

```
import json
```

Podstawowe Operacje

Wykonywanie Żądania GET Żądanie GET jest najprostszym typem żądania HTTP, które pozwala na pobranie danych z określonego URL.

```
# Wykonanie żądania GET
response = requests.get("https://api.github.com")

# Wyświetlenie statusu odpowiedzi
print(response.status_code)
```

Output:

```
200
```

Wykonywanie Żądania POST Żądanie POST jest używane do wysyłania danych do serwera, na przykład w formularzach.

```
# Dane do wysłania
payload = {"key1": "value1", "key2": "value2"}

# Wykonanie żądania POST
response = requests.post("https://httpbin.org/post", data=payload)

# Wyświetlenie odpowiedzi
```

```
print(response.text)
```

Obsługa Odpowiedzi Po wykonaniu żądania, możesz uzyskać różne informacje z odpowiedzi.

```
# Sprawdzenie statusu odpowiedzi
print(response.status_code)  # 200 oznacza sukces

# Pobranie treści odpowiedzi jako tekst
print(response.text)

# Pobranie treści odpowiedzi jako JSON (jeśli dostępny)
try:
    data = response.json()
    print(data)
except ValueError:
    print("Odpowiedź nie jest w formacie JSON.")
```

Przykładowe Zastosowania

Pobieranie Strony WWW Możesz pobierać zawartość stron internetowych i analizować ją lub przetwarzać dalej.

```
# Pobranie strony głównej Google
response = requests.get("https://www.google.com")

# Sprawdzenie statusu odpowiedzi
if response.status_code == 200:
    # Wyświetlenie pierwszych 500 znaków treści strony
    print(response.text[:500])
else:
    print(f"Nie udało się pobrać strony. Status: {response.status_code}")
```

Interakcja z API Biblioteka `Requests` jest często używana do komunikacji z API, które udostępniają dane w formacie JSON.

```
# Przykładowe API do pobrania informacji o użytkownikach GitHub
url = "https://api.github.com/users/HydraScienceFundation"

# Wykonanie żądania GET
response = requests.get(url)
```

```python
# Sprawdzenie statusu odpowiedzi
if response.status_code == 200:
    user_data = response.json()
    print(f"Login: {user_data['login']}")
    print(f"ID: {user_data['id']}")
    print(f"URL: {user_data['html_url']}")
else:
    print(f"Nie udało się pobrać danych. Status: {response.status_code}")
```

Obsługa Błędów Podczas pracy z `Requests` ważne jest, aby obsługiwać potencjalne błędy, takie jak problemy z połączeniem czy niepoprawne odpowiedzi.

```python
try:
    response =
    requests.get("https://api.github.com/users/HydraScienceFundation",
    timeout=5)
    response.raise_for_status()  # Sprawdza, czy status jest OK (200)
    user_data = response.json()
    print(user_data)
except requests.exceptions.HTTPError as http_err:
    print(f"Błąd HTTP: {http_err}")
except requests.exceptions.ConnectionError as conn_err:
    print(f"Błąd połączenia: {conn_err}")
except requests.exceptions.Timeout as timeout_err:
    print(f"Przekroczono limit czasu: {timeout_err}")
except requests.exceptions.RequestException as req_err:
    print(f"Inny błąd: {req_err}")
except ValueError:
    print("Odpowiedź nie jest w formacie JSON.")
```

Podsumowanie Biblioteka `Requests` to potężne narzędzie do komunikacji z internetowymi serwisami i API, które pozwala na łatwe pobieranie i wysyłanie danych. Im więcej będziesz z niej korzystać, tym bardziej zaawansowane operacje będziesz w stanie wykonywać.

Dodatkowe Zasoby Aby pogłębić swoją wiedzę na temat biblioteki `Requests` i pracy z API, warto skorzystać z poniższych źródeł:

- **Oficjalna Dokumentacja Requests:** https://requests.readthedocs.io/
- **Poradniki i Artykuły:**
  - Real Python - Requests Tutorial

  - Towards Data Science - Using Requests in Python

- **Społeczność:**
  - Stack Overflow - Tag requests
  - Forum GitHub Requests

6.5 Przykłady użycia bibliotek z API Warszawy

W poniższym rozdziale przedstawimy na przykładach, jak korzystać z opisanych wcześniej bibliotek, używając API miasta Warszawy. Kod omawiany poniżej jest dostępny w wygodnej do kopiowania formie w pliku `API_poradnik_przyklady_bibliotek_z_api_Warszawy.ipynb`, zapisanym w formacie Jupyter Notebook.

6.5.1 Wczytywanie klucza

Klucze API nie powinny być przechowywane bezpośrednio w kodzie, ponieważ w przypadku jego udostępnienia osobom trzecim, mogłyby one wykorzystać klucz do autoryzowania się, podając się za nas. Dobrą praktyką jest przechowywanie kluczy w osobnym pliku, który nie jest udostępniany publicznie. W tym poradniku będziemy przechowywać klucz w pliku `api_key.txt`, który musi znajdować się w tym samym katalogu co notebook. Klucz zostanie załadowany do zmiennej `api_key`, do której będziemy się odnosić w dalszej części poradnika, korzystając z poniższego kodu:

```
# Otwieramy plik api_key.txt i wczytujemy klucz
with open('api_key.txt', 'r') as file:
    api_key = file.read().strip()

# Wyświetlenie klucza (opcjonalne, tylko do testów)
print(api_key)
```

6.5.2 Pobieranie danych

Wysyłanie żądań o uzyskanie danych realizujemy za pomocą funkcji `get()` oferowanej przez bibliotekę `requests`, podając w argumencie napis będący adresem URL, do którego chcemy uzyskać dostęp. Aby zobaczyć przesłaną zawartość, możemy użyć funkcji `json()`, która konwertuje obiekt w formacie JSON na słownik.

Napiszemy funkcję, która pozwala nam odpytać API Warszawy o znanym URL przy użyciu naszego klucza API.

Listing 41: Funkcja wykonująca zapytanie GET

```
import requests
import json

def request_get(url, api_key):
    """
    Wykonuje zapytanie GET do dynamicznego URL-a z kluczem API.

    Args:
        url (str): Podstawowy URL.
```

```
        api_key (str): Klucz API.

    Returns:
        Response: Odpowiedź z serwera (obiekt `requests.Response`).
    """
    # Formatujemy dynamiczny URL
    dynamic_url = f"{url}&apikey={api_key}"

    # Wykonujemy zapytanie GET
    response = requests.get(dynamic_url)

    # Zwracamy odpowiedź
    return response
```

Po zdefiniowaniu funkcji możemy użyć jej do odpytania API Warszawy o przystanki wraz z ich współrzędnymi.

```
api_warszawa_url = "https://api.um.warszawa.pl/api/action/"  # URL API Warszawy
dbfunction = "dbstore_get/?id=ab75c33d-3a26-4342-b36a-6e5fef0a3ac3"  #
    Współrzędne przystanków
url = api_warszawa_url + dbfunction  # URL endpointa przystanków z ich
    współrzędnymi

response = request_get(url, api_key)
response
```

Output:

```
<Response [200]>
```

Po wywołaniu tego programu powinniśmy zobaczyć w pierwszej linii kod zwrotny HTTP. Kod 200 oznacza poprawne wykonanie żądania GET. Następnie możemy podejrzeć otrzymane dane w formacie JSON, aby upewnić się, że są poprawne.

```
data_raw = response.json()  # Eksportujemy otrzymane dane do formatu JSON
data_raw
```

Output:


```
{'result': [{'values': [{'value': '1001', 'key': 'zespol'},
    {'value': '01', 'key': 'slupek'},
    {'value': 'Kijowska', 'key': 'nazwa_zespolu'},
    {'value': '2201', 'key': 'id_ulicy'},
    {'value': '52.248455', 'key': 'szer_geo'},
    {'value': '21.044827', 'key': 'dlug_geo'},
    {'value': 'al.Zieleniecka', 'key': 'kierunek'},
```

```
    {'value': '2024-08-15 00:00:00.0', 'key': 'obowiazuje_od'}]},
  {'values': [{'value': '1001', 'key': 'zespol'},
    {'value': '02', 'key': 'slupek'},
    {'value': 'Kijowska', 'key': 'nazwa_zespolu'},
    ...
```

W outputcie jest odpowiedź na nasze żądanie, czyli zawartość bazy danych. W takiej formie nie jest zbyt czytelna, jednak nadal dostarcza nam wartościowych informacji. Możemy zobaczyć, w jakiej formie są nam dostarczane dane, co jest bardzo użyteczne w późniejszej pracy z nimi. W tym przypadku dane są w postaci słownika z jednym kluczem *result*, a wartością tego klucza są informacje o bazie, również w postaci słownika. Na przykład klucz *fields* zawiera nazwy i typy wszystkich kolumn, a *records* wszystkie wpisy.

Ponieważ API Warszawy w tym endpointzie dostarcza dane w postaci jednoelementowego słownika słowników, wygodne będzie "obranie"jednej warstwy. Możemy to zrobić za pomocą komendy:

```
data = data_raw["result"]
```

Tak przygotowane dane użyjemy w następnych częściach poradnika.

6.5.3 Eksportowanie danych do formatu CSV

Popularnym formatem używanym przez wielu ludzi, w tym również tych niezwiązanych z programowaniem, jest CSV (Comma-Separated Values). Naturalnie może pojawić się potrzeba wyeksportowania danych pobranych w formacie JSON do tego formatu, aby przesłać je innym lub pracować na nich w innych programach, takich jak Microsoft Excel. W tym celu napiszemy funkcję, która umożliwi nam konwersję danych JSON do pliku CSV:

Listing 42: Funkcja eksportująca dane JSON do pliku CSV

```
import csv

def export_json_to_csv(data, file_name):
    """
    Eksportuje dane JSON (listę słowników) do pliku CSV.

    Args:
        data (list): Dane JSON, które są listą słowników.
        file_name (str): Nazwa pliku CSV, do którego zapisać dane.

    Returns:
        None
    """
    try:
```

```
        # Upewniamy się, że dane są listą słowników
        if isinstance(data, list) and all(isinstance(item, dict) for item in
    data):
            # Otwieramy plik w trybie zapisu
            with open(file_name, mode='w', newline='', encoding='utf-8') as
    csv_file:
                # Inicjalizujemy writer CSV
                writer = csv.DictWriter(csv_file, fieldnames=data[0].keys())

                # Zapisujemy nagłówki i dane
                writer.writeheader()
                writer.writerows(data)

            print(f"Dane zostały zapisane do pliku: {file_name}")
        else:
            print("Dane JSON nie są listą słowników. Eksport nie jest możliwy.")
    except Exception as e:
        print(f"Wystąpił błąd podczas eksportu: {e}")
```

Po zdefiniowaniu tej funkcji możemy ją wykonać za pomocą poniższego kodu:

```
# Eksport danych do CSV
export_json_to_csv(data, "output.csv")
```

Po wykonaniu tej komórki w katalogu z notebookiem powinien pojawić się plik `output.csv` z danymi.

6.5.4 Importowanie danych JSON do pandas DataFrame

Posiadamy dane o przystankach warszawskich w formacie JSON:


```
[{'values': [{'value': '1001', 'key': 'zespol'},
   {'value': '01', 'key': 'slupek'},
   {'value': 'Kijowska', 'key': 'nazwa_zespolu'},
   {'value': '2201', 'key': 'id_ulicy'},
   {'value': '52.248455', 'key': 'szer_geo'},
   {'value': '21.044827', 'key': 'dlug_geo'},
   {'value': 'al.Zieleniecka', 'key': 'kierunek'},
   {'value': '2024-08-15 00:00:00.0', 'key': 'obowiazuje_od'}]},
 ...]
```


Aby zaimportować te dane do pandas DataFrame, wykonamy następujące kroki:

1. **Importowanie bibliotek**: Użyjemy bibliotek `pandas` i `json`.

2. **Przetworzenie danych**: Przekształcimy listę słowników, gdzie każdy słownik zawiera listę par klucz-wartość, na bardziej płaską strukturę odpowiednią dla DataFrame.

3. **Utworzenie DataFrame**: Stworzymy DataFrame z przetworzonych danych.

Poniżej przedstawiamy kod realizujący te kroki:

Listing 43: Importowanie danych JSON do pandas DataFrame

```
import pandas as pd
import json

# Przetwarzanie danych JSON na listę słowników
processed_data = []
for entry in data:
    entry_dict = {}
    for item in entry['values']:
        entry_dict[item['key']] = item['value']
    processed_data.append(entry_dict)

# Tworzenie DataFrame
df = pd.DataFrame(processed_data)

# Wyświetlenie DataFrame
print(df.head(10))
```

Wyjaśnienia:

- `processed_data`: Tworzymy nową listę, gdzie każdy element jest słownikiem z kluczami odpowiadającymi atrybutom przystanków.
- `for entry in data_json`: Iterujemy przez każdy wpis w danych JSON.
- `for item in entry["values"]`: Iterujemy przez listę wartości w każdym wpisie.
- `entry_dict[item["key"]]`: Przypisujemy wartość do odpowiedniego klucza w nowym słowniku.
- `df = pd.DataFrame(processed_data)`: Tworzymy pandas DataFrame z przetworzonych danych.
- `print(df.head(10)`: Wyświetlamy zawartość DataFrame (10 pierwszych rekordów).

Output:

```
  zespol slupek nazwa_zespolu id_ulicy   szer_geo   dlug_geo  \
0   1001     01      Kijowska     2201  52.248455  21.044827
1   1001     02      Kijowska     2201  52.249078  21.044443
2   1001     03      Kijowska     2201  52.248928  21.044169
3   1001     04      Kijowska     2201  52.249969  21.041588
4   1001     05      Kijowska     1203  52.250319  21.043861
5   1001     06      Kijowska     1203  52.250078  21.043848
6   1001     07      Kijowska     1203  52.250228  21.043690
7   1001     08      Kijowska     1203  52.249944  21.044087
8   1001     88      Kijowska     1203  52.249944  21.044087
9   1002     01     Ząbkowska     2201  52.251325  21.038457

                 kierunek          obowiazuje_od
0          al.Zieleniecka  2024-08-15 00:00:00.0
1               Ząbkowska  2024-08-15 00:00:00.0
2          al.Zieleniecka  2024-08-15 00:00:00.0
3               Ząbkowska  2024-08-15 00:00:00.0
4          al.Zieleniecka  2024-08-15 00:00:00.0
5  Dw.Wschodni (Kijowska)  2024-08-15 00:00:00.0
6               Ząbkowska  2024-08-15 00:00:00.0
7  Dw.Wschodni (Kijowska)  2024-08-15 00:00:00.0
8  Dw.Wschodni (Kijowska)  2024-08-15 00:00:00.0
9                Kijowska  2024-08-15 00:00:00.0
```

Dane zaprezentowane w tej formie są znacznie czytelniejsze – każdej nazwie przystanku (np. Kijowska) odpowiada parametr "zespół". Jednakże, ponieważ istnieje wiele przystanków o tej samej nazwie, są one rozróżniane za pomocą pola "słupek".

Posiadając dane o przystankach warszawskich w wygodnej formie pandas DataFrame, otwierają się przed nami liczne możliwości analizy i przetwarzania informacji. Przede wszystkim, możemy łatwo filtrować dane według różnych kryteriów, takich jak nazwa zespołu przystankowego, kierunek, czy współrzędne geograficzne, co umożliwia szybką identyfikację interesujących nas przystanków. Dodatkowo, dzięki funkcjom agregacji i grupowania, możemy analizować liczbę przystanków na poszczególnych ulicach lub w różnych dzielnicach miasta, co jest przydatne przy planowaniu tras komunikacyjnych czy optymalizacji infrastruktury transportowej.

6.5.5 Analiza i Wizualizacja danych

W celu przećwiczenia wizualizacji danych spróbujemy wykonać histogram ilości słupków dla przystanków. Do tego celu posłuży nam biblioteka `matplotlib`. Na początek potrzebujemy danych, ile dany zespół ma słupków. Możemy to zrobić za pomocą następującego kodu:

Listing 44: Tworzenie DataFrame z liczbą słupków dla każdego zespołu

```
# Tworzenie nowego DataFrame z liczbą słupków dla każdego zespołu
new_df = df.groupby(['zespol', 'nazwa_zespolu']).agg(ilosc_slupkow=('slupek',
    'count')).reset_index()

# Wyświetlenie nowego DataFrame
print("\nNowy DataFrame z liczbą słupków dla każdego zespołu:")
print(new_df)
```

Output:

```
Nowy DataFrame z liczbą słupków dla każdego zespołu:
     zespol        nazwa_zespolu  ilosc_slupkow
0      1001             Kijowska              9
1      1002            Ząbkowska              5
2      1003          Dw.Wileński             17
3      1004        Ratuszowa-ZOO              8
4      1005           pl.Hallera             13
...     ...                   ...            ...
2975   R-07        ZEA WORONICZA              2
2976   R-10     ZEA OSTROBRAMSKA              4
2977   R-11       ZEA KLESZCZOWA              2
2978   R-13          ZEA STALOWA              2
2979   R-19  WYDZIAŁ WŁOŚCIAŃSKA              2

[2980 rows x 3 columns]
```

Możemy przeprowadzić wstępną analizę danych o liczbie słupków:

Listing 45: Obliczanie statystyk liczby słupków

```
print(f"Minimalna ilość słupków: {new_df['ilosc_slupkow'].min()}")
print(f"Maksymalna ilość słupków: {new_df['ilosc_slupkow'].max()}")
print(f"Średnia ilość słupków: {new_df['ilosc_slupkow'].mean()}")
print(f"Mediana ilości słupków: {new_df['ilosc_slupkow'].median()}")
```

Output:

```
Minimalna ilość słupków: 1
Maksymalna ilość słupków: 30
Średnia ilość słupków: 2.785234899328859
Mediana ilości słupków: 2.0
```

Jak widać, rozkład ilości słupków jest rozkładem lewo-skośnym (mediana mniejsza od średniej). Zobaczmy, który przystanek ma najwięcej słupków:

Listing 46: Znajdowanie przystanku z największą ilością słupków

```
# Znalezienie indeksu maksymalnej wartości w kolumnie 'ilosc_slupkow'
max_index = new_df['ilosc_slupkow'].idxmax()

# Pobranie rekordu z największą ilością słupków
max_record = new_df.loc[max_index]

print("Rekord z największą ilością słupków:")
print(max_record)
```

Output:

```
Rekord z największą ilością słupków:
zespol                      6059
nazwa_zespolu    Metro Młociny
ilosc_slupkow               30
Name: 2615, dtype: object
```

Posiadając dane o liczbie słupków przypisanych do poszczególnych zespołów przystanków w wygodnej formie pandas DataFrame, możemy przeprowadzić różnorodne analizy wizualne. Jednym z przydatnych narzędzi do tego celu jest histogram, który pozwala na zobrazowanie rozkładu liczby słupków wśród wszystkich zespołów przystanków. Dzięki bibliotece `matplotlib` możemy łatwo stworzyć taki wykres, co umożliwia szybkie zrozumienie, jak liczba słupków jest rozłożona w całym zbiorze danych.

Poniżej przedstawiamy kod Python, który generuje histogram pokazujący, ile przystanków ma określoną liczbę słupków (`ilosc_slupkow`) na podstawie danych w DataFrame `new_df`.

Listing 47: Tworzenie histogramu liczby słupków na przystanek

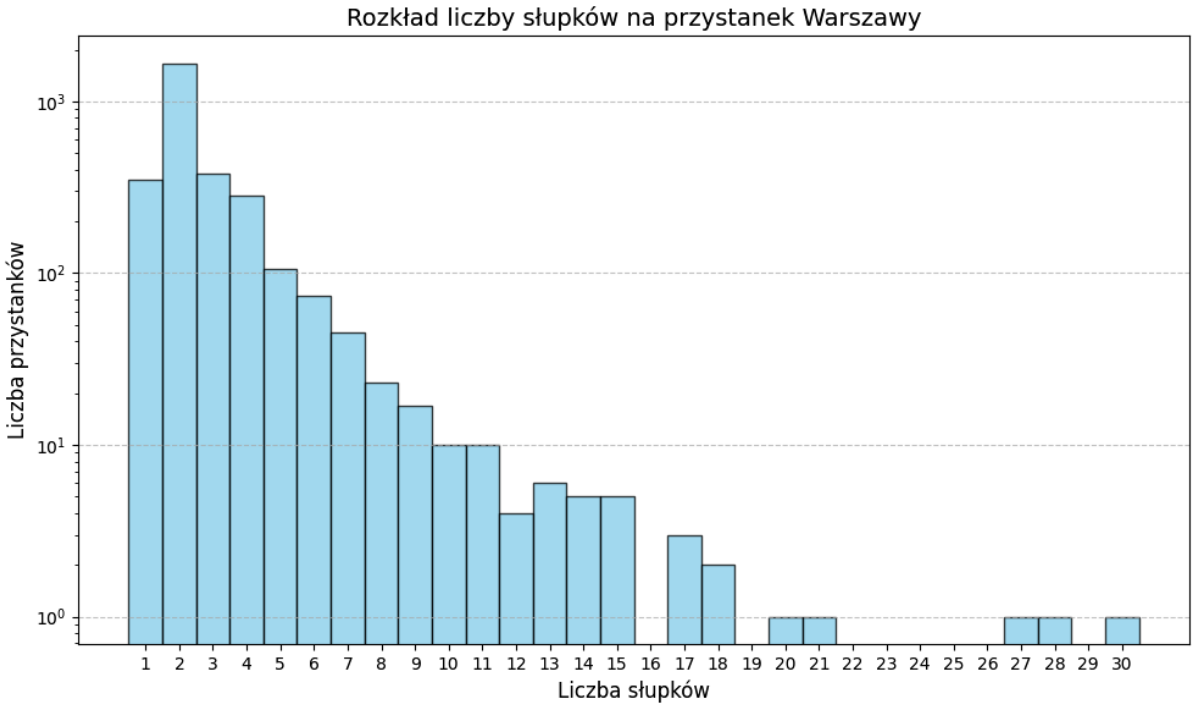
```
import pandas as pd
import matplotlib.pyplot as plt

# Ustawienia wykresu
plt.figure(figsize=(10, 6))  # Rozmiar wykresu (szerokość, wysokość) w calach

# Tworzenie histogramu
plt.hist(
    new_df['ilosc_slupkow'],
    bins=range(1, new_df['ilosc_slupkow'].max() + 1),  # Ustalanie binów od 1
   do max + 1
    edgecolor='black',  # Czarna krawędź dla słupków
    alpha=0.7,          # Przezroczystość słupków
    color='skyblue',    # Kolor słupków
    align = 'left'      # Centrowanie podpisów
```

```
)

# Dodanie tytułu i etykiet osi
plt.title('Rozkład liczby słupków na przystanek Warszawy', fontsize=14)
plt.xlabel('Liczba słupków', fontsize=12)
plt.ylabel('Liczba przystanków', fontsize=12)

# Ustawienie etykiet osi X na wartości całkowite
plt.xticks(range(1, new_df['ilosc_slupkow'].max() + 1))

# Dodanie siatki dla lepszej czytelności
plt.grid(axis='y', linestyle='--', alpha=0.7)

# Zastosowanie skali logarytmicznej ze względu na duży rozrzut danych
plt.yscale("log")

# Dostosowanie układu wykresu
plt.tight_layout()

#Zapisywanie wykresu do pliku
plt.savefig('histogram_slupkow.png', dpi=300)

# Wyświetlenie wykresu
plt.show()
```

Wyjaśnienie Kodu:

- `plt.figure(figsize=(10, 6))`: Ustawiamy rozmiar wykresu na 10 cali szerokości i 6 cali wysokości.
- `plt.hist(...)` : Tworzymy histogram:
  - `new_df['ilosc_slupkow']`: Dane do histogramu.
  - `bins=range(1, new_df['ilosc_slupkow'].max() + 2)`: Ustalamy zakres binów od 1 do maksymalnej liczby słupków + 1, aby uwzględnić wszystkie wartości.
  - `edgecolor='black'`: Dodajemy czarne krawędzie do słupków dla lepszej widoczności.
  - `alpha=0.7`: Ustawiamy przezroczystość słupków na 70%.
  - `color='skyblue'`: Ustawiamy kolor słupków na jasnoniebieski.
  - `align = 'left'`: Centrowanie podpisów
- `plt.title(...), plt.xlabel(...), plt.ylabel(...)`: Dodajemy tytuł wykresu oraz etykiety osi X i Y z odpowiednimi rozmiarami czcionek.

- `plt.xticks(...)`: Ustawiamy etykiety na osi X na wartości od 1 do maksymalnej liczby słupków.
- `plt.grid(...)`: Dodajemy siatkę poziomą z przerywaną linią i przezroczystością 70% dla lepszej czytelności.
- `plt.yscale("log")`: Zastosowanie skali logarytmicznej ze względu na duży zakres danych.
- `plt.tight_layout()`: Dostosowujemy układ wykresu, aby uniknąć nakładania się elementów.
- `plt.savefig()`: Zapisuje wykres do pliku
- `plt.show()`: Wyświetlamy wykres.



Rysunek 24: Rozkład liczby słupków na przystanek Warszawy

Mam nadzieję, że ten wstęp przybliżył Ci podstawy programowania oraz pokazał, jak efektywnie wykorzystywać API miasta Warszawy przy użyciu podstawowych bibliotek. Dzięki zdobytej wiedzy będziesz mógł realizować różnorodne projekty, analizować dane oraz tworzyć wizualizacje, które ułatwią zrozumienie informacji pochodzących z API. W kolejnych częściach poradnika przedstawimy bardziej zaawansowane i praktyczne przykłady wykorzystania API Warszawy w Pythonie. Pokażę, jak integrować dane z różnych źródeł, przeprowadzać zaawansowane analizy, które mogą wspierać podejmowanie decyzji oraz optymalizację procesów. Dzięki temu będziesz mógł wykorzystać zdobytą wiedzę w

bardziej skomplikowanych zastosowaniach, rozwijając swoje umiejętności programistyczne i analityczne.

7 Praktyczne użycia API

7.1 Konsultacje społeczne - wykorzystanie

7.1.1 Opis Zadania

Konsultacje społeczne są istotnym elementem procesu decyzyjnego w nowoczesnych miastach, takich jak Warszawa. Pozwalają mieszkańcom aktywnie uczestniczyć w kształtowaniu przestrzeni publicznej, polityki lokalnej czy planów inwestycyjnych, a także wyrażać swoje opinie na tematy bezpośrednio wpływające na ich życie. Jednak złożoność i obszerność dokumentacji związanej z konsultacjami często stanowi wyzwanie zarówno dla mieszkańców, jak i organizacji pozarządowych, które starają się monitorować ich przebieg oraz wpływ.

7.1.2 Co to znaczy dla NGO

Organizacje pozarządowe mogą wspierać procesy konsultacyjne, analizując dostępne dane, przygotowując przystępne podsumowania i informując społeczności lokalne o najważniejszych wnioskach. Dzięki temu umożliwiają mieszkańcom łatwiejszy dostęp do kluczowych informacji i wspierają ich zaangażowanie w życie miasta.

W Warszawie ułatwić to może wykorzystanie danych z miejskich API, w ramach którego miasto udostępnia między innymi informacje o prowadzonych konsultacjach społecznych.

7.1.3 Cel i rezultaty zadania

Celem tego poradnika jest pokazanie jak za pomocą miejskiego API można pozyskać dane dotyczące przeprowadzonych konsultacji społecznych w Warszawie. Następnie wykorzystamy narzędzia AI, takie jak ChatGPT, by automatycznie streścić obszerny tekst do formy krótkiego i przystępnego podsumowania, jakie mogłoby być udostępniane mieszkańcom zainteresowanym sprawami swojej społeczności.

Zadanie to będzie składało się z dwóch części:

1. W części I pobierzemy z API Warszawy dane dotyczące przeprowadzonych konsultacji społecznych w mieście. Dane te mogą zawierać szczegóły dotyczące tematyki, lokalizacji, terminów i wyników konsultacji.

2. W części II skrócimy opis konsultacji do przystępnego podsumowania. Wykorzystamy dwa podejścia:

   (a) Manualne streszczenie za pomocą ChatGPT: Skopiujemy treść dokumentu i ręcznie poprosimy ChatGPT o wygenerowanie krótkiego podsumowania.

(b) Wykonamy automatyzację za pomocą OpenAI API: Zautomatyzujemy proces streszczania, korzystając z interfejsu API, co umożliwi szybkie przetwarzanie większej liczby dokumentów. Uwaga! Wykorzystanie OpenAI API wymaga uiszczenia opłaty przez użytkownika za każde zapytanie.

7.1.4 Analiza problemu

W celu realizacji zadania należy zbiorczo pobrać dane o przeprowadzonych konsultacjach i następnie streścić długi tekst za pomocą ChatGPT - manualnie, a także za pomocą OpenAI API.

Krok 1: Wczytanie API key Warszawy

```
def load_api_key(filepath='api_key.txt'):
    """Filepath zakłada obecność pliku api_key.txt z kluczem API w tym samym
    folderze co ten notebook"""
    try:
        with open(filepath, 'r') as file:  # otwarcie pliku .txt zawierającego
   klucz API
            api_key = file.read().strip()  # przypisanie zawartości pliku do
   lokalnej zmiennej 'api_key'
        return api_key                     # słowo kluczowe 'return' konczy
   metodę i zwraca wartość zmiennej
    except FileNotFoundError:              # jeśli wystapił błąd nieznalezienia
   pliku, nie zgłaszaj od razu błedu, tylko...
        print("API key file not found.")   # ... napisz o tym komunikat
        return None                        # i niech metoda zwróci wartość
   'None'
    except Exception as e:                 # podobnie jeśli wystąpią inne błędy
        print(f"An error occurred: {e}")
        return None

api_key = load_api_key()                   # do zmiennej 'api_key' przypisz to
   co zwróci metoda 'load_api_key' (to może być 'None')
```

Krok 2: Dostęp do danych konsultacji społecznych

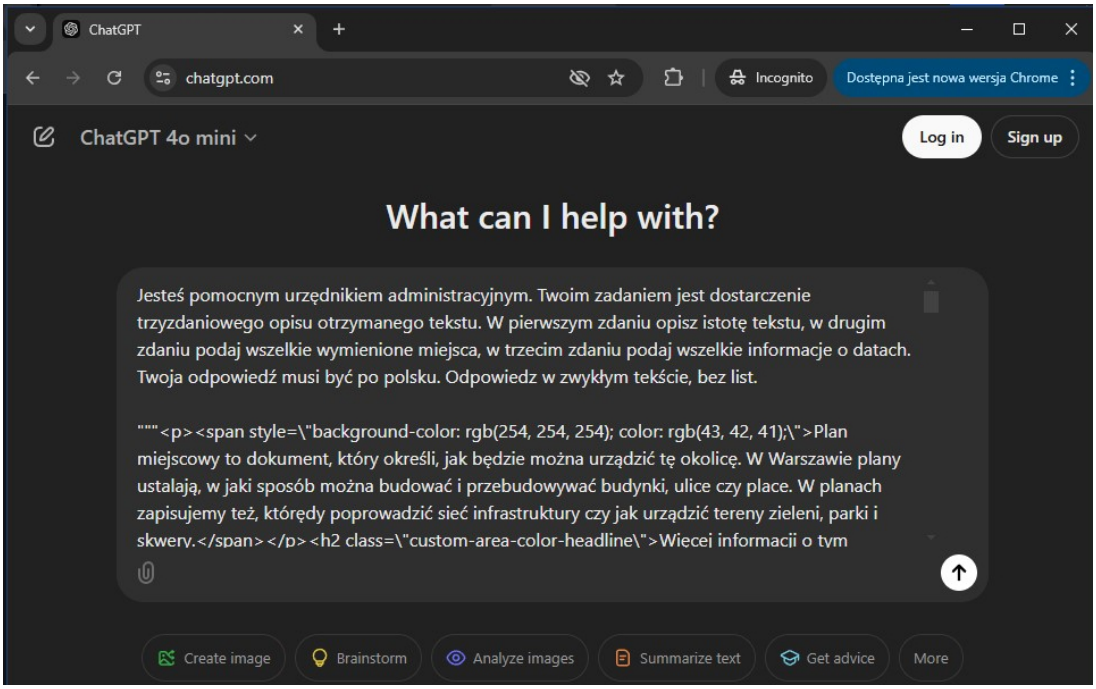
```
from datetime import datetime, timedelta
import requests
import json
import time
```

Listing 48: Pobranie informacji o konsultacjach

```
consult_url =
    f'https://api.um.warszawa.pl/api/action/public_consultation/?apikey={api_key}'

consult_response = requests.get(consult_url)
consult_inText = json.loads(consult_response.text)
consult_data = consult_inText['result']['data']

with open('consult_data.json', 'w', encoding='utf-8') as f:
    json.dump(consult_data, f, ensure_ascii=False, indent=4)

if consult_data:
    print(f'Pobrano {len(consult_data)} elementy dot. konsultacji społecznych')
```

API miasta zwraca tylko 3 konsultacje - to dobry pretekst by spróbować streścić dłuższe opisy konsultacji do 3 zdań. Jest to dobry pretekst bo łatwo wyobrazić sobie sytuację pracy z setkami tekstów wymagających usystematyzowanego streszczenia.

Krok 3: Manualna analiza z chatGPT Pod adresem https://chatgpt.com/ dostępny jest ChatGPT, gdzie bez tworzenia konta można wypróbować ten sam model, który poniżej używany jest do automatycznego streszczania długich opisów konsultacji.



Rysunek 25: Enter Caption

Warto pamiętać, że choć ChatGPT jest bardzo popularny, nie jest jedynym dostępnym produktem generatywnej sztucznej inteligencji. Warto zapoznać się z konkurencyjnymi modelami, które mogą być tańsze, lepsze w konkretnym zadaniu, lub możliwe do darmowego użycia na własnym komputerze - o to wszystko można też na początek zapytać chatGPT!

Krok 4: Przykładowa automatyczna analiza z API chatGPT

```
import openai
from openai import OpenAI
```

UWAGA! - API chat GPT to usługa płatna Interakcja z chatGPT poprzez API lub bibliotekę Pythona wymaga posiadania klucza dostępu (i utworzenia konta OpenAI). Interakcja z chatGPT poprzez API jest płatna, poniższe 3 streszczenia konsultacji kosztowały poniżej $0.01 (poniżej jednego centa) w grudniu 2024.

Listing 49: Stwórz zapytanie dla chatGPT

```
gpt_key = load_api_key(filepath='gpt-key')

client = OpenAI(api_key=gpt_key)
thread = client.beta.threads.create()
assistant_content = "Jesteś pomocnym urzędnikiem administracyjnym. Twoim
    zadaniem jest dostarczenie trzyzdaniowego opisu otrzymanego tekstu. W
    pierwszym zdaniu opisz istotę tekstu, w drugim zdaniu podaj wszelkie
    wymienione miejsca, w trzecim zdaniu podaj wszelkie informacje o datach.
    Twoja odpowiedź musi być po polsku. Odpowiedz w zwykłym tekście, bez list."
print(thread)
```

Listing 50: Konstruowanie odpowiedzi z API ChatGPT

```
consultation_dict_flag = True # Kontrola nad przypadkowym wywołaniem

consultation_dict = {}

if consultation_dict_flag:
    for consult_element in consult_data:
        description = consult_element['attributes']['description']
        title = consult_element['attributes']['title']
        consult_elem_id = consult_element['id']
        short_description = consult_element['attributes']['short-description']

        response = client.chat.completions.create(
            model="gpt-4o-mini",
            messages=[
                {"role": "system", "content": assistant_content},
                {"role": "user", "content": f'Provide summary for:
    "{description}" '}
            ]
        )
        response_message = response.choices[0].message.content.strip()
```

```
        consultation_dict[title] = {
            "gpt-description": response_message,
            "short_description": short_description
        }
```

Listing 51: Wydrukowanie odpowiedzi API

```
dictionary_text = json.dumps(consultation_dict, sort_keys=True, indent=4)
decoded_text = dictionary_text.encode('utf-8').decode('unicode_escape')
print(decoded_text)
```

Listing 52: Opdpowiedz z ChatGPT API

```
{
    "Jaki␣plan␣miejscowy␣dla␣obszaru␣w␣rejonie␣ulic␣W wozowa ,␣Kabacki␣
       Dukt,␣Wilczy␣D    ,␣Ryba t w?": {
        "gpt-description": "Tekst␣dotyczy␣planu␣miejscowego,␣kt ry␣ma␣
           na␣celu␣okre lenie␣zasad␣zagospodarowania␣obszaru␣Ursynowa␣w
           ␣Warszawie,␣w␣tym␣budowy␣i␣ewentualnych␣zmian␣w␣
           infrastrukturze␣oraz␣terenach␣zieleni.␣...",
        "short_description": "<p>Trwaj ␣prace␣nad␣miejscowym␣planem␣
           zagospodarowania␣...␣</p>"
    },
    "Jakie␣nowe␣Studium␣dla␣Warszawy?": {
        "gpt-description": "Tekst␣przedstawia␣Studium␣uwarunkowa ␣i␣
           kierunk w␣zagospodarowania␣przestrzennego␣m.st.␣Warszawy,␣
           kt re␣definiuje␣wizj ␣rozwoju␣miasta␣na␣nadchodz ce␣dekady
           ␣oraz␣koncentruje␣si ␣na␣zr wnowa onym␣rozwoju␣i␣...</p>""
␣␣␣␣},
␣␣␣␣"Jakie zmiany w uchwale o obszarze rewitalizacji?":␣{
␣␣␣␣␣␣␣␣"gpt-description":␣"Tekst dotyczy celu konsultacji zwi zanych
   ze zmianami uchwa y dotycz cej obszaru rewitalizacji w Warszawie
   oraz szczeg    w zwi zanych z prawem pierwokupu. Wspomniane s
   obszary takie jak Groch w, ...</p>"
```

7.2 Parkingi miejskie - Scraping Danych

7.2.1 Opis Zadania

W dużych miastach dostępność miejsc parkingowych jest jednym z najważniejszych wyzwań dla mieszkańców oraz osób i instytucji zarządzających infrastrukturą miejską. Monitorowanie zajętości poszczególnych miejsc parkingowych w dłuższym okresie może pomóc zrozumieć, jak wykorzystywane są parkingi w różnych częściach miasta, zidentyfikować najbardziej obciążone lokalizacje, godziny szczytu oraz zmiany sezonowe w zajętości parkingów. Dzięki temu władze miasta mogą podejmować lepsze decyzje dotyczące zarządzania istniejącymi miejscami parkingowymi, wprowadzania zmian w systemach opłat czy planowania nowych parkingów tam, gdzie są najbardziej potrzebne.

Organizacje pozarządowe działające na rzecz mieszkańców mogą również wykorzystać te dane. Na przykład będą one przydatne, by precyzyjnie wskazywać problemy z infrastrukturą, które wpływają na codzienne funkcjonowanie lokalnych społeczności i proponować konkretne rozwiązania, poprawiające jakość życia mieszkańców i wspierające efektywne zarządzanie przestrzenią miejską.

7.2.2 Cel i rezultaty zadania

W ramach tego przykładu wykorzystamy dane udostępniane przez Zarząd Dróg Miejskich w Warszawie o aktualnej zajętości wybranych miejsc parkingowych w Warszawie. Pokażemy, jak przy użyciu scrapingu danych API można pozyskiwać i gromadzić dane o statusie zajętości miejsc parkingowych, a następnie jak przygotować je do dalszego przetwarzania i analizy.

7.2.3 Analiza problemu

Krok 1: Wczytanie API key Warszawy

```
def load_api_key(filepath='api_key.txt'):
    """Filepath zakłada obecność pliku api_key.txt z kluczem API w tym samym
    folderze co ten notebook"""
    try:
        with open(filepath, 'r') as file:  # otwarcie pliku .txt zawierającego
   klucz API
            api_key = file.read().strip()  # przypisanie zawartości pliku do
   lokalnej zmiennej 'api_key'
        return api_key                     # słowo kluczowe 'return' konczy
   metodę i zwraca wartość zmiennej
    except FileNotFoundError:              # jeśli wystąpił błąd nieznalezienia
   pliku, nie zgłaszaj od razu błedu, tylko...
        print("API key file not found.")   # ... napisz o tym komunikat
```

```
        return None                         # i niech metoda zwróci wartość
    'None'
    except Exception as e:                  # podobnie jeśli wystąpią inne błędy
        print(f"An error occurred: {e}")
        return None

api_key = load_api_key()                    # do zmiennej 'api_key' przypisz to
    co zwróci metoda 'load_api_key' (to może być 'None'))
```

Krok 2: Niezbędne definicje i stałe (tj. wartości niezmienne i do ponownego używania)

Listing 53: Wgranie bibliotek

```
from datetime import datetime, timedelta  # załaduj z modułu 'datetime'
    operacje na typie daty, oraz obliczanie róznic pomiędzy datami
import requests                           # załaduj moduł pozwalający wysyłać
    zapytania HTTP
import json                               # załaduj moduł obsługujący format
    danych JSON
import time
```

Listing 54: Defninicja zapytania API

```
parking_url =
    f'https://api.um.warszawa.pl/api/action/parking_get_list/?apikey={api_key}'
```

Krok 3: Pobranie i zapis danych

Po zdefiniowaniu bibliotek i przygotowaniu zapytania do API, uruchamiana jest pętla, która działa do ręcznego zatrzymania (Ctrl+C). W każdej iteracji wysyłane jest zapytanie do API, a dane o parkingach odczytywane są z ścieżki ['result']['Parks']. Następnie zapisujemy dane do pliku JSON, dodając do nazwy pliku datę i czas pobrania dla łatwego rozróżnienia wersji. Na koniec w konsoli wyświetlany jest komunikat o pobranych danych, np. „Pobrano informację o 20 parkingach o 2024-12-28-15-45-30".

Skrypt robi 60-sekundową przerwę przed kolejną iteracją.

```
while True:
    parking_response = requests.get(parking_url)
    parking_response_inText = json.loads(parking_response.text)
    parking_data = parking_response_inText['result']['Parks']

    # Odczytanie aktualnej daty oraz konfiguracja foormatu w jakim czas jest
    zapisywany
```

```
    current_datetime = datetime.now()
    formatted_datetime = current_datetime.strftime("%Y-%m-%d-%H-%M-%S")

    # Zapisz dane do pliku json. Np. parking-data-2024-12-28-22-13-30.json
    # uzycie zmiennej encoding='utf-8' powoduje prawidłowy zapis i odczyt
   polskich znaków
    with open(f'parking-data-{formatted_datetime}.json', 'w', encoding='utf-8')
   as f:
        json.dump(parking_data, f, ensure_ascii=False, indent=4)

    if parking_data:
        print(f'Pobrano informację o {len(parking_data)} parkingach o
   {formatted_datetime}')

    time.sleep(60) # Odczekaj do ponownego odczytu 60s. Szybszy odczyt danych
   nie jest potrzebny gdyż dane z parkingow pobierane są co minutę
```

Krok 4: Odczyt zapisanych danych i zapisanie do pliku programu Excel

Kod przeszukuje bieżący folder w poszukiwaniu plików JSON zawierających dane o parkingach. Pliki są otwierane w trybie odczytu, co zapobiega przypadkowemu nadpisaniu danych historycznych. Dzięki ustalonemu wzorcowi nazewnictwa plików ich przetwarzanie odbywa się automatycznie i w odpowiedniej kolejności. Załadowane dane są obrabiane i strukturyzowane za pomocą pakietu pandas, a następnie zapisywane w pliku Excel. Tak przygotowane dane można łatwo analizować, generować z nich raporty oraz otwierać w dowolnym programie obskubujący arkusze dancyh, co umożliwia ich wykorzystanie przez osoby bez specjalistycznej wiedzy technicznej.

```
free_places_data = []

for file_name in os.listdir('.'):
    if file_name.startswith('parking-data-') and file_name.endswith('.json'):
        try:
            with open(file_name, 'r', encoding='utf-8') as f:
                data = json.load(f)

                for entry in data:
                    free_places = entry.get('free_places_total', {})
                    name = entry.get('name', 'Unknown')
                    date_time = file_name.replace('parking-data-',
   '').replace('.json', '')

                    free_places_data.append({
```

```
                        'Nazwa Parkingu': name,
                        'Miejsca dla niepelnosprawnych':
    free_places.get('disabled', None),
                        'Miejsca publiczne': free_places.get('public', None),
                        'Miejsca na samochody elektryczne':
    free_places.get('electric', None),
                        'Data': date_time
                    })
        except (json.JSONDecodeError, KeyError) as e: # Kontrola obsługi
    błędów. W przypadkuch problemu otworzeniem dotaniemy stosowną informacje.
    Sprawdz czy plik nie zostałzmodyfikowany
            print(f"Błąd odczytu pliku {file_name}: {e}")

df = pd.DataFrame(free_places_data)

output_file = 'free-places-summary.xlsx'
df.to_excel(output_file, index=False, engine='openpyxl') # Export pliku eo
    arkusza escel

print(f"Zapisano {output_file}")
```

7.3 Parkingi miejskie - Wizualizacja Danych

7.3.1 Opis Zadania

Każdy, kto prowadził kiedyś samochód w większym mieście prawdopodobnie wie, że znalezienie miejsca parkingowego często bywa problemem. Miejsca parkingowe są ważnym elementem i częścią infrastruktury miasta, która bezpośrednio wpływa na życie mieszkańców. Ich odpowiednie rozmieszczenie pozwala oszczędzać czas, zmniejsza korki i pomaga lepiej zorganizować ruch na ulicach. Szczególnie w Strefie Płatnego Parkowania Niestrzeżonego (SPPN) kluczowe jest, aby miejsca były jednocześnie dobrze rozplanowane, dostępne dla kierowców i nie dominowały przestrzeni miejskiej.

W Warszawie do tego celu można korzystać z danych miejskiego API, które udostępnia Zarząd Dróg Miejskich. API pozwala między innymi na pobranie szczegółowych informacji o lokalizacji miejsc parkingowych w strefie SPPN. Te dane można wykorzystać do stworzenia czytelnych map i analiz.

7.3.2 Co to znaczy dla NGO

Organizacje pozarządowe mogą odegrać istotną rolę w analizowaniu i monitorowaniu sytuacji związanej z parkingami, w tym na przykład wpływać na decyzje o tworzeniu i likwidacji parkingów miejskich. Do takich zadań jak identyfikowanie obszarów o największym zapotrzebowaniu na miejsca parkingowe, wskazywania niedoborów w infrastrukturze lub stref z nadmierną liczbą parkingów, niezastąpione są dane miejskie. Dodatkowo, prezentowanie ich w przystępnej formie, na przykład poprzez wizualizacje mapowe jest kluczowe dla wspierania mieszkańców w łatwym dostępie do informacji i ułatwienia im zrozumienia sytuacji.

7.3.3 Cel i rezultaty zadania

W tym poradniku pokażemy, jak pobrać dane o miejscach parkingowych w Warszawie z miejskiego API i przedstawić je na mapie. Zobaczymy, jak takie dane można przetworzyć i wykorzystać, aby lepiej zrozumieć sytuację parkingową w mieście.

7.3.4 Analiza problemu

Krok 1: wczytanie API key Warszawy

```
def load_api_key(filepath='api_key.txt'):
    """Filepath zakłada obecność pliku api_key.txt z kluczem API w tym samym
    folderze co ten notebook"""
    try:
        with open(filepath, 'r') as file:  # otwarcie pliku .txt zawierającego
   klucz API
            api_key = file.read().strip()  # przypisanie zawartości pliku do
   lokalnej zmiennej 'api_key'
```

```
        return api_key                      # słowo kluczowe 'return' konczy
    metodę i zwraca wartość zmiennej
    except FileNotFoundError:               # jeśli wystapił błąd nieznalezienia
    pliku, nie zgłaszaj od razu błedy, tylko...
        print("API key file not found.")    # ... napisz o tym komunikat
        return None                         # i niech emtoda zwróci wartość
    'None'
    except Exception as e:                  # podobnie jeśli wystąpią inne błędy
        print(f"An error occurred: {e}")
        return None

api_key = load_api_key()                    # do zmiennej 'api_key' przypisz to
    co zwróci metoda 'load_api_key' (to może być 'None')
```

Krok 2: pobranie danych miejsc parkingowych

```
from datetime import datetime, timedelta  # załaduj z modułu 'datetime'
    operacje na typie daty, oraz obliczanie róznic pomiędzy datami
import requests                           # załaduj moduł pozwalający wysyłać
    zapytania HTTP
import json                               # załaduj moduł obsługujący format
    danych JSON
```

Kolejnym punktem jest wczytanie danych tak samo jak w sekcji 7.2 z tą różnicą że dane odczytujemy tylko raz

```
parking_url =
    f'https://api.um.warszawa.pl/api/action/parking_get_list/?apikey={api_key}'

parking_response = requests.get(parking_url)
parking_response_inText = json.loads(parking_response.text)
parking_data = parking_response_inText['result']['Parks']

current_datetime = datetime.now()
formatted_datetime = current_datetime.strftime("%Y-%m-%d")

with open(f'parking-data-{formatted_datetime}.json', 'w', encoding='utf-8') as
    f:
    json.dump(parking_data, f, ensure_ascii=False, indent=4)

if parking_data:
    print(f'Pobrano {len(parking_data)} elementy dot. miejsc parkingowych')
```

Krok 3: Przypisanie obiektu mapy folium do zmiennej 'm'

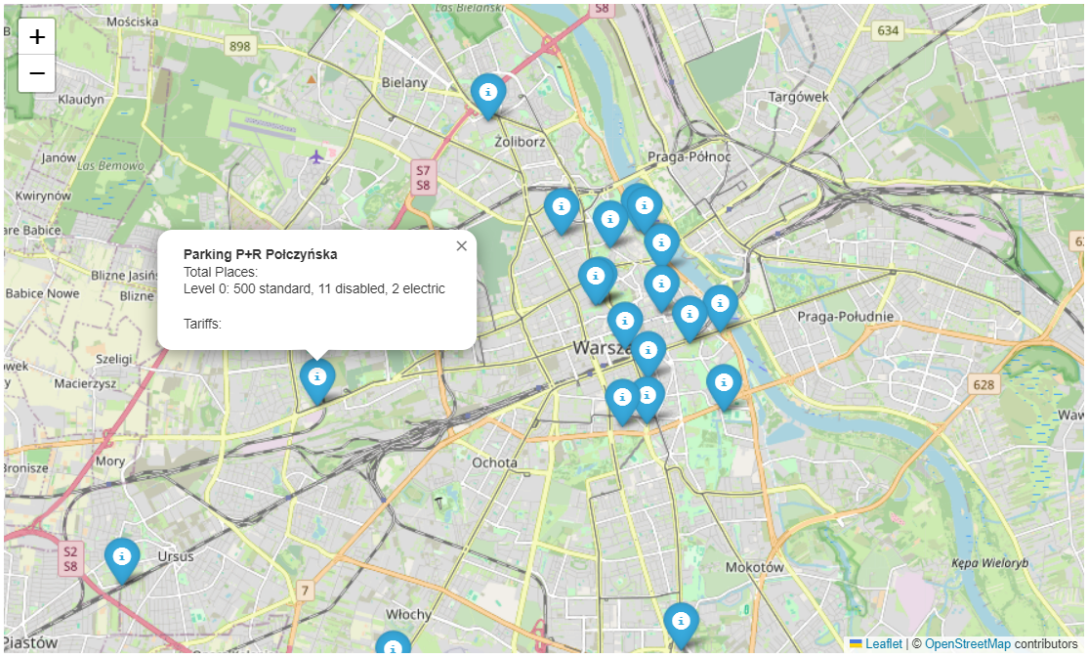
```
import folium  # folium pozwala na wyświetlanie interakrywnych map Ziemi
# obitk nowej mapy tworzymy wycentrowniem na Warszawe
m = folium.Map(location=[52.2319581, 21.0067249], zoom_start=12)
```

Krok 4: Dodanie oznaczeń do mapy Kolejnym etapem jest dodanie oznaczeń do mapy na podstawie załadowanych danych.

```
for entry in parking_data:
    latitude = float(entry['latitude'])
    longitude = float(entry['longitude'])

    total_places_info = "\n".join(
        [f"Level {place['level']}: {place['standard']} standard,
    {place['disabled']} disabled, {place['electric']} electric"
         for place in entry['total_places']]
    )

    tariffs_info = "\n".join(
        [
            f"{tariff['VehicleType']}: " + ", ".join(
                [f"{price['StayLength']} hr: {price['StayPrice']} PLN" for
    price in tariff['Prices']]
            )
            for tariff in entry['tariffs']
        ]
    )

    popup_content = f"<b>{entry['name']}</b><br>Total
    Places:<br>{total_places_info}<br><br>Tariffs:<br>{tariffs_info}"
    popup = folium.Popup(popup_content, max_width=300)

    folium.Marker(
        location=[latitude, longitude],
        popup=popup,
        icon=folium.Icon(color='blue', icon='info-sign')
    ).add_to(m)

m.save("parking_map.html")
```

Krok 5: Wyświetlenie mapy Warszawy z zaznaczonymi lokalizacjami parkingów Na sam koniec w celu wyświetlenia mapy w notebook wywołujemy zmienną z mapą

```
m
```



Rysunek 26: Mapa z lokalizacjami parkingow i podstawowymi informacjami

7.4 ZTM stats

7.4.1 Opis Zadania

Komunikacja miejska odgrywa kluczową rolę w codziennym funkcjonowaniu mieszkańców dużych miast, takich jak Warszawa. Jest podstawą mobilności społecznej, umożliwiając mieszkańcom łatwe i szybkie przemieszczanie się między domem, pracą, szkołą, miejscami rekreacji czy usługami publicznymi. Dobrze rozwinięta sieć komunikacji publicznej zmniejsza zależność od samochodów osobowych, co z kolei ogranicza korki, obniża poziom zanieczyszczenia powietrza oraz poprawia jakość życia. Zapewnia również równy dostęp do infrastruktury miejskiej, szczególnie dla osób, które nie mają własnych środków transportu, takich jak seniorzy, osoby młodsze czy mieszkańcy o niższych dochodach. W kontekście rosnącej liczby ludności w miastach, rozwój efektywnej, dostępnej i zrównoważonej komunikacji miejskiej jest jednym z kluczowych wyzwań dla współczesnych metropolii, które chcą być przyjazne zarówno dla swoich mieszkańców, jak i środowiska naturalnego.

Organizacje pozarządowe mogą odgrywać ważną rolę w ulepszaniu i rozwijaniu transportu publicznego, monitorując jego funkcjonowanie, analizując dostępne dane i sygnalizując zarówno potrzeby mieszkańców, jak i nieefektywności systemu. Mogą na przykład przeprowadzać badania dotyczące dostępności przystanków, częstotliwości kursów czy rozkładów jazdy, a także identyfikować opóźnienia, obszary miasta z ograniczonym dostępem do komunikacji, miejsca nierównomiernej dystrybucji połączeń czy zaniedbane trasy.

W Warszawie w tym celu przydatne może być wykorzystanie danych z miejskich API, które umożliwia między innymi uzyskanie informacji o infrastrukturze ZTM czy położeniu pojazdów w czasie rzeczywistym.

7.4.2 Cel i rezultaty zadania

Celem tego zadania jest zaprezentowanie sposobu pozyskiwania danych dotyczących komunikacji miejskiej z zasobów API Warszawy. Na potrzeby zadania pozyskamy dane o liczbie przystanków w Warszawie oraz liczbie linii autobusowych obsługujących przystanki, by sprawdzić, jaka część przystanków obsługuje mało linii, a ile jest kluczowymi węzłami komunikacyjnymi z wieloma połączeniami.

Pobrane dane o przystankach i liniach autobusowych zostaną przedstawione na histogramie ilustrującym rozkład ilości linii autobusowych na warszawskich przystankach. Przykład korzysta z API miasta Warszawy do pobrania danych, biblioteki *pandas* do zmiany formatu danych oraz bibliotek *seaborn* i *matplotlib* do przygotowania i wyświetlenia wykresu.

Krok 1: Import modułów i wczytanie API key

```
import json
import requests
```

```
def load_api_key(filepath='api_key.txt'):
    """Filepath zakłada obecność pliku api_key.txt z kluczem API w tym samym
    folderze co ten notebook"""
    try:
        with open(filepath, 'r') as file:  # otwarcie pliku .txt zawierającego
    klucz API
            api_key = file.read().strip()  # przypisanie zawartości pliku do
    lokalnej zmiennej 'api_key'
        return api_key                     # słowo kluczowe 'return' konczy
    metodę i zwraca wartość zmiennej
    except FileNotFoundError:              # jeśli wystapił błąd nieznalezienia
    pliku, nie zgłaszaj od razu błedu, tylko...
        print("API key file not found.")   # ... napisz o tym komunikat
        return None                        # i niech metoda zwróci wartość
    'None'
    except Exception as e:                 # podobnie jeśli wystąpią inne błędy
        print(f"An error occurred: {e}")
        return None

api_key = load_api_key()                   # do zmiennej 'api_key' przypisz to
    co zwróci metoda 'load_api_key' (to może być 'None')
```

Krok 2: Pobranie danych przystanków Poniższy kod pobiera dane o przystankach komunikacji miejskiej z API Warszawy, przetwarza je do bardziej przejrzystego formatu i zapisuje w pliku JSON.

```
stop_response = requests.get(
    url="https://api.um.warszawa.pl/api/action/dbstore_get",
    params={
        "id": "ab75c33d-3a26-4342-b36a-6e5fef0a3ac3",
        "apikey": api_key,
    })
```

```
stop_response_inText = json.loads(stop_response.text)
stop_data = stop_response_inText['result']

converted_stop_data = []
for element in stop_data:
    item = {}
    for prop in element['values']:
        item[prop['key']] = prop['value']
    converted_stop_data.append(item)
```

```
with open('converted_stop_data.json', 'w', encoding='utf-8') as f:
    json.dump(converted_stop_data, f, ensure_ascii=False, indent=4)

if converted_stop_data:
    print(f'Pobrano {len(converted_stop_data)} elementów danych o przystankach')
```

Krok 3: pobranie danych linii autobusowych Poniższy kod pobiera informacje o autobusach przypisanych do przystanków komunikacji miejskiej w Warszawie za pomocą API.

```
url_timetable = "https://api.um.warszawa.pl/api/action/dbtimetable_get"

resource_busstop = "b27f4c17-5c50-4a5b-89dd-236b282bc499"
resource_lines = "88cd555f-6f31-43ca-9de4-66c479ad5942"
resource_table = "e923fa0e-d96c-43f9-ae6e-60518c9f3238"
```

```
def get_lines(busstop_id, busstop_nr):
    return requests.get(
        url_timetable,
        params={
            "id": resource_lines,
            "apikey": api_key,
            "busstopId": busstop_id,
            "busstopNr": busstop_nr,
        },
    )

buses_list = []
for element in converted_stop_data:
    new_element = {
        "zespol": element["zespol"],
        "slupek": element["slupek"],
        "buses": get_lines(element["zespol"], element["slupek"])
    }
    buses_list.append(new_element)
```

```
bus_counter = 0
for stop in buses_list:
    for bus in stop['buses']:
        bus_counter += 1
```

```
print(f"Pobrano {bus_counter} informacji o autobusach.")
```

Krok 4: Zmiana formatu danych Poniższy kod tworzy nową listę *buses_list_cleared*, w której każdemu słupkowi (zawierającemu dane o autobusach) przypisywane są przetworzone informacje. W pętli najpierw odczytujemy surowe dane JSON a nastepnie wybrane wartości z każdego wpisu i zapisujemy je w tablicy,

Następnie powstała lista zapisywana jest do pliku *buses_list_cleared.json*, aby zachować dane w czytelnym formacie. Ostatni fragment kodu zlicza wystąpienia błędnych wpisów „Błędna" i wyświetla ich udział procentowy w konsoli. Dzięki temu szybko sprawdzimy, ile odpowiedzi z API zawiera nieprawidłowe dane.

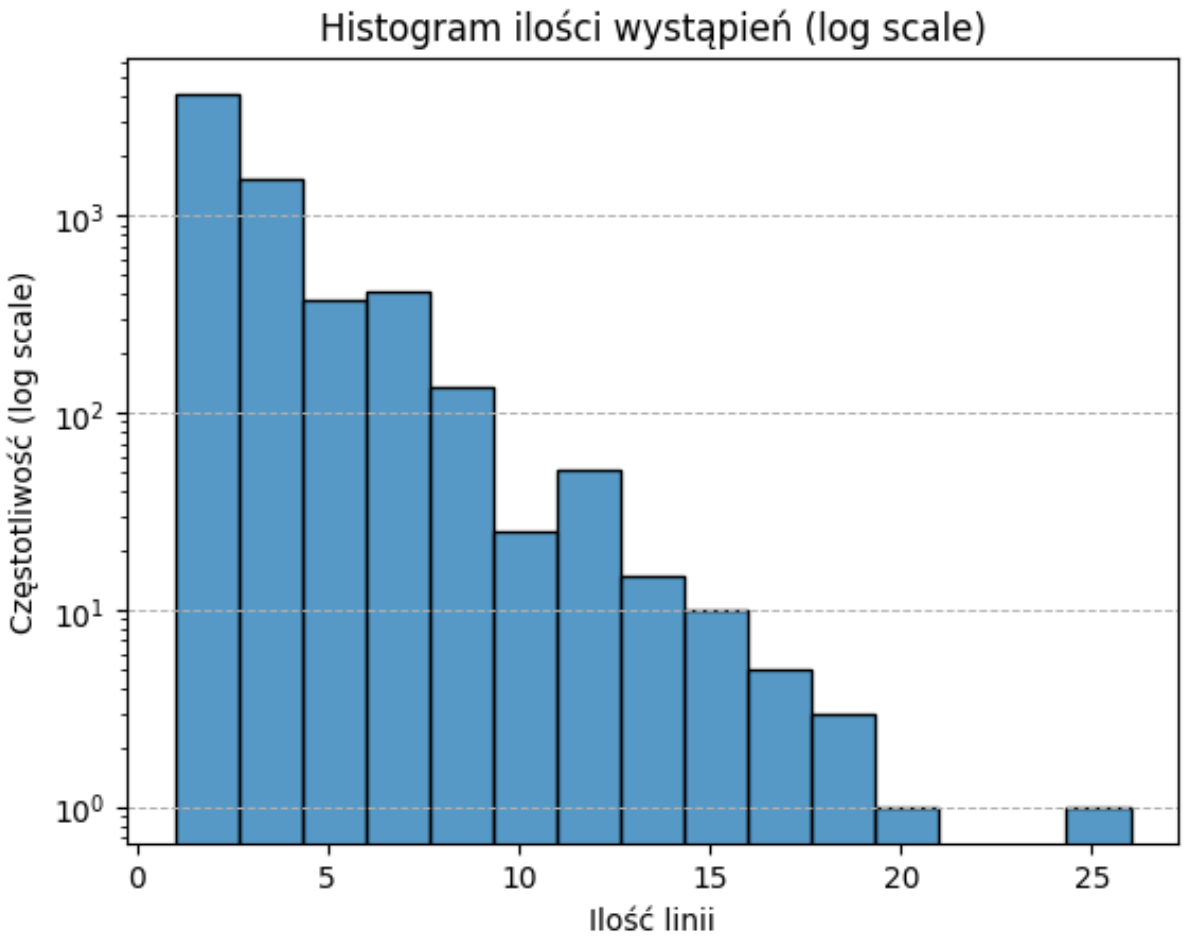
```
buses_list_cleared = []
for i in range(0, len(buses_list)):
    temporary_value = json.loads(buses_list[i]['buses'].text)
    result = temporary_value.get("result", [])
    try:
        new_element = {
            "zespol": buses_list[i]["zespol"],
            "slupek": buses_list[i]["slupek"],
            "buses": [item["value"] for res in result for item in
   res.get("values", [])]
        }
    except AttributeError as e:
        new_element = {
            "zespol": buses_list[i]["zespol"],
            "slupek": buses_list[i]["slupek"],
            "buses": result
        }
    buses_list_cleared.append(new_element)

with open('buses_list_cleared.json', 'w', encoding='utf-8') as f:
    json.dump(buses_list_cleared, f, ensure_ascii=False, indent=4)

missing_info_counter = 0
for element in buses_list_cleared:
    if 'Błędna' in element['buses']:
        missing_info_counter += 1

print(f"API zwróciło {missing_info_counter}
   ({100*missing_info_counter/bus_counter:.2f}%) błędów")
```

Krok 5: Przygotowanie i wyświetlenie wykresów

Aby przygotować dane potrzebne do stworzenia wykresu, należy przejść po liście *buses_list_cleared*, która przechowuje listy wszystkich linii dla danego autobusu, zliczyć ich ilość i zapisać w liście wynikowej *aggregated_buses_counts*. Następnie przy użyciu bibliotek *seaborn* oraz *matplotlib* można narysować wykres otrzymanych danych.

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
from collections import Counter

aggregated_buses_counts = []

for item in buses_list_cleared:                 # Przejdź po liście autobusów
    if isinstance(item.get("buses"), list):     # Jeśli dany autobus posiada
    jakiekolwiek linie...
        aggregated_buses_counts.append(len(item.get("buses")))
        # ... dodaj ich ilość do listy wynikowej

sns.histplot(aggregated_buses_counts, bins=15, kde=False)   # Stwórz histogram

plt.yscale('log')              # Oś y w skali logarytmicznej dla przejrzystości

plt.xlabel('Ilość linii')
plt.ylabel('Częstotliwość (log scale)')
plt.title('Histogram ilości wystąpień (log scale)')

plt.grid(axis='y', linestyle='--', linewidth=0.7)

plt.show()
```

Po wykonaniu kodu w odpowiedzi dostajemy gotowy histogram:



Rysunek 27: Histogram przedstawia rozkład liczby linii autobusowych na przystanek