



Python Backend Developer Script

© 2005-2021 ALX Sp. z o.o. Spółka komandytowa

ALX

Contents

Wstęp	7
Programowanie i jego historia	7
Języki programowania	10
Krótka historia języków programowania	10
Python - pierwsze spojrzenie.	16
Zanim zainstalujemy Pythona	19
Gałęzie	19
Wersje	19
Narzędzia online	19
Instalacja i konfiguracja środowiska	20
Instalacja interpretera języka Python	20
Windows	20
Linux i MacOS	21
Ipython	21
IDE	22
Pycharm	22
Visual Studio Code	23
Stworzenie wirtualnego środowiska	23
Stworzenie pierwszego projektu	24
Podstawy języka Python	24
Uruchamianie programów	25
Interaktywne środowisko	25
Plik z kodem	27
Wsparcie PyCharm	27
Korzystanie z pomocy wbudowanej	28
Błąd i wyjątki	29
Wartość, zmienna, referencja, obiekt	29
help - system pomocy	33
Podstawowe typy danych	37
Liczby	37

Napisy	39
Wartości logiczne	40
Zmienne	42
Rozszerzone operatory przypisania	43
Struktury danych	43
Tuple	43
Listy	46
Zbiory	47
Stowniki	49
Napisy	50
Instrukcja warunkowa	50
Instrukcja pass	52
Pętle	52
Pętla while	52
Pętla for	52
Wyrażenie break i continue	53
Blok else	53
Funkcja range()	54
Wyrażenie listowe i inne	54
Funkcje	55
Programowanie Obiektowe	57
Klasy	58
Metody	58
Metoda specjalna __init__	59
Metody statyczne	60
Metody klasowe	60
Dynamiczne atrybuty	61
Pozostałe metody specjalne	61
Enkapsulacja	62
Dziedziczenie	63
Nadpisywanie metod i funkcja super()	64
Wielodziedziczenie	64
Wyjątki	65
Generowanie wyjątków	65
Obsługa wyjątków	66
Sekcja finally	67
Definiowanie wyjątków	67
Wbudowane wyjątki	68

Organizacja kodu	68
Moduły	68
Poszukiwanie modułów	69
Zależności cykliczne	70
Pakiety	70
Importowanie względne	71
Elementy komunikacji w sieci internetowej	72
Adres URL	72
Protokół HTTP	73
Jak wygląda obsługa żądania?	75
Metody	76
Kody HTML	81
Nagłówki	82
Django	83
Wstęp	83
Instalacja, konfiguracja	85
Virtualenvwrapper	86
Tworzymy projekt	86
Pliki projektu	90
manage.py	90
Jaką rolę spełniają pozostałe pliki?	91
Jak działa aplikacja w Django	91
Aplikacje Django	93
Pliki aplikacji	94
Routing i pierwsze widoki. (urls.py)	95
Dla dociekliwych: Jak działa urls.py	100
Połączenie z bazą	101
SQLite	101
Inne bazy	102
Widok	104
Modele i ORM	105
Obiektowy dostęp do bazy danych	106
pobranie wiersza z bazy	107
Pobieranie grup wierszy z bazy	107
Ćwiczenie	109
Pierwsze testy	109
Szablony	110
wstawianie zmiennych (i niektórych innych wyrażeń)	112

filtry	113
Pliki statyczne	116
Fixture	117
Tworzenie fixture	117
Wczytywanie fixture	117
Używanie fixture w testach	118
Widok pokazujący wybrany wpis	119
Panel Administratora	121
Logowanie do Panelu Administratora.	122
Rejestracja modelu w Panelu Administratora	122
Określenie sformułowania dla liczby mnogiej modelu	122
Zmiana wersji językowej Panelu Admina	123
jednoczesne edycje	123
pliki statyczne	123
Dostosowywanie wyglądu panelu administracyjnego	124
przykrywanie szablonów panelu administracyjnego	125
Tworzenie własnego szablonu dziedziczącego po gotowym dla wybranego modelu	127
Szablony w jednym folderze	127
Drobne poprawki	128
Rzutowanie modelu na napis	128
Wyświetlenie kilku kolumn w widoku listy	135
Linki wpisane na sztywno w szablon	135
Próba pobrania wpisu, którego nie ma	136
Na skróty	137
Ćwiczenie:	138
Ćwiczenie:	138
Stronicowanie, czyli paginacja	139
Relacje pomiędzy obiektami	141
Nowy model - Tagi	141
Ćwiczenie:	142
Ćwiczenia	143
Ćwiczenia:	145
Dziedziczenie szablonów	147
Ćwiczenia:	148
DRY	150
korzystanie z ustawień zapisanych w settings.py	151

Widoki oparte na klasach	152
Ćwiczenie	152
Tworzenie widoku klasowego	152
Mixiny	154
widoki generyczne	154
Dodawanie obrazków	160
udostępnianie statycznych plików	162
udostępnianie statycznych plików	163
Komentarze	170
Zabezpieczenie przed CSRF	170
Middleware	171
Procesory kontekstu	172
Formularze	173
Zwykły formularz - prosty przykład	173
jak korzystać z zabezpieczenia przed CSRF	177
django.forms	178
klasa ModelForm	180
Serwery aplikacji i http	181
Serwer developerski	181
Gunicorn	182
Apache + mod_wsgi	183
Nginx	185
Deployment	190
Dodatki	194
Najprostsza aplikacja w Django	194
Sygnaty	196
Redis	196
Celery	197
REST	197
Client-Server – architektura typu klient-serwer	197
Stateless (bezstanowość)	198
Cache	198
Uniform Interface	198
Layered System	199
Code on Demand	199
Django REST Framework	200

Wstęp

Niniejszy skrypt jest materiałem uzupełniającym do kursu *Programista Backend - Python* prowadzonego przez firmę ALX. Zawiera on informacje omawiane podczas szkolenia przez trenera, niekiedy dodatkowo je uzupełniając i systematyzując. Celem niniejszego skryptu jest przekazanie w możliwie najprostszy sposób wiedzy niezbędnej do programowania w języku Python oraz przedstawienie jego możliwości. Dzięki skryptowi uczestnik szkolenia będzie mógł utrwalic oraz uzupełnić swoją wiedzę zdobytą podczas kursu, jak i z łatwością odtworzyć środowisko pracy na swoim komputerze, czy powtórzyć przeprowadzane tam ćwiczenia.

Zanim przejdziemy jednak do konfiguracji niezbędnych nam narzędzi oraz nauki składni języka Python, musimy zrozumieć czym jest samo programowanie oraz czym charakteryzuje się wybrany przez nas język programowania.

Programowanie i jego historia

Programowanie jest procesem polegającym na projektowaniu, pisaniu, testowaniu oraz utrzymywaniu programów komputerowych. Pisanie programów komputerowych, nazywane także implementacją, procesem developmentu odbywa się poprzez tworzenie kodu źródłowego w wybranym języku programowania. Programista, trzymając się zasad składni wybranego języka programowania, tworzy kod źródłowy programu, który później uruchamiany jest przy wykorzystaniu narzędzi odpowiednich dla danego środowiska.

Prapoczątków programowania można by dopatrywać się w konstrukcji różnego rodzaju maszyn, automatów, których działanie sterowane było na różne wymyślne sposoby, między innymi przez różne układy kół zębatych - jak w starożytnym mechanizmie z Antykithiry, czy w pascalinie - mechanicznym sumatorze skonstruowanym około 1645 roku przez Błażeja Pascala. Ulepszoną jej formę zaproponował około 1670 roku Gottfried Wilhelm Leibniz, który wprowadził tam rodzaj cylindrycznej zębatki, zwanej też bębnem schodkowym lub po prostu kołem Leibniza. Jego konstrukcja umożliwiała ponadto mnożenie, dzielenie, a nawet wyciąganie pierwiastków kwadratowych

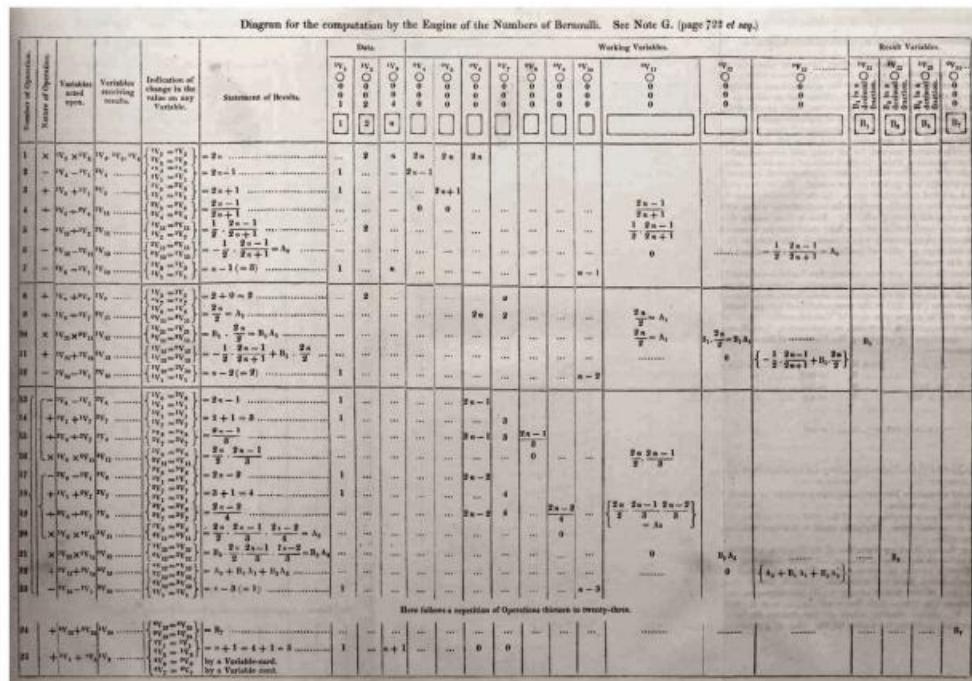


Figure 0.1: image-20210611095157549

W XIX wieku pojawił się szereg mechanicznych kalkulatorów. Wśród nich znalazły się dwie maszyny rachunkowe (1815 i 1817) autorstwa a Abrahama Jakub Sterna, arytmometr Charlesa Xaviera Thomasa z 1820 r. czy szereg maszyn różnicowych, skonstruowanych w latach 1822-1849 przez Charles'a Babbage'a i jego współpracowników. Ostatnie tego typu mechaniczne kalkulatory konstruowane były jeszcze w XX wieku – przykładem może być kieszonkowy kalkulator z 1948 r. autorstwa Curta Herzstarka. Obok zastosowań naukowych czy rachunkowych usprawnienia takie pojawiały się też w przemyśle – na przykład w maszynach tkackich sterowanych za pomocą specjalnych kart z dziurkami – zwanych od swego wynalazcy maszynami żakardowymi. Około 1834 roku Babbage wpadł na pomysł skonstruowania bardziej uniwersalnej maszyny – takiej, która mogłaby wykonywać różne operacje w zależności od tego, jak zostałaby zaprogramowana. Był to projekt wyprzedzający swoją epokę o ponad 100 lat – dopiero wtedy udało się skonstruować maszynę, która umożliwiałaby realizację takiego zadania. W tym jednak miejscu – obok projektu samej maszyny – powstały też pomysły na pierwsze algorytmy, które mogły być realizowane przez taką maszynę i w tym sensie były to pierwsze programy komputerowe. Pierwszym opublikowanym programem komputerowym był diagram wykonany przez Augustę Adę King, hrabinę Lovelace, córkę lorda Byrona, który został

opublikowany w 1842 r w jej uwagach do artykułu L.F. Menabrea Sketch of The Analytical Engine Invented by Charles Babbage. (Figure 2).

Takie były początki. Pierwsze w pełni funkcjonalne komputery zaczęły powstawać dopiero w XX wieku, głównie na potrzeby militarnych zastosowań. Jednym z takich wyzwań było obliczanie trajektorii balistycznych na potrzeby artylerii. Opracowanie takich tablic było kluczowe dla skuteczności ostrzału artyleryjskiego. Zadanie żmudne i pracochłonne – idealne dla komputera. Z niezłym powodzeniem zajął się tym komputer Harvard Mark I – skonstruowany pod okiem Howarda Aikena. Innym zastosowaniem była kryptografia, a w szczególności łamanie szyfrów wroga – tu szczególne załugi miał komputer Colossus Mark II, ukończony w 1943 r., w którego konstruowanie zaangażowany był Alan Turing – jedna z kluczowych postaci informatyki. Tuż przed II Wojną Światową i w czasie jej trwania powstało jeszcze kilka innych unikalnych komputerów, takich jak konstrukcje Z1, Z2 i Z3 Konrada Zuse, czy komputer Johna V. Atanasoffa. U schyłku wojny powstaje też jeden z najstaryjszych komputerów tego okresu – ENIAC. Okres, który nastąpił po wojnie, a szczególnie ostatnie 30-40 lat możemy określić jako okres globalnej komputeryzacji. Jeszcze kilkadziesiąt lat temu komputery były bardzo rzadkimi i często wielkimi urządzeniami. Były to bardzo rzadkie dobro, a skorzystać z niego mogli tylko nieliczni. Nieliczni też potrafili to robić. Dziś komputery mamy niemal wszędzie. Wyszły one z wielkich pomieszczeń i szaf i przeszły na nasze biurka, a później wślizgnęły się do naszych kieszeni. To, że telefony, tablety, a nawet zegarki mają dziś moc obliczeniową dużo większą niż komputery z XX wieku, nikogo dziś już nie dziwi. W końcu w obecnych czasach komputery znajdziemy nawet w inteligentnych lodówkach czy maszynach do gotowania. Liczba użytkowników komputerów i ich różnych form jest więc ogromna. Coraz częściej przydaje się też umiejętność programowania takich urządzeń – w pracy, w szkole czy choćby dla zabawy. Choć programowanie nadal kojarzy się z wiedzą tajemną, to jednak w dużym stopniu jest to skojarzenie niesłuszne. W dzisiejszych czasach programowanie jest w zasięgu każdego z nas. Fakt jest, że przez dziesięciolecia zarezerwowane było jednak tylko dla wąskiej grupy wykształcanych specjalistów. Wspomniany ENIAC programowano początkowo poprzez łączenie kablami różnych jego modułów – to było wyzwanie bo wymagało doskonałej znajomości samego komputera i zrozumienia zasad jego działania. Innym sposobem było zastosowanie specjalnych kart z dziurkami, przy użyciu których wprowadzano do komputera instrukcje. Przygotowanie takich kart to też był nie lada wyczyn, szczególnie gdy trzeba było przygotować ich kilka, a nawet i kilkadziesiąt tysięcy. Sposób programowania początkowo był więc bardzo związany z samym komputerem. A że komputery bardzo się między sobą różniły – to każde takie urządzenie programowało się inaczej. Wraz z rozwojem technologii i wprowadzaniem nowych typów komputerów pojawiło się zapotrzebowanie na nowe rozwiązania w dziedzinie programowania.

Języki programowania

Na przestrzeni kilkudziesięciu lat historii informatyki powstało wiele języków programowania. Niektóre z nich zostały już zapomniane i zastąpione nowszymi rozwiązaniami, jednak wciąż bardzo duża grupa języków programowania jest aktywnie używana. Istnienie wielu języków programowania jednocześnie spowodowane jest licznymi aspektami.

Po pierwsze, każdy język programowania różni się zarówno składnią (tzn. strukturą zapisu kodu źródłowego) jak i ekosystemem narzędzi oraz rozwiązań towarzyszących danemu językowi. Dzięki temu niektóre języki programowania sprawdzają się lepiej w danej klasie problemów niż inne. Zadaniem programisty jest dobranie najlepiej pasującego języka programowania oraz towarzyszących mu narzędzi do rozwiązania danego problemu.

Kolejnym powodem istnienia wielu języków programowania jest aspekt historyczny. Wymyślenie nowego, lepszego, bardziej nowoczesnego języka programowania, nie oznacza bezproblemowego przejścia na nowo powstałe rozwiązanie przez wszystkich programistów i firmy implementujące swoje produktu w innych technologiiach. Porzucanie swoich dotychczasowych rozwiązań ze względu na pojawienie się kolejnego języka programowania byłoby oczywiście bez sensu ze względów biznesowych.

Do grona najbardziej popularnych języków programowania (wg. rankingu TIOBE) należą C, C++, Java, C# i Python.

Krótką historią języków programowania

Pierwsze komputery programowano w języku maszynowym, czyli takiej formie, która była bezpośrednio zrozumiała dla procesora. W kodzie takim instrukcje i ich argumenty zapisywane są w postaci liczb binarnych, reprezentujących elementarne instrukcje takie jak operacje arytmetyczne czy pobieranie i zapisywanie w konkretnych miejscach danych. W 1948 r. brytyjska matematyczka Kathleen Booth opracowała język asemblera, który ułatwiał pisanie kodu. W asemblerze zastąpiono liczbowe kody instrukcji procesora krótkimi słowami. Takie słowa nazywa się mnemonikami. Są one znacznie czytelniejsze dla programisty – np. słowa ADD czy SUB od razu kojarzą się z odpowiadającymi im instrukcjami (porównaj angielskie słowa add i subtract). Specjalny program zwany assemblerem zamieniał następnie takie mnemoniki na odpowiadające im kody w kodzie maszynowym. Wymagało to trochę więcej pamięci komputera, ale zapewniało znacznie poprawioną czytelność kodu. Warto zauważyć, że Kathleen to kolejna kobieta, która wniosła duży wkład w rozwój języków programowania. W początkach informatyki było takich kobiet więcej.

W 1949 John Mauchly (jeden z twórców ENIAC'a) zaproponował język Brief Code, dziś znany bardziej jako Short Code. Był to jeden z pierwszych języków wysokiego poziomu (będzie jeszcze o tym mowa w dalszej części skryptu), charakteryzujący się dodatkowo tym, że był interpretowany (o

tym też później). Choć wykonanie napisanego w nim kodu było wolniejsze niż w przypadku kodu maszynowego, to jednak samo napisanie programu przez programistę było znacznie łatwiejsze i szybsze.

W 1954 roku John Backus z IBM stworzył język FORTRAN. Idee w nim zawarte do dziś wywierają wpływ na języki programowania. Język ten zyskał dużą popularność, szczególnie w świecie nauki, i do dziś wielu naukowców pisze w nim programy. W tym okresie odczuwano coraz większą potrzebę, by programowanie wychodziło ze ścisłe naukowego świata. Odpowiedzią na to zapotrzebowanie był język COBOL. Zaprojektowano go z myślą o tworzeniu aplikacji biznesowych. Język powstał w 1960 r. i do dziś jest używany – szczególnie w różnych aplikacjach związanych z działaniem banków, gdzie kluczowa jest stabilność i bezpieczeństwo. Kolejnym krokiem w upowszechnianiu programowania był język Algol oraz bazujący na nim i na FORTRANIE język BASIC. Ten ostatni powstał w roku 1964 i adresowany był do szerokiego grona odbiorców. Był przystępny i świetnie sprawdzał się w różnych amatorskich i półprofesjonalnych zastosowaniach. W Polsce najbardziej znane były jego odmiany dostępne dla popularnych w latach 80-tych i początku lat 90-tych XX wieku komputerów Commodore, Amstrad, Atari czy ZX Spectrum. Kolejne ważne języki to stworzony w 1969 r. przez Niklaus'a Wirth'a Pascal – napisany z myślą o studentach i nauce programowania – oraz powstały w tym samym roku język C, który szybko stał się wiodącym językiem programowania systemów operacyjnych i różnego rodzaju aplikacji. Powstał w nim chociażby system Unix – z którego wywodzą się późniejsze implementacje takie jak GNU/Linux czy macOS. Dał on też początek takim językom jak stworzony w 1979 r. C++ czy powstały w 2000 r. C# (czytaj: si-szarp). W oparciu o składnię C++ oraz pewnych mechanizmów wziętych z jeszcze innego języka – Smalltalka – powstał jeden z najpopularniejszych obecnie języków programowania – Java. Historie te można mnożyć i mnożyć. Samych języków programowania powstało już bowiem setki, a nawet tysiące, często bazując na swoich poprzednikach i zapożyczając z nich wybrane cechy, mieszając z innymi, modyfikując i dostosowując do konkretnych potrzeb. A jak było z Pythonem i jak to się stało, że jest teraz tak popularny?



Figure 0.2: image 20210611095118826

Twórcą Pythona jest Holender Guido van Rossum (Figure 3). W latach 80-tych XX wieku pracował on w instytucie matematyczno-informatycznym (CWI) w Amsterdamie. Rozwijał tam między innymi język ABC – nastawiony na cele edukacyjne i prototypowanie. Projekt ten mimo wielu zalet nie odniósł sukcesu. Van Rossum zaczął więc pracować w CWI nad innymi rzeczami. Zauważ, że większość pracy wykonanej pod nowy projekt (zwany Ameba) stanowią skrypty powłoki systemowej (tzw. shell scripts) albo programy napisane w C. Guido dostrzegał pewne stabe strony obu tych obszarów. Pomyślał o stworzeniu języka, który byłby czymś pomiędzy tymi rozwiązaniami. Język taki korzystałby z jego doświadczeń z pracy nad ABC i opieratby się na konwencjach i infrastrukturze Unixowej, ale bez wiązania się z samym Unixem. Pozwalałby na napisanie znacznie bardziej związkowych programów, a do tego dużo bardziej czytelnych, dzięki czemu mógłby być atrakcyjny dla wielu programistów związanych ze środowiskiem Unix/C – po prostu mogliby szybciej i wygodniej pisać wiele nowych rzeczy. Jak sam wspomina, w grudniu 1989 roku nadarzyła się okazja, by te przemyślenia zrealizować. Miał wtedy trochę wolnego czasu w związku ze świąteczną przerwą, postanowił więc napisać interpreter nowego języka. Od tego czasu język odbył długą drogę. To w końcu blisko 30 lat rozwoju. Guido dość szybko zdecydował się na to, by język ten stał się wolnym oprogramowaniem – tzn. takim, który byłby dla każdego dostępny za darmo. Obecnie rozwojem języka zarządza specjalna fundacja – Python Software Foundation, a wokół niego na całym świecie zebrała się ogromna społeczność, której i Ty możesz być częścią.

Python, w dużej mierze dzięki swojej społeczności, stał się językiem o bardzo szerokich zastosowaniach. Powstała ogromna ilość dodatkowych narzędzi i modułów – z bardzo różnych dziedzin.

Dzięki temu Pythona używa się nawet w tak zaskakujących obszarach jak biologia czy muzykologia. Świetnie sprawdza się jako język inżynierów, obliczeń naukowych. Można w nim szybko i wydajnie tworzyć zarówno proste skrypty, jak i skomplikowane aplikacje – zarówno te z interfejsem graficznym, jak i te dostępne w wierszu poleceń. Można też tworzyć w nim gry, a od biedy nawet aplikacje na Androida. Specjalna wersja MicroPython używana jest na płytach rozwojowych, na bazie których możemy konstruować różne urządzenia – przykłady takich platform to pyboard czy bardziej edukacyjny Micro:bit. Dwa najpopularniejsze obecnie zastosowania to aplikacje internetowe oraz bardzo popularny, szeroko pojęty obszar data science – to tu Python odnotowuje największy wzrost popularności i jest obecnie jednym z wiodących języków tych technologii. Świetnie nadaje się też do celów edukacyjnych, czego wyrazem jest choćby i to, że można w Polsce zgłaszać go jako preferowane środowisko na maturze z informatyki.

Znamy już побieżnie zarys drogi, jaką odbyła informatyka od swoich początków – do czasów stworzenia Pythona i innych współczesnych języków programowania. Jak już wspomniano, obecnie mamy ich całą masę. Spójrzmy więc jeszcze raz ogólniej na samą koncepcję języka programowania, zdefiniujmy jakieś kryteria, które pomogą nam języki od siebie odróżniać, i na koniec umieścmy gdzieś w tym wszystkim Pythona.

Przede wszystkim – do czego taki język służy? Najogólniej mówiąc możemy powiedzieć, że język programowania to sposób opisu programu komputerowego, czyli zapisu pewnych instrukcji, obliczeń, które na jego podstawie wykona komputer. Instrukcje takie, gdy zapisane są w sposób czytelny dla człowieka, nazywamy często kodem źródłowym. Z kolei postać zrozumiałą dla komputera to tzw. kod maszynowy. By komputer zrozumiał nasz kod, musi on zostać podany w sposób zrozumiały dla maszyny lub musi być na taką postać przetłumaczony. Uczęc się jakiegoś języka programowania (poza językiem maszynowym oczywiście), uczymy się pisać kod źródłowy. Uczymy się też wykorzystywać dostarczone z językiem narzędzia do tworzenia kodu maszynowego lub po prostu uruchomiania naszych programów. Nauka polega na poznaniu pewnych zasad, wokół których zbudowany jest dany język. Zasady te opisują składnię języka – czyli dostępne symbole oraz reguły grupowania tych symboli w większe struktury. Kolejnym elementem jest tzw. semantyka – czyli precyzyjna definicja znaczenia i funkcji, jakie pełnią poszczególne symbole w programie. Ważną częścią języka są też typy danych. Czym innym jest napis, a czym innym np. liczba całkowita. To są właśnie przykłady typów. Często też wraz z językami przychodzi tzw. biblioteka standardowa – czyli taki zestaw gotowych narzędzi, które pozwalają na realizację różnych często spotykanych zadań.

W związku z różnorodnością języków często dzielimy je na różne grupy ze względu na pewne cechy. Jedną z takich cech nazywamy poziomem. Mówimy o językach niskopoziomowych i wysokopoziomowych. Poziom ten można sobie wyobrazić jako pewną odległość od „sprzętu”. Języki blisko związane ze sprzętem są niskopoziomowe, a te, które o sprzęcie za wiele wiedzieć nie muszą, nazywane są wysokopoziomowymi. Języki niskiego poziomu wymagają dobrej znajomości architektury komputera i są tym samym bardziej zbliżone do kodu maszynowego. Niskopoziomowy

program napisany na konkretny komputer może być trudno przenieść na inny, szczególnie gdy będzie się on znaczco różnić budową i/lub systemem operacyjnym. Konieczne może wtedy okazać się przepisywanie go, co może być bardzo kosztowne. Z drugiej jednak strony tak napisany program może działać bardzo szybko – ponieważ jest zoptymalizowany pod kątem danej jednostki. Języki wysokiego poziomu są niejako oderwane od sprzętu. To specjalne narzędzia dbają o to, by potem przygotować wersję zrozumiałą dla konkretnego komputera. W związku z tym ten sam program w większości przypadków zadziała na różnych maszynach, o ile tylko jest dostępny dla nich ten mechanizm tłumaczący.

Innym podziałem języków jest sposób wykonywania programu. Wiąże się to nieroźłącznie z przygotowaniem wersji zrozumiałej dla komputera – szczególnie dla języków wysokiego poziomu. Tłumaczeniem kodu źródłowego na język maszynowy zajmują się specjalne programy, które z grubsza można podzielić na kompilatory i interpretatory. Kompilatory to takie programy, które tłumaczą kod napisany w jednym języku na odpowiadający mu kod w innym języku. W naszym przypadku chodzi o język maszynowy. Wynikiem komplikacji jest plik wykonywalny. Na Windows mogą to być np. pliki z rozszerzeniem .exe. Interpretatory z kolei tłumaczą i wykonują kod na bieżąco. Tzn. interpreter bierze jakąś linijkę kodu z pliku źródłowego, tłumaczy ją, przetłumaczona linijka jest wykonywana, brana kolejna linijka i proces jest powtarzany. Każde z tych rozwiązań ma swoje mocne i słabe strony. Programy napisane w językach kompilowanych będą na ogół miały większą wydajność, czyli będą działały szybciej. Ale z drugiej strony taki program jest ponownie ściślej związany z komputerem, na którym był kompilowany. Trudniej jest więc go przenosić pomiędzy różnymi komputerami – w wielu przypadkach trzeba by na nich ponownie przeprowadzać proces komplikacji. Języki interpretowane co prawda mogą dawać programy o nieco mniejszej wydajności (stosuje się różne sztuczki, by te różnice niwelować), ale łatwiej jest je przenosić pomiędzy różnymi platformami – wystarczy, że mamy kod źródłowy i interpreter na daną maszynę i to on już zajmie się całą robotą.

Kolejnym rozróżnieniem mogą być paradigmaty programowania, na których oparty jest język. Paradymat to, mówiąc prościej, pewien sposób widzenia, podejścia. Określone podejście może lepiej sprawdzać się w rozwiązywaniu jakiejś grupy problemów. Z najczęściej stosowanych paradigmatów warto wymienić takie, jak programowanie funkcyjne, obiektowe, strukturalne. Czym one dokładnie są, nie będziemy teraz tłumaczyć, ale warto wiedzieć, że występują i że przykładów jest jednak znacznie więcej. Niektóre języki umożliwiają programowanie w jednym wybranym paradymacie, inne zaś pozwalają używać różnych.

Jeszcze innym rozróżnieniem jest tzw. sposób kontroli typów. Co to są typy, już wspomnieliśmy – to pewne wartości, które w sposób istotny mogą się od siebie różnić. W sposobie ich kontroli rozróżnia się takie pojęcia jak typowanie statyczne i dynamiczne, jak również typowanie słabe i silne. By to zrozumieć, wyobraź sobie dwa pojemniki i kilka klocków o różnych kształtach. Pierwsze pudełko nazwiemy „typowaniem dynamicznym”. Możemy tam włożyć jakiś klocek, a następnie zamienić go na inny – o innym kształcie. Drugie pudełko nazwiemy „typowaniem statycznym”. Wkładamy tam klo-

cek, a następnie zalewamy pudełko np. cementem – to specjalny programistyczny cement i schnie w ułamek sekundy. Teraz możemy klocek wyjąć i zostaje po nim miejsce – taka forma. W to miejsce możemy włożyć tylko taki klocek, który ma te same kształty, co poprzedni (przyjmijmy, że klocki są tak dobrane, by inaczej się nie dąć), i to jest właśnie istota typowania statycznego. Statyczny – oznacza niezmienność. Dynamiczny to jego przeciwnieństwo – czyli zmienność. W programowaniu też możemy mieć takie pojemniki – nazywamy je często zmiennymi. Taka zmienna ma dwie podstawowe cechy – wartość i typ. Weźmy sobie np. liczbę 5 i słowo „Ala”. W pierwszym przypadku wartość to 5, a typ to liczba. Bo 5 jest liczbą. W przypadku „Ala” typ to napis, a wartość to trzy litery, które się na ten napis składają. W programowaniu napisy najczęściej umieszcza się pomiędzy znakami apostrofu lub cudzysłowu. Czasem częścią napisu są też liczby, np. „R2D2”. Napisy możemy ze sobą łączyć. Można to zapisać tak „Robot” + „R2D2”, czego wynikiem będzie napis „RobotR2D2”. Znak „plus” często nazywa się w takich sytuacjach „operatorem”. Jeśli teraz zadziałam tym operatorem na dwie liczby, np: 5 + 5, to spodziewam się raczej wyniku dodawania – czyli 10. Gdybym jednak napisał „5” + „5”, to w myśl przykładu z robotem spodziewałbym się wyniku „55”. Widzisz więc, że stosując operator +, oczekuję innego wyniku dla różnych typów. Inaczej dla liczb, a inaczej dla napisów. A co się stanie, kiedy spróbuję zrobić tak: „5” + 5? Można ten problem rozwiązać na kilka sposobów. W językach z tzw. silnym typowaniem w takich przypadkach w programie zostanie wywołany błąd. Tzn. język powie nam – przepraszam, ale nie wiem, co masz na myśli. Języki ze słabym typowaniem zamienią któryś z typów na inny. Zapewne w większości języków ze słabym typowaniem dostaniemy tu w wyniku napis „55”. Podejście do typowania może mieć duży wpływ na sposób, w jaki będą działały nasze programy. Dlatego zawsze warto mieć świadomość tego, jak to działa w danym języku. Kolejnym kryterium podziału języków jest ich przeznaczenie. Programy mogą mieć różne zastosowania. Może to być np. zaprogramowania jakichś urządzeń – takich jak telewizory, telefony czy komputery samochodowe. Może chodzić o jakieś skomplikowane obliczenia matematyczne. Mogą to być też jakieś aplikacje desktopowe, takie jak różnego rodzaju edytory, kalkulatory czy chociażby gry. Mogą to być również aplikacje internetowe. Każdy z takich obszarów ma swoje specyficzne wymagania, do których pasować mogą inne języki programowania. To taka specjalizacja. I tak np. większość aplikacji działających na Androidzie napisana będzie w Javie. Statystycy bardzo często sięgają po język R. Przez wiele lat najczęstszym językiem wykorzystywanym do tworzenia aplikacji internetowych był PHP, który do dziś jest bardzo mocny w tym obszarze. Są też języki, które dobrze radzą sobie w różnych obszarach. Mówimy wtedy, że są to języki ogólnego zastosowania. Przymierząc się do danego projektu, warto zastanowić się nad tym podziałem – po to, by dobrać do niego odpowiednie narzędzie. W początkowej fazie dobrym pomysłem jest sięgnięcie po język ogólnego zastosowania.

Świętym punktem wyjścia, a czasem w zupełności wystarczającym środkiem jest właśnie Python. Patrząc na powyższe kryteria, możemy scharakteryzować go jako język wysokiego poziomu z dynamicznym, ale silnym typowaniem, pozwalający na stosowanie różnych paradygmatów, a do tego o ogólnym zastosowaniu. Dodatkowo cechuje go bardzo przejrzysta i stosunkowo prosta składnia, rozbudowana o wiele nowoczesnych funkcji, takich jak funkcje dekoratorów, generatorów itp.

dowana biblioteka standardowa oraz bardzo bogaty i ciągle rozwijający się zbiór dodatkowych bibliotek i modułów, po które możemy sięgnąć, gdy przyjdzie taka potrzeba.

Python - pierwsze spojrzenie.

W oparciu o powyższe rozróżnienia możemy scharakteryzować język Python jako interpretowany język programowania ogólnego zastosowania, wysokiego poziomu, z silnym i dynamicznym typowaniem.

Język Python wspiera wiele paradygmatów programowania, tzn. podejść do wytwarzania oprogramowania, dlatego jest językiem uniwersalnego zastosowania. Dynamiczne typowanie, automatyczne zarządzanie pamięcią oraz rozbudowana biblioteka standardowa ułatwiają szybką implementację projektów.

Python rozwijany jest przez organizację non-profit Python Software Foundation jako projekt open source i wspiera on wszystkie najważniejsze systemy operacyjne.

Głównymi zastosowaniami języka Python jest programowanie serwisów webowych, obliczenia naukowe, analiza danych oraz narzędzia developerskie.

Przyjrzyjmy się teraz odrobinę samemu językowi, dokonując krótkiego przeglądu jego składni, instrukcji i typów. To zaledwie wstęp do nauki samego języka. W kolejnych rozdziałach będziemy przyglądać się tym zagadnieniom bliżej. Wtedy też poznamy bardziej złożone zagadnienia, takie jak definiowanie własnych funkcji czy typów (znanych w Pythonie jako klasy). Tradycyjnie już naukę programowania zaczyna się od wypisania na ekranie tekstu „Hello World”. W Pythonie te zadanie jest bardzo proste. Poniżej zaprezentuję linijkę kodu, która je zrealizuje. Umówmy się, że takie fragmenty kodu będziemy nazywać listingami. Np:

```
1 print('hello world!')
```

Taki kod należałoby jakaś uruchomić, ale nie będziemy tego robić w tym miejscu. Na razie po prostu poczytaj. Wróćisz do tych przykładów po instalacji Pythona i zapoznaniu się ze sposobami uruchamiania jego kodu. W tym miejscu zwróć uwagę na coś innego. Po pierwsze, na to, że pojawiło się magiczne słówko `print`, nawiasy i coś, co już może być znajome – czyli napis `Hello World!`. Słówko `print` to wbudowana w Pythona *funkcja*, która odpowiada za wypisywanie na ekran (by być bardziej precyzyjnym na standardowe wyjście). To, co ma być wypisane, podawane jest w nawiasie – mówimy, że jest to argument tej funkcji. W Pythonie jest więcej wbudowanych funkcji, które poznamy. Można też definiować własne. Ważne jest to, że gdy używamy po ich nazwie nawiasów `()`, to mówimy, że

wywołujemy jakąś funkcję. Kod na powyższym listingu to właśnie przykład wywołania funkcji `print`. Jeśli coś będzie między nawiasami, to mówimy, że wywołujemy ją z argumentami, jeśli nic nie będzie, to wywołujemy ją bez argumentów. Niektóre funkcje mogą wymagać podania jakichś argumentów, a inne nie. O tym sobie jeszcze powiemy – na tym etapie warto zapamiętać, że po prostu jest coś takiego jak funkcja i że możemy ją wywołać. Jako argument do funkcji `print` możemy przekazać też jakąś zmienną. Czasem jest to bardziej wygodne. Co to są te zmienne? Pokażmy to na poniższym listingu. Dodatkowo wprowadzę w nim komentarze. Komentarzem są napisy, które następują po znaku `#`. Takie napisy nie mają wpływu na działanie programu – służą jako swego rodzaju notatki. Tutaj po prostu ponumerowałem przy ich pomocy linie.

```
1      x = 10          # 1
2      y = 20          # 2
3      wynik = x + y # 3
4      print(x+y)    # 4
5      print(wynik)   # 5
```

W linii 1 i 2 utworzyłem zmienne `x` i `y` i przypisałem do nich wartości – odpowiednio 10 i 20. W linii 3 też utworzyłem zmienną i nazwałem ją `wynik`. Tym razem przypisałem do niej wartość wyrażenia `x + y`. W liniach 4 i 5 wywołuję znowu funkcję `print`. Raz jako argument podane jest wyrażenie, a za drugim razem zmienna, w której wcześniej zapisana została wartość wyrażenia. W obu przypadkach funkcja `print` wypisze taki sam wynik. W Pythonie mamy kilka podstawowych typów zmiennych. By za bardzo nie komplikować spraw, w poniższym listingu przedstawię tylko 4 wybrane – najprostsze typy. Inne poznamy w kolejnych rozdziałach.

```
1      liczba_calkowita = 1000      # 1 int
2      liczba_zmiennoprzecinkowa = 3.14 # 2 float
3      wartosc_logiczna = True       # 3 bool
4      napis = "Jestem napisem"     # 4 str
5      print(type(liczba_calkowita)) # 5
6      print(napis + liczba_calkowita) # 6 ???
```

W linii 5 pokazałem przykład tego, jak możemy sprawdzić typ danej zmiennej. Służy do tego kolejna wbudowana w Pythona funkcja – `type`. W komentarzach po numerze linii umieszczone są nazwy tych typów stosowane w Pythonie. Te wartości zobaczymy, gdybyśmy wywołali linijkę 5 – podając jako argument funkcji `type` odpowiednie nazwy zmiennych. Zauważ, że tym razem argumentem funkcji `print` jest wywołanie funkcji `type`. Tak też można. Kiedyś jeszcze do tego wrócimy. Pewnie już wiesz ze szkoły, co to są liczby całkowite i zmiennoprzecinkowe. Z punktu widzenia komputera to dość ważne rozróżnienie. Nie będziemy teraz się nad tym jednak specjalnie rozwodzić. Wiedz jednak, że Python umie te dwa typy ze sobą „pogodzić”. Choć są to formalnie różne typy, to stanowią one podzbior większej rodziny – tzw. typów numerycznych i dzięki temu możemy przeplatać operacje na nich. Na przykład jeśli w Pythonie dodamy do siebie 1 + 2.1, to wynikiem będzie liczba 3.1. Gdybyśmy jednak chcieli wykonać operację z linii 6 – nasz program zakończyłby się błędem. To, co jest też tutaj

ważne, to to, że separatorem dziesiętnym liczb jest kropka, a nie przecinek. To po prostu taka konwencja. W części świata stosuje się kropkę, w części przecinek. W informatyce niemal zawsze jest to kropka. Co ciekawe, do tej rodziny typów numerycznych zalicza się też typ zwany wartością logiczną (bool). W Pythonie mamy dwie wartości logiczne: True – czyli prawda i False – czyli fałsz. Przydają się one w wielu miejscach w programowaniu, stąd znajomość zasad nimi rządzących jest bardzo ważna. Być może znasz już elementarną logikę z zajęć matematyki; to właśnie jej zasady rządzą tymi wartościami. Nauczmy się je wykorzystywać.

Kolejnym ważnym elementem języków programowania są instrukcje warunkowe. Stosujemy je wtedy, kiedy chcemy, by nasz program zachował się inaczej, gdy jakiś warunek jest spełniony, a inaczej, kiedy spełniony nie jest. Powiedzmy, że w jakimś miejscu programu obliczyliśmy sobie zmienną `x` – kryje się pod tą zmienną dowolna liczba. Chcemy sprawdzić, czy jest większa od 9, czy nie. Możemy to zrobić w ten sposób (... oznaczają, że coś w tym kodzie powinno być wcześniej – tutaj powinno pojawić się przypisanie wartości do zmiennej `x`)

```
1 if x > 9:  
2     print("OK")  
3 print("Teraz wykonam jakąś dalszą część programu")
```

W tych paru linijkach poniżej dzieje się tak naprawdę bardzo wiele. Omówmy to po kolei.

Po pierwsze, pojawiają się pewne słowa, takie jak `if` – oznacza ono „jeżeli”. Program możemy więc sobie przeczytać na głos:

```
1 ż  
2 Jeeli x jest większe niż 9, wypisz "OK".
```

Po słowie `if` powinien nastąpić więc jakiś warunek. Coś chcemy tam sprawdzić. Ten warunek tak naprawdę powinien sprowadzić się do jakieś wartości logicznej – do prawdy (True) lub fałszu (False). Następnie informujemy Pythona, że kończymy warunek znakiem `:`. Po tym znaku powinien się pojawić blok kodu, który wizualnie jest przesunięty nieco w prawo. Jeśli warunek jest spełniony, to będzie wykonywać właśnie ten przesunięty fragment kodu. Jeśli warunek nie jest spełniony, to interpreter zignoruje ten blok i przejdzie do kolejnej linii, która takiego przesunięcia nie ma. Jeśli w `x` będzie wartość 1, to fragment kodu z listingu zaowocuje jedynie wypisaniem „Teraz wykonam jakąś dalszą część programu”. Jeśli w `x` będzie np. wartość 100, to wcześniej pojawi się napis OK.

Instrukcje w programowaniu często grupuje się w bloki kodu, które powinny być wykonywane razem. Wcięcia i stosowanie nowej linii to taki Pythonowy sposób na organizację bloków kodu. W innych językach bloki kodu często grupuje się przy użyciu nawiasów, a instrukcje oddziela się od siebie średnikami. Programiści mają wtedy dużą dowolność w sposobie zapisu. Dzięki sztuczce z wcięciami i nowymi liniami w Pythonie nawiasów używać nie musimy. Wymusza to jednocześnie czytelniejsze i bardziej zunifikowane zapisanie kodu. Każdy programista musi trzymać się tych reguł, inaczej program nie zadziała. Jest to pewne ograniczenie, ale skutkuje ono tym, że program napisany przez jed-

nego programistę znacznie łatwiej czyta się innemu programiście. Jest to wskazywane jako jedna z największych zalet Pythona. Mechanizm stosowania wcięć nazywany jest w świecie Pythona indentacją – od angielskiego słowa indent oznaczającego właściwie wcięcie lub akapit. Pisząc programy w Pythonie, z pewnością szybko załapiesz, jak tych wcięć poprawnie używać.

Zanim zainstalujemy Pythona

Gałęzie

Python ma dwie, różniące się od siebie istotnie gałęzie. Python2 i Python3. Od kilku lat (a mamy rok 2021) to jednak Python3 jest wiodącą gałęzią a jego poprzednia wersja używana jest głównie w starszych, wymagających utrzymywania projektach. Nowe projekty tworzy się już przy pomocy Pythona3. Często też i te stare przepisuje się na nową wersję. Wersja 2 nie jest już wspierana przez twórców Pythona od 01.2020.

Wersje

Wersje Pythona oznaczane są w notacji:

X.Y.Z

- X oznacza gałąź: 2 lub 3
- Y oznacza wersję (*version*)
- Z oznacza mikro wersję (*micro version*)

Wersje i mikrowersje to po prostu kolejne liczby. Im większa, tym późniejsze wydanie. Mikro wersja (Z) to głównie poprawki bezpieczeństwa, czy optymalizacje. Oznacza to, że kod napisany w wersji X.Y.9 powinien działać też w X.Y.1 i odwrotnie. Natomiast przy zmianie wersji zachowana jest tylko zgodność wsłecznia. Oznacza to, że kod napisany w wersji X.1 będzie działać w wersji X.9, ale odwrotnie już niekoniecznie. Kod z różnych gałęzi (zmiana liczby X) najprawdopodobniej nie zadziała poprawnie, choć czasem może się to udać. W chwili pisania tego skryptu najnowszą wersją Pythona jest 3.9.5. Pierwsza wersja Pythona 3.9 opublikowana została 05.10.2020, jej wsparcie skończy się w 2025 roku. Informacje o nowych i dostępnych wersjach znajdziemy na stronie: <https://www.python.org/downloads/>

Narzędzia online

Zanim zainstalujemy Pythona na lokalnym komputerze warto wspomnieć o tym, że małe programy możemy uruchomić w dostępnych online narzędziach takich jak:

https://www.onlinegdb.com/online_python_compiler

<https://repl.it>

Instalacja i konfiguracja środowiska

Zanim przystąpimy do pisania naszych pierwszych programów w języku Python musimy zainstalować oraz skonfigurować wszystkie niezbędne nam narzędzia. Poniższy opis będzie dotyczył konfiguracji środowiska programistycznego w systemie operacyjnym *Microsoft Windows*.

Uwaga do bardziej zaawansowanych użytkowników:

By poważnie pracować z Pythonem należy mieć go dostępnego na własnym komputerze. W codziennej praktyce programisty Pythona okazuje się, że czasem musimy mieć więcej niż jedną jego wersję zainstalowaną jednocześnie. Mogą w tym pomóc takie narzędzia jak pyenv

<https://github.com/pyenv/pyenv>

Instalacja interpretera języka Python

Windows

Najważniejszym narzędziem, które zainstalujemy jako pierwsze, jest interpreter języka Python. Jest to program, którego zadaniem jest wczytywanie, analizowanie oraz wykonywanie napisanego przez nas kodu. Dzięki niemu tworzone przez nas programy mogą zostać uruchomione.

Instalator dla najnowszej wersji interpretera Python dla systemu *Microsoft Windows* możemy pobrać z oficjalnej strony projektu - python.org/downloads. Wystarczy kliknąć w przycisk "Download Python 3.x.x" umieszczony na samej górze witryny. Po pobraniu instalatora należy go uruchomić.

W pierwszym kroku instalacji zaznaczamy opcję "Add Python 3.x to PATH" oraz wybieramy domyślny scenariusz instalacji klikając w przycisk "Install Now". Po kilku chwilach powinniśmy ujrzeć finalny ekran instalatora mówiący o poprawnym przeprowadzeniu całego procesu.

W celu sprawdzenia poprawności instalacji interpretera języka Python musimy uruchomić wiersz poleceń systemu Windows (tzw. konsolę). Robimy to poprzez wpisanie frazy cmd w wyszukiarkę programów i plików umieszoną w menu startowym na pasku zadań.

Po uruchomieniu konsoli wpisujemy komendę `python`. Jeżeli nasza instalacja przebiegła pomyślnie, w tym momencie powinno uruchomić się interaktywne środowisko języka Python, wypisując nam na konsolę wersję interpretera oraz znak zachęty `>>>`. W tym momencie możemy zamknąć wiersz poleceń.

Linux i MacOS

Na systemach z rodziny MacOS i Linux Python przeważnie jest już zainstalowany. Jednakże czasem może to być jeszcze wersja 2 Pythona. W zależności od systemu operacyjnego, czy dystrybucji Linuxa proces instalacji może być nieco inny, dlatego warto wtedy skorzystać z jakiegoś aktualnego poradnika. Dla MacOS dostępne są wersje instalacyjne na stronie Pythona:

<https://www.python.org/downloads/mac-osx/>

Można go też instalować z poziomu wiersza poleceń. Np.:

<https://docs.python-guide.org/starting/install3/osx/>

Podobnie można zrobić to na systemach z rodziny Linux:

<https://docs.python-guide.org/starting/install3/linux/>

<https://phoenixnap.com/kb/how-to-install-python-3-ubuntu>

Ipython

Po zainstalowaniu Pythona dostępny jest

Po zainstalowaniu Pythona dobrym pomysłem jest zainstalowania IPythona. Jest to Dowiedzieliśmy się też o pewnych cechach charakteryzujących języki programowania i zobaczyliśmy, gdzie w tym bogatym świecie możemy umieścić Pythona.

Z kolei w artykule „Jak zainstalować i używać Pythona” poznaliśmy sposoby na to, w jaki sposób można napisać skrypt w Pythonie i go uruchomić. Wspomniałem tam o REPL – interaktywnym środowisku Pythona. Ten skrót wziął się od angielskich słów `read-eval-print loop`, czyli czytaj-wykonuj-drukuj zapętl. Narzędzie to jest takim prostym, interaktywnym (czyli reagującym na nasze działania) środowiskiem programistycznym. Jest szczególnie przydatne i wygodne wtedy, kiedy chcemy poeksperymentować z krótkimi fragmentami kodu. Większość języków interpretowanych ma takiego własnego REPLa. Pojawiają się też dodatki, które go wzbogacają o nowe funkcjonalności i poprawiają komfort korzystania z takich narzędzi. My też skorzystamy z takiego narzędzia. W świecie Pythona dużą i zasłużoną sławę zyskał sobie IPython. Dziś zainstalujemy go i zaczniemy się z nim oswajać. By to zrobić, będziemy musieli skorzystać z jednego z narzędzi umożliwiających wydawanie

komputerowi poleceń tekstowych. Na Mac czy Linux może być to program terminal. W przypadku Windows może to być program Command Prompt albo Windows Power Shell. Te programy powinno dać się znaleźć w zainstalowanych aplikacjach. Uruchom więc teraz jeden z nich i wpisz następującą komendę:

```
pip install ipython
```

Jeśli na twoim systemie Python jest poprawnie zainstalowany, to w oknie terminala powinno pojawić się coś takiego jak w poniżej. Trzy kropki oznaczają, że jakiś fragment został pominięty:

```
1 $ pip install ipython
2 Collecting ipython
3   Downloading ...
4 ...
5   Installing collected packages: ipython
6     Successfully installed ipython-7.8.0
```

Pip to taki specjalny program, instalowany razem z Pythonem, który potrafi zainstalować różne zewnętrzne moduły pythonowe (opowiemy sobie o nim więcej przy innej okazji). Po takiej operacji IPython może zostać uruchomiony:

```
1 $ ipython
2 Python 3.7.2 (default, May  8 2019, 11:42:43)
3 Type 'copyright', 'credits' or 'license' for more information
4 IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help.
5
6 In [1]:
```

Jest trochę podobnie jak po uruchomieniu zwykłego REPL. Tutaj zamiast znaków zachęty w postaci >>> otrzymujemy ponumerowane wejścia (In) i jak się potem przekonamy, także wyjścia (Out). Dodatkowo mogą być one kolorowane.

IDE

Kolejnym narzędziem z którego będziemy korzystać podczas programowania w języku Python jest zintegrowane środowisko programistyczne (ang. Integrated Development Environment, IDE). Programy tego typu pełnią rolę edytora tekstu rozbudowanego o liczne funkcjonalności wspierające proces programowania (np. kolorowanie składni języka, integracja z narzędziami do kontroli wersji, etc.). To w nim będziemy pisać i rozwijać nasze programy.

Pycharm

Istnieje wiele IDE wspierających język Python. Jednym z najpopularniejszych jest PyCharm. Posiada ono dwie główne wersje - Professional, która jest płatna oraz Community w darmowej wersji.

Najnowszą wersję PyCharm możemy pobrać ze strony [jetbrains.com/pycharm/download](https://www.jetbrains.com/pycharm/download/). Do naszych celów w zupełności wystarczy prostsza oraz darmowa wersja *Community*, dlatego też wybieramy tę opcję. Po pobraniu instalatora należy go uruchomić.

Podczas instalacji wybieramy domyślne ustawienia na każdym etapie klikając przycisk "Next". Po zakończonej instalacji wybieramy opcję "*Run PyCharm Community Edition*" i klikamy przycisk "Finish".

Po uruchomieniu PyCharm zaznaczamy opcję "*Do not import settings*" i klikamy "OK". Na kolejnym okienku, gdzie możemy wybrać ustawienia skrótów klawiszowych oraz skonfigurować wygląd naszego IDE, także pozostajemy z domyślnymi wartościami klikając przycisk "OK".

Visual Studio Code

Stworzenie wirtualnego środowiska

Dobrą praktyką podczas rozwijania programów pisanych w języku Python jest tworzenie oddzielnych wirtualnych środowisk dla każdego projektu. Dzięki wirtualnym środowiskom możemy w prosty i organizowany sposób zarządzać zależnościami w różnych projektach. Dodatkowo ułatwiają one pracę z różnymi wersjami interpretera języka Python. Korzystanie z wirtualnych środowisk jest opcjonalne, jednak jest to zalecane podejście i warto z niego korzystać już od samego początku.

Wirtualne środowisko jest niczym więcej jak zdefiniowaną strukturą katalogów i plików w której będą przechowywane zainstalowane przez nas zależności oraz wybrana wersja interpretera Python. W celu stworzenia wirtualnego środowiska musimy uruchomić systemowy wiersz poleceń cmd.

W pierwszym kroku utworzymy katalog `workspace` w którym będziemy przechowywać wszystkie pliki związane z naszymi projektami. Wewnątrz tego katalogu stworzymy kolejny katalog - `envs` - w którym będą znajdować się wszystkie nasze wirtualne środowiska oraz katalog na projekty - `projects`.

```
1 mkdir workspace
2 cd workspace
3 mkdir envs
4 mkdir projects
```

Następnie, znajdująca się na poziomie katalogu `workspace`, stworzymy nasze pierwsze wirtualne środowisko o nazwie `bootcamp`. W tym celu wykorzystamy znajdujący się w bibliotece standardowej języka Python moduł `venv`:

```
1 python -m venv envs\bootcamp
```

Powyższe polecenie powinno utworzyć nowy katalog `bootcamp` w katalogu `envs`. Możemy to zweryfikować poprzez wykonanie polecenia `dir envs`, które wypisze nam na konsolę zawartość

wskazanego katalogu.

Stworzenie pierwszego projektu

Po zainstalowaniu interpretera języka Python, instalacji *PyCharm* oraz przygotowaniu wirtualnego środowiska, możemy stworzyć nasz pierwszy projekt.

W tym celu uruchamiamy *PyCharm* i z ekranu początkowego wybieramy opcję “Create New Project”. Jako lokalizacje naszego projektu wybieramy (poprzez kliknięcie w ikonkę trzech kropek po prawej stronie) katalog *workspace/projects*. Do wybranej ścieżki dopisujemy nazwę projektu - w naszym wypadku *\bootcamp*. Finalnie lokalizacją naszego projektu powinno być (z dokładnością do nazwy użytkownika):

```
1 C:\Users\kurs\workspace\projects\bootcamp
```

Następnie musimy wskazać intepreter języka Python z jakiego będziemy korzystać w ramach nowo tworzonego projektu. W tym celu wybierzemy intepreter znajdujący się w stworzonym przez nas wcześniej wirtualnym środowisku. Po kliknięciu ikonki koła zębatego po prawej stronie musimy wskazać następującą ścieżkę do pliku *python.exe*:

```
1 C:\Users\kurs\workspace\envs\bootcamp\Scripts\python.exe
```

Na zakończenie klikamy przycisk “Create”. W tym momencie stworzyliśmy nowy projekt w *PyCharm* z odpowiednio skonfigurowanym wirtualnym środowiskiem.

Podstawy języka Python

Każdy język programowania posiada ściśle zdefiniowaną strukturę oraz zestaw instrukcji używanych podczas implementacji programów. Strukturę tę nazywamy *składnią języka* i dzięki jej znajomości możemy komunikować się z komputerem poprzez wybrany przez nas język programowania. Składnia języka jest jednym z głównych aspektów rozróżniających jeden język programowania od drugiego.

Poza składnią, każdy język programowania posiada swój zbiór podstawowych typów danych oraz funkcji i operatorów umożliwiających ich transformację. Typy te są podstawowymi blokami budowy każdego programu.

W niniejszym rozdziale poznamy składnię języka Python oraz jego podstawowe typy i struktury danych.

Uruchamianie programów

Zanim przejdziemy do poznawania podstawowych elementów składniowych języka Python musimy nauczyć się uruchamiania naszych programów.

Interaktywne środowisko

Najprostszym sposobem na wykonanie kodu napisanego w języku Python jest skorzystanie z interaktywnego środowiska interpretera - nazywanego także konsolą lub REPL (ang. *read-eval-print loop*). Rozwiązanie to jest idealne jeżeli chcemy eksperymentować lub jednokrotnie zaimplementować i uruchomić dany program, dzięki czemu podejście to bardzo dobrze sprawdza się podczas nauki języka.

W celu uruchomienia interaktywnego środowiska interpretera języka Python musimy skorzystać z wiersza poleceń systemu *Windows* - cmd. Na początku w uruchomionej konsoli musimy aktywować stworzone przez nas wcześniej wirtualne środowisko. W tym celu uruchamiamy skrypt `activate` znajdujący się w katalogu wirtualnego środowiska:

```
1 workspace\envs\bootcamp\scripts\activate
```

Jeżeli aktywacja środowiska przebiegła pomyślnie to na początku kolejnej linii w naszej konsoli powiniśmy zobaczyć jego nazwę w nawiasach okrągłych - (bootcamp).

W kolejnym kroku możemy uruchomić interaktywne środowiska Python'a poprzez komendę `python`. Od tego momentu kolejne wpisywane instrukcje będą uruchamiane przez interpreter języka Python, a nie wiersz poleceń systemu *Windows*. Jeżeli chcielibyśmy zamknąć konsolę Python'a i wrócić do konsoli systemowej, wystarczy wpisać komendę `exit()`.

Po uruchomieniu konsoli języka Python zostanie wypisana aktualna wersja używanego przez nas interpretera oraz znak zachęty `>>>` oznaczający gotowość na przyjęcie instrukcji.

Jako pierwszą instrukcję w uruchomionej przez nas konsoli Python'a wykonajmy operację dodawania dwóch liczb:

```
1 >>> 2 + 2
2 4
```

Wynik operacji dodawania został wypisany w kolejnej linii interaktywnego środowiska. Tym samym uruchomiliśmy nasz pierwszy "program" w języku Python.

IPython Dobrym rozszerzeniem dla interaktywnego środowiska jest IPython. By go zainstalować w terminalu (w linii poleceń) wpisujemy:

```
1 pip install ipython
```

Po zainstalowaniu, by uruchomić ten interpreter wpisujemy w terminalu:

```
1 ipython
```

Zyskujemy wiele ciekawych funkcjonalności. Podpowiadanie kodu, kolorowanie składni. Ułatwiony dostęp do pomocy, np. zamiast pisać `help(max)`, by dostać pomoc dla funkcji `max`, możemy użyć notacji:

```
1 max?
```

Podwójny znak zapytania dodatkowo wyświetli nam kod pythona, w którym funkcja jest definiowana. Nie zadziała to jednak dla wszystkich funkcji, ponieważ część z nich została stworzona w języku C. Oprócz tego dostępnych szereg "magicznych" poleceń, które dostarczają dodatkowych, niedostępnych w standardowym interpreterze funkcjonalności. Listę dostępnych poleceń magicznych dostaniemy poprzez użycie polecenia `%lsmagic`. Pomoc dla konkretnych poleceń dostaniemy z wykorzystaniem znaku zapytania. Tak jak w poniższym przykładzie

```
1 In [1]: %lsmagic
2 Out[1]:
3 Available line magics:
4 %alias %alias_magic %autoawait %autocall %autoindent %
   automagic %bookmark %cat %cd %clear %colors %conda %
   config %cp %cpaste %debug %dhist %dirs %doctest_mode %ed
   %edit %env %gui %hist %history %killbgscripts %dir %
   less %lf %lk %ll %load %load_ext %loadpy %logoff %logon
   %logstart %logstate %logstop %ls %lsmagic %lx %macro %
   magic %man %matplotlib %mdir %more %nv %notebook %page
   %paste %pastebin %pdb %pdef %pdoc %pfile %pinfo %pinfo2
   %pip %popd %pprint %precision %prun %psearch %psource %
   pushd %pwd %pycat %pylab %quickref %recall %rehashx %
   reload_ext %rep %rerun %reset %reset_selective %rm %rmdir %
   %run %save %sc %set_env %store %sx %system %tb %time %
   timeit %unalias %unload_ext %who %who_ls %whos %xdel %
   xmode
5
6 Available cell magics:
7 %%! %%HTML %%SVG %%bash %%capture %%debug %%file %%html %%
   javascript %%js %%latex %%markdown %%perl %%prun %%pypy
   %%python %%python2 %%python3 %%ruby %%script %%sh %%svg
   %%sx %%system %%time %%timeit %%writefile
8
9 Automagic is ON, % prefix IS NOT needed for line magics.
10
11 In [2]: %pwd?
12 Docstring:
13 Return the current working directory path.
```

```
14
15     Examples
16     -----
17     ::
18
19     In [9]: pwd
20     Out[9]: '/home/tsuser/sprint/ipython'
21     ...
```

Plik z kodem

Innym podejściem do wykonywania pythonowych programów jest uruchamianie interpretera wraz z podaną ścieżką do pliku zawierającego kod. Z tej metody będziemy korzystać za każdym razem gdy będziemy implementować bardziej rozbudowane programy wymagające napisania wielu linii kodu.

Przy pomocy *PyCharm* otwórzmy wcześniej utworzony przez nas projekt `bootcamp`. W strukturze projektu wyświetlonej w panelu po lewej stronie kliknijmy prawym klawiszem myszy na główny katalog projektu i wybierzmy opcję `New > Python File` i nazwijmy nowy plik `hello.py`. Rozszerzenie `*.py` jest standardowym rozszerzeniem plików zawierających kod napisany w języku Python.

Wewnątrz pliku `hello.py` umieśćmy następującą instrukcję:

```
1 print("Hello World!")
```

Powyższy program używa wbudowanej funkcji `print()` do wypisania napisu `Hello World!` na konsole. W celu jego uruchomienia musimy użyć interpretera wraz z podaną ścieżką do pliku z kodem. Komendę tę musimy wykonać z aktywnym środowiskiem wirtualnym oraz znajdującym się w katalogu `workspace\projects\bootcamp`:

```
1 python hello.py
```

Po uruchomieniu naszego programu zapisanego w pliku `hello.py` na konsolę powinien zostać wypisany komunikat `Hello World!`.

Wsparcie PyCharm

PyCharm umożliwia nam także wykonywanie programów z poziomu IDE. Wystarczy kliknąć prawym klawiszem myszy na nazwę pliku oraz wybrać opcję `Run 'hello'`. Nasz prosty program zostanie uruchomiony, a jego rezultat wyświetlony w dolnym panelu. Także z poziomu *PyCharm* możemy uruchomić interaktywne środowisko języka Python w celu przeprowadzania prostych eksperymentów - z górnego menu wybieramy `Tools > Python Console...` i po chwili w dolnym panelu powinniśmy zobaczyć znany już nam znak zachęty `>>>` po którym możemy wprowadzać instrukcje.

Alternatywy Coraz popularniejsza alternatywa dla Pycharma jest np. VisualStudioCode.

<https://code.visualstudio.com/>

Jest to darmowe, wieloplatformowe środowisko, wspierające różne języki programowania.

Korzystanie z pomocy wbudowanej

Często pisząc programy zastanawiamy się dlaczego to nie działa? Jak tego używać? W takich sytuacjach dobrze jest umieć skorzystać z pomocy, jaką oferuje nam sam Python w postaci dokumentacji swoich funkcji. Dokumentacja to tekst, przygotowany przez twórcę programu, który informuje innych programistów o tym, w jaki sposób mogą skorzystać z jego programu. Do korzystanie z tych podpowiedzi w Pythonie służy specjalna funkcja `help`, do której przekazujemy jako argument nazwę funkcji, której dokumentację chcemy przeczytać:

```
1 In [1]: help(print)
2
3 Help on built-in function print in module builtins:
4
5 print(...)
6     print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
7
8     Prints the values to a stream, or to sys.stdout by default.
9     Optional keyword arguments:
10    file: a file-like object (stream); defaults to the current sys.
11        stdout.
12    sep:   string inserted between values, default a space.
13    end:   string appended after the last value, default a newline.
14    flush: whether to forcibly flush the stream.
```

W IPython jest to jeszcze prostsze. Wystarczy, że po nazwie funkcji wpisany zostanie znak ?:

```
1 In [2]: print?
2 Docstring:
3 print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
4
5 Prints the values to a stream, or to sys.stdout by default.
6 Optional keyword arguments:
7 file: a file-like object (stream); defaults to the current sys.stdout.
8 sep:   string inserted between values, default a space.
9 end:   string appended after the last value, default a newline.
10 flush: whether to forcibly flush the stream.
11 Type:    builtin_function_or_method
```

Błąd i wyjątki

Błędy i wyjątki rzucane w Pythonie to kolejna porcja informacji o tym, co zrobiliśmy źle. Są bardzo pomocne. W lepszym ich wykorzystaniu przydaje się też znajomość angielskiego, ale zawsze też można wesprzeć się tłumaczem. Namy wiele sytuacji, które mogą powodować błędy. Stąd też jest wiele rodzajów wyjątków i ich nazw. W miarę poznawania Pythona oswajamy się też z tymi wyjątkami i uczymy się z nich korzystać do szybszego zrozumienia

Szukanie pomocy O właściwie – zadawanie pytań i poszukiwanie odpowiedzi na nie to jedna z najczęstszych czynności programisty – szczególnie tego początkującego. Jeśli nie znasz angielskiego na odpowiednim poziomie, to nic straconego. Na początku możesz użyć ogólnodostępnych narzędzi do tłumaczeń. Z czasem na pewno nabierzesz biegłości. Gdy masz więc pytania i wątpliwości, to śmiało szukaj na nie odpowiedzi. Po pierwsze, znaleźć je możesz w oficjalnej dokumentacji Pythona (<https://docs.python.org/3/>). Choć trzeba przyznać, że to często dość wymagające źródło. Dużo pytań i wyjaśnień znajdziesz na platformie <https://stackoverflow.com/> – to jedno z najważniejszych miejsc w sieci, w którym programiści rozmawiają o swoich problemach. Jest też trochę miejsc, w których pomoc możesz znaleźć w języku polskim. Np. na Facebooku dostępne są grupy dla początkujących i zaawansowanych Pythonistów. W Internecie jest też forum <https://pl.python.org/forum/>. Wielu doświadczonych Pythonistów znaleźć można także na kanale IRC. Ten ostatni to taka stara usługa, rodzaj czatu, bardzo popularnego w latach 90. ubiegłego wieku. Zanim jednak zaczniesz bombardować starszych kolegów pytaniami, postaraj się znaleźć odpowiedź na własną rękę - choćby korzystając z google. Dopiero kiedy spróbujesz kilka razy i nie będziesz sobie radzić, poproś o pomoc kogoś bardziej doświadczonego. To takie dobre zasady. Ale dość tych porad. Wracajmy do Pythona!

Wartość, zmienna, referencja, obiekt

Jednym z pierwszych programów, jakie piszemy w jakimkolwiek języku, jest wypisanie powitania. W Pythonie może to wyglądać tak jak poniżej:

```
1 In [1]: x = "Programista"
2 In [2]: x
3 Out[2]: 'Programista'
4 In [3]: print(x)
5 Programista
6 In [4]: y = print(x)
7 Programista
8 In [5]: y
9 In [6]:
10 In [7]: print(y)
11 None
```

Co tu się dzieje. W In [1] do nazwy `x` przypisujemy wartość "Programista". Od teraz będziemy ten `x` w zależności od kontekstu nazywać zmienną (ang. *variable*), nazwą (ang. *name*) albo referencją (ang. *reference*). Referencja to taka wartość, która zawiera informację o położeniu innej wartości. U nas ta referencja ma nazwę `x` i wskazuje na napis "Programista". W In [2] po prostu odwołujemy się do zmiennej `x`. Jak widzimy tym, razem pojawia się sekcja Out[2]. To oznacza, że poprzednie działanie zwraca jakąś wartość. W tym przypadku jest to po prostu napis.

W In [2] wywołujemy funkcję `print`. Zauważmy, że tym razem nie ma sekcji Out. Oznacza to, że funkcja `print` niczego nie zwraca, choć, jak widzimy poniżej, pojawił się napis. By być precyzyjnym, powinieneś jednak napisać, że funkcja `print` zwraca specjalny obiekt, który nazywa się `None`. To słowo oznacza po angielsku tyle co: żaden, nikt, nic. Możemy więc napisać, że takie funkcje zwracają nic. A czasem – dość przewrotnie – po polsku mówimy, że one niczego nie zwracają. W Pythonie takie „nic” opisuje się właśnie słowkiem `None`. Kryje się za nim specjalny obiekt, osobny typ, który bywa przydatny. Z czasem zobaczymy, w jakich sytuacjach. Zastanów się nad tym chwilę i poeksperymentuj. Wróćmy jeszcze do naszego kodu. Samo przypisanie wartości do nazwy odbywa się przy użyciu znaku `=`. Co to jest jednak to przypisanie? Wyobraź sobie szafkę, która ma wiele szufladek. Obok niej biurko, a na nim notatnik. Do szuflad będziemy chować różne rzeczy, a w notatniku na biurku będziemy notować, co gdzie włożyliśmy. W jedną z szuflad wsadzamy kartkę z napisem "Programista". Szuflady są ponumerowane. Teraz w notatniku notujemy, że pod nazwą `x` od teraz będzie kryła się zawartość tej szuflady, w której kryje się ta kartka z napisem. Python robi za nas coś podobnego. Możemy w nim nawet sprawdzić numer tej szuflady, w taki sposób jak w poniżej:

```
1 In [4]: id(x)
2 Out[4]: 140087084787512
```

Jak widać, została tu zwrócona jakaś duża liczba. Możemy ją potraktować właśnie jako numer naszej szuflady – czyli adres miejsca w pamięci. W szufladzie, jak pamiętasz, znajduje się kartka z napisem. W Pythonie moglibyśmy powiedzieć, że ta kartka jest obiektem (ang. *object*), a wartością jest sam napis. I teraz za każdym razem, kiedy gdzieś w programie użyjemy nazwy `x`, to Python otwiera odpowiednią szufladę i przekazuje w to miejsce tę kartkę. Oczywiście to duże uproszczenie. W twojej głowie mogą teraz powstać pytania o to, co się dalej dzieje z tą kartką, a co jest w szufladzie, a co, jeśli wiele razy obok siebie użyjemy tego `x`? Czy mogę do szuflady włożyć wiele kartek naraz? Świeście, jeśli tak jest. O takie pytania nam chodzi. Będziemy małymi krokami szukać na nie odpowiedzi. Wróćmy do tej nowej funkcji. Zobaczmy, co znajdziemy w jej dokumentacji:

```
1 In [1]: id?
2
3 Signature: id(obj, /)
4
5 Docstring:
6
7 Return the identity of an object.
```

```
8 This is guaranteed to be unique among simultaneously existing objects.  
9  
10 (CPython uses the object's memory address.)  
11  
12 Type: builtin_function_or_method
```

Pojawia się tu pojęcie `object`. Termin ten bywa różnie definiowane w różnych językach programowania. Na nasze potrzeby możemy przyjąć, że w Pythonie obiektem jest wszystko to, co można przypisać do zmiennej. Jak się przekonamy później, obiekty mogą mieć atrybuty i metody. Metody to różnego rodzaju działania. Atrybuty to z kolei różne cechy obiektu. Obiektem może być np. pies. Jego atrybutem może być waga, a metodą np. polecenie: `daj_glos`. Więcej opowiemy sobie o tym w artykule poświęconym programowaniu obiektowemu, już teraz jednak przyda się nam pewna elementarna wiedza o tych zagadnieniach, tym bardziej że w Pythonie w zasadzie wszystko jest obiektem. W naszym wcześniejszym przykładzie to kartka jest obiektem. Z kolei napis na niej to wartość tego obiektu. Jak widzieliśmy, nasz obiekt powstawał w momencie przypisania do zmiennej, ale, co ciekawe, może on istnieć także bez tej nazwy, bez przypisania do zmiennej – bez referencji do niego. Możemy stworzyć kartkę z napisem i nie odnotować jej w naszym notatniku. Referencja nie powstała, ale kartka ma się dobrze. Takich kartek jednak nie chcemy przechowywać zbyt długo. Gdyby było ich bardzo dużo mogłyby utrudnić nam pracę. Warto więc co jakiś czas je posprzątać. Jak by to wyglądało w Pythonie? Podajmy w funkcji `id` wartość i zobaczymy, że także dla niej zostanie zwrotny adres obiektu

```
1 In [5]: id("Programista")  
2  
3 Out[5]: 140247295987904
```

Zauważ, że tym razem jest to już nieco inną liczbą, mimo że napis jest ten sam. Nie zawsze jednak ten adres będzie się różnił. Jako ciekawostkę potraktuj fakt, że jeśli nieznacznie zmodyfikujemy nasz napis, to otrzymamy nieco inne zachowanie. Zobacz:

```
1 In [1]: x1 = "Programista_backend"  
2  
3 In [2]: y1 = "Programista_backend"  
4  
5 In [3]: x1 is y1  
6 Out[3]: True  
7  
8 In [4]: x2 = "Programista backend"  
9  
10 In [5]: y2 = "Programista backend"  
11  
12 In [6]: x2 is y2  
13 Out[6]: False
```

Tym razem *identity* naszych obiektów jest takie same w przypadku `x1` i `y1`. Python w celach op-

tymalizacyjnych w niektórych przypadkach odkłada takie obiekty w pamięci podręcznej – jest to tzw. *cache*. Pozwala to oszczędzać pamięć. Zamiast przechowywać wiele identycznych kartek w różnych szufladach – trzymamy jedną i tam, gdzie trzeba, przekazujemy tylko jej adres. Dzięki temu pozostałe są wolne. W przypadku napisów stanie się tak wtedy, gdy napis składa się z samych znaków alfanumerycznych (litery i cyfry) i znaków podkreślenia. Nie jest to jednak mechanizm, na którym powinniśmy polegać, ponieważ te metody optymalizacji mogą się zmieniać wraz z nowymi wydaniami Pythona. Potraktuj to więc jako ciekawostkę. Wiąże się z tym jednak pewne zagadnienie, które może być dla nas ważne w przyszłości.

Kiedy w Pythonie chcemy porównać dwie wartości, używamy operatora porównania `==`. O samych operatorach powiemy sobie jeszcze trochę później. Na razie wystarczy, że zapamiętasz, że gdy chcesz porównać ze sobą jakieś dwie wartości, sprawdzić, czy są to np. takie same liczby, czy napisy, to używasz właśnie tego podwójnego znaku `==`. Działa to tak jak poniżej:

```
1 In [7]: x == y
2
3 Out[7]: True
```

Słowo `True` poznaliśmy już w poprzednim artykule. Oznacza ono prawdę. Jego przeciwieństwem jest słówko `False`. W tym przypadku nasze porównanie powiedziało, że to, co jest po lewej i prawej stronie operatora `==`, jest takie samo. Tak naprawdę chodzi tu o to, że pewne wartości są tu takie same. Jeśli naszym obiektem są liczby, to porównywać będziemy ze sobą wartości tych liczb. Jeśli napisy, to sprawdzimy wtedy, czy te napisy są identyczne. Ale ten operator nie odpowiada na pytanie, czy są to te same obiekty. Operator `==` nie sprawdza więc tego, czy odwołujemy się do tej samej szuflady – a wiemy już, że czasem wartości mogą leżeć w różnych szufladach naszej pamięci, mimo że wydają się identyczne. Kiedy chcemy sprawdzić, czy nasze zmienne (referencje) wskazują na te same obiekty, używamy innego operatora – jest nim słowo `is`.

```
1 In [8]: x is y
2
3 Out[8]: True
```

Zwrócona wartość `True` oznacza, że `x` i `y` wskazują na ten sam obiekt. Czytając dosłownie, moglibyśmy powiedzieć, że `x` jest `y`-kiem. Ale chodzi tu o to, że to, na co wskazuje `x`, jest tym samym, na co wskazuje `y`. Może się zdarzyć też inna sytuacja. Wartości będą takie same, ale `x` i `y` będą wskazywać na dwa różne obiekty – czyli na dwie różne szuflady.

W Pythonie operator `is` używany jest bardzo często wtedy, kiedy chcemy porównać coś z wartościami logicznymi `True/False` oraz z `None`. Robimy tak dlatego, że te wartości logiczne są zawsze tymi samymi obiektami. To znaczy, że każdy z nich ma swoją własną szufladę i nie są tworzone nowe szufladki, które by je zawierały. Takie obiekty nazywamy też czasem singletonami. W zapisie chodzi o to, by zamiast pisać:

jakas_wartosc == True pisać raczej tak: jakas_wartosc is True. Czasem może się to okazać istotne. Prześledź uważnie poniższy przykład. Widać na nim różne wyniki zastosowania operatorów == oraz is

```
1 In [1]: x = True
2 In [2]: x is True
3 Out[2]: True
4
5 In [3]: x == True
6 Out[3]: True
7
8 In [4]: x = 1
9 In [5]: x is True
10 Out[5]: False
11
12 In [6]: x == True
13 Out[6]: True
14
15 In [7]: x = 2
16 In [8]: x is True
17 Out[8]: False
18
19 In [9]: x == True
20 Out[9]: False
```

Do wartości logicznych wróćmy jeszcze później, kiedy będziemy omawiać typy. W tej chwili postaram się jednak zapamiętać te podstawowe fakty o wartościach, zmiennych, nazwach, obiektach i referencjach. Poeksperymentuj z funkcją id dla różnych napisów. Ale też i dla liczb czy wartości logicznych.

help - system pomocy

Poznaliśmy już funkcję help i zobaczyliśmy, jak ją wywołać dla jakiejś innej funkcji. Skoro help to też funkcja, to zobacz, co się stanie, jeśli spróbujesz wywołać pomoc dla niej samej? Uwaga: czasem ta dokumentacja jest długa. By przejść dalej wystarczy, wcisnąć klawisz Enter. Z pomocy można wyjść, wciskając klawisz q. Spróbuj poniższych wywołań:

```
1 In [1]: help(help)
2
3 ...
4 In [2]: help()
5
6 ...
7
8 In [3]: ?
9
10 ...
```

Python, podobnie jak inne języki programowania, składa się z pewnych słów kluczowych. Kiedy tworzymy jakieś nowe zmienne, musimy pamiętać o tym, że nie mogą się one nazywać tak jak te słowa kluczowe. Warto więc wiedzieć, jakie są to słowa. Możemy to sprawdzić w pomocy:

```
1 In [5]: help()
2
3 Welcome to Python 3.6's help utility!
4
5 ...
6
7 help> keywords
8
9 Here is a list of the Python keywords. Enter any keyword to get more
   help.
10
11 False      def       if        raise
12
13 None       del       import    return
14
15 True       elif      in       try
16
17 and        else      is        while
18
19 as         except    lambda   with
20
21 assert     finally  nonlocal  yield
22
23 break      for      not
24
25 class      from     or
26
27 continue   global   pass
```

Jeśli spróbujesz stworzyć zmienną z któregoś z tych słów, to zostanie zwrócony błąd:

```
1 In [6]: class = 1
2
3 File "<ipython-input-6-0251f39d4826>", line 1
4
5     class = 1
6         ^
7
8 SyntaxError: invalid syntax
```

Podobnie zakończy się próba wywołania dla nich pomocy: `help(class)` - bezpieczniej będzie użyć tu tego słowa w postaci napisu: `help("class")`. Można też dotrzeć do opisu słów kluczowych właśnie poprzez tę rozbudowaną pomoc dostępną po wywołaniu funkcji `help()`. Postaraj się zrobić to samodzielnie.

Zauważ, że w Listingu 13 nie ma na liście funkcji print, help czy id. Te ostatnie są tak zwany funkcjami wbudowanymi. Pełną ich listę (dla Pythona 3.7) znajdziesz w Tabeli 1.

Built-in Functions				
<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	

Figure 0.3: img

Tabela 1. Funkcje wbudowane w Pythona. Możesz też o nich przeczytać w dokumentacji Pythona: <https://docs.python.org/3/library/functions.html>

W przypadku funkcji wbudowanych możliwe jest użycie ich nazw jako nazwy zmiennej. Należy jednak pamiętać o tym, że jeśli to zrobimy, to stracimy możliwość użycia takiej funkcji. Mówimy wtedy, że przesłaniamy daną funkcję czy obiekty. W Pythonie funkcje to też obiekty. A więc ich nazwy to referencje do tych obiektów. Zobaczmy poniżej przykład przestonienia wbudowanej funkcji print poprzez zastosowanie tej nazwy dla zmiennej

```
1 In [1]: print("Dziaa")  
2  
3 Dziaa  
4  
5 In [2]: print = 10  
6  
7 In [3]: print("Dziaa?")  
8  
9 TypeError Traceback (most recent call last)
```

```
10 <ipython-input-3-24d45c998241> in <module>
11
12 ----> 1 print("łdziaa?")
13
14
15 TypeError: 'int' object is not callable
16
17 In [4]: print
18
19 Out[4]: 10
```

W In [2] powiedzieliśmy Pythonowi, że od teraz nazwa `print` wskazuje na obiekt, który jest liczbą 10. Zmieniliśmy po prostu szufladki, na które wskazuje `print`. Możemy też zrobić odwrotnie, co może być przyczyną różnych kłopotów. Prześledź to na Listingu 16.

```
1 In [1]: x=print
2
3 In [2]: x("to łdziaa")
4
5 to łdziaa
6
7
8
9 In [3]: y=10
10
11 In [4]: print=y
12
13 In [5]: print('czy to zadziala?')
14
15 TypeError Traceback (most recent call last)
16
17 <ipython-input-5-297252317ec0> in <module>
18
19 ----> 1 print('czy to zadziala?')
20
21 TypeError: 'int' object is not callable
22
23
24 In [6]: print = __builtins__.print
25
26 In [7]: print("Uff znówu łdziaa")
27
28 Uff znówu łdziaa
```

Na koniec zapamiętaj jeszcze, że nazwy zmiennych powinny składać się z liter, cyfr i znaku podkreślenia. Nie mogą natomiast zaczynać się od cyfry. Zobacz, co się stanie, gdy spróbujesz obejść tę zasadę. Gdy skończysz możesz zamknąć IPythona. By to zrobić, wpisz po prostu `exit()` lub naciśnij jednocześnie `Ctrl+d`. (W przypadku Windows `Ctrl + z`, a potem `enter` lub `F6` i potem `Enter`.)

Podstawowe typy danych

Prawie każdy program komputerowy ma za zadanie przechowywanie oraz przetwarzanie dostarczonych mu danych. Źródło danych może być różne tak samo jak różna może być ich struktura. Jednak w każdym z tych przypadków dane takie muszą być finalnie reprezentowane jako zbiór danych typów podstawowych.

W języku Python podstawowymi typami danych są:

- liczby całkowite - `int`
- liczby zmiennoprzecinkowe - `float`
- napisy - `str`
- wartości logiczne - `bool`

Liczby

Liczby tworzone są w Python'ie poprzez użycie literałów - w tym wypadku znaków specjalnych i cyfr.

Poniżej znajdują się przykłady tworzenia liczb całkowitych:

```
1 123
2 0
3 -1000
```

Analogicznie wygląda tworzenie liczb zmiennoprzecinkowych - w tym wypadku możemy jednak określić także część ułamkową tworzonej przez nas liczby korzystając ze znaku kropki:

```
1 1.0
2 123.456
3 -99.99
4 10.
5 .1
```

Zarówno liczby całkowite jak i zmiennoprzecinkowe możemy tworzyć także przy użyciu konstruktorów - `int()` oraz `float()`. Instrukcje te wygenerują nam odpowiednio liczbę 0 oraz 0.0.

Oczywiście samo tworzenie liczb nie miałoby sensu, jeżeli nie moglibyśmy przeprowadzać na nich operacji arytmetycznych. Do podstawowych operatorów wbudowanych w składnię języka Python należą:

- operator dodawania - `+`
- operator odejmowania - `-`
- operator mnożenia - `*`
- operator dzielenia - `/`

- operator dzielenia całkowitoliczbowego - //
- operator modulo (reszta z dzielenia) - %
- operator potęgi - **

Zastosowanie operatorów arytmetycznych dla liczb całkowitych zostało przedstawione w poniższej sesji interaktywnego środowiska:

```
1 >>> 2 + 2
2 4
3 >>> 100 - 50
4 50
5 >>> 9 * 5
6 45
7 >>> 10 / 3
8 3.333333333333335
9 >>> 10 // 3
10 3
11 >>> 10 % 3
12 1
13 >>> 2 ** 10
14 1024
```

Jak widać po wynikach wykonanych operacji, jedynie operator dzielenie jako wynik zwrócił liczbę zmiennoprzecinkową. W przypadku operacji na liczbach zmiennoprzecinkowych wynikiem zawsze będzie liczba zmiennoprzecinkowa:

```
1 >>> 2.0 + 2.0
2 4.0
3 >>> 10.0 // 3.0
4 3.0
```

Python umożliwia także mieszanie obydwu typów liczbowych w ramach jednej operacji - w takim wypadku wynikiem będzie także liczba typu **float**.

```
1 >>> 2 + 2.0
2 4.0
```

W przypadku obliczeń składających się z większej liczby składników możemy użyć nawiasów okrągłych () w celu priorytetyzacji działań:

```
1 >>> (2 * (5 + 5)) ** 2
2 400
```

Napisy

Napisy służą do przechowywania ciągu znaków i reprezentowane są przez typ `str`. W celu stworzenia napisu musimy zadany ciąg znaków otoczyć pojedynczym ('') lub podwójnym ("") cudzysłowem z obu stron. Poniżej przedstawiono przykłady kilku poprawnych składniowo napisów:

```
1 'abc'
2 "abc"
3 'napis z " w środku'
4 "napis z ' w środku"
5 ''
6 ""
```

Jak widać z powyższych przykładów dzięki możliwości zamiennej korzystania z pojedynczego lub podwójnego znaku cudzysłowu możemy umieszczać znaki cudzysłowu także wewnętrz napisu.

W celu utworzenia napisów wielolinijkowych składnia języka Python korzysta z potrójnego powtórzenia cudzysłowu - """ lub """. Przykład:

```
1 """
2 pierwsza linia
3 druga linia
4 """
```

Analogicznie do typów odpowiedzialnych za liczby, także napisy mogą być tworzone przy użyciu konstruktora. W przypadku napisów operacja `str()` utworzy pusty napis - ''. Konstruktor `str()` możemy przyjmować także jako parametr wartości innych typów. W przypadku liczb zadana wartość zostanie zamieniona na napis:

```
1 >>> str(123)
2 '123'
```

Operatorem używanym bardzo często w przypadku obiektów typu `str` jest operator dodawania + - jest on odpowiedzialny za połączenia (konkatenację) dwóch napisów ze sobą. W celu wypisania na konsoli dwóch połączonych ze sobą napisów możemy użyć znanej nam już funkcji `print` w następujący sposób:

```
1 print('Hello' + ' World!')
```

Po uruchomieniu powyższego programu na konsoli zobaczymy połączony napis Hello World!.

Formatowanie napisów Termin formatowanie napisów oznacza tworzenie nowych napisów na podstawie innych wartości. Najprostszym sposobem formatowania napisów w języku Python jest użycie operatora dodawania w celu konkatenacji dwóch napisów ze sobą. Rozwiążanie to jest jednak

dość niewygodne w przypadku chęci połączenia większej liczby napisów oraz wartości innych typów w napis wyjściowy.

W celu ułatwienia tego zadania do składni języka Python został wprowadzony specjalny rodzaj napisów rozpoczynający się od litery *f*, nazywany też *f-strings*. W tego typu napisach możliwe jest umieszczenie nawiasów klamrowych {} pomiędzy którymi możemy umieścić inną instrukcję języka Python, której rezultat zostanie wstawiona w miejsce nawiasów {} w wynikowym napisie.

```
1 print(f'Wynik dodawania to {2 + 2}!')
```

Po uruchomieniu powyższego przykładu na konsolę zostanie wypisany komunikat Wynik dodawania to 4!.

Napisy typu *f-strings* wykorzystywane są często wraz ze zmiennymi, o których dowiemy się w jednym z kolejnych rozdziałów.

Wartości logiczne

Wartości logiczne reprezentowane są w języku Python poprzez typ *bool*. Mogą przyjmować one jedynie dwie wartości - prawda lub falsz. Do ich zapisu używa się literałów *True* i *False*.

Wartości logiczne posiadają także swoje operatory logiczne:

- operator koniunkcji - *and*
- operator alternatywy - *or*
- operator negacji - *not*

Poniższa sesja interaktywnego środowiska obrazuje użycie każdego z nich:

```
1 >>> True and False
2 False
3 >>> True and True
4 True
5 >>> True or False
6 True
7 >>> False or False
8 False
9 >>> not True
10 False
11 >>> not False
12 True
```

W przypadku złożonych wyrażeń możemy użyć nawiasów okrągłych w celu priorytetyzacji działań:

```
1 >>> (True or False) and not True
2 False
```

Wartości logiczne tworzone są także jako wynik użycia operatorów porównania na innych typach wbudowanych. Podstawowymi operatorami porównania w języku Python są:

- operator mniejsze od - <
- operator mniejsze od lub równe - <=
- operator większe od - >
- operator większe od lub równe - >=
- operator równości - ==
- operator różny od - !=

Poniższa sesja interaktywnego środowiska pokazuje wykorzystanie operatorów porównania na przykładzie liczb całkowitych:

```
1 >>> 1 < 2
2 True
3 >>> 2 < 1
4 False
5 >>> 1 <= 1
6 True
7 >>> 2 > 1
8 True
9 >>> 2 >= 1
10 True
11 >>> 1 == 1
12 True
13 >>> 1 != 2
14 True
```

Wartości logiczne mogą być tworzone także przy wykorzystaniu konstruktora `bool()`, który domyślnie zwraca wartość `False`. Konstruktor ten jako parametr może przyjmować także obiekty innych typów podstawowych - w tym liczby i napisy. W przypadku liczb 0 oraz 0.0 spowoduje utworzenie wartości `False`, każda inna liczba da w wyniku `True`. W przypadku obiektów typu `str` jedynie pusty napis '' napis spowoduje utworzenie wartości `False`. Jeżeli do konstruktora `bool()` przekażemy bardziej złożoną strukturę danych (tuple, liste, słownik, zbiór - kolekcje te omówimy w jednym z kolejnych rozdziałów) -wróci on `True` jeżeli dana kolekcja zawiera conajmniej jeden element.

Poniżej przedstawiono przykłady użycia konstruktora `bool()`:

```
1 >>> bool()
2 False
3 >>> bool(0)
4 False
5 >>> bool(100)
6 True
7 >>> bool('')
8 False
9 >>> bool('abc')
```

```
10 True
11 >>> bool([])
12 False
13 >>> bool([1, 2, 3])
14 True
```

Zmienne

Znając podstawowe typy danych języka Python - tzn. liczby, napisy, wartości logiczne - możemy przejść do poznania kolejnego ważnego aspektu nie tylko Python'a ale także każdego innego języka programowania - zmiennych.

Zmiennych używamy do przechowywania wartości pod zadaną nazwą. Pełnią one rolę pudełek wraz z etykietami umożliwiających przechowywanie w środku danych.

Nazwa zmiennej może składać się z liter (małych i dużych), cyfr (nie jako pierwszy znak) oraz znaku podkreślenia `_`. Do przypisywania wartości zmiennej używamy znaku operatora przypisania `=`.

Przykładowe utworzenie zmiennej o nazwie `wiek` typu `int` o wartości 18 wygląda następująco:

```
1 wiek = 18
```

Tak utworzoną zmienną możemy użyć w dalszej części naszego programu tak samo jak używaliśmy bezpośrednio tworzonych wartości.

Dobrą praktyką podczas programowania w języku Python jest używanie nazw zmiennych składających się z małych liter. W celu zapewnienia czytelności przy zmiennych składających się z kilku wyrazów, do ich rozdzielenia należy użyć znaku podkreślenia `_`.

Poniższy przykład przedstawia wykorzystanie zmiennych do przeprowadzenia prostych obliczeń polegających na obliczeniu pola prostokąta o zadanych wymiarach:

```
1 szerokosc = 10
2 wysokosc = 20
3 pole = szerokosc * wysokosc
4 print(f'Pole prostokata: {pole}')
```

Po uruchomieniu naszego programu na konsolę zostanie wypisany następujący komunikat:

```
1 Pole prostokata: 200
```

Wartość do zmiennej można przypisywać wiele razy oraz wartości te mogą być także różnego typu:

```
1 x = 10
2 x = x * 2
3 x = 'abc'
```

Często używanym określeniem na wartości w kontekście danego typu danych jest *obiekt*. Możemy powiedzieć zatem że wartość 1 jest obiektem typu `int`, a napis 'abc' jest obiektem typu `str`, etc.

Rozszerzone operatory przypisania

Operacją wykonywaną często wraz z przypisaniem wartości do zmiennej jest przeprowadzenie dodatkowych obliczeń na tej samej zmiennej do której chcemy przypisać jej nową wartość.

Przykładem może być zwiększenie zmiennej o zadaną wartość:

```
1 x = 10  
2 x = x + 5
```

Powyższy przykład może zostać zapisany w skróconej formie z wykorzystaniem rozszerzonego operatora przypisania `+=`:

```
1 x = 10  
2 x += 5
```

W obu przypadkach wartość zmiennej `x` jest finalnie równa 15. Każdy z operatorów arytmetycznych posiada swoją rozszerzoną wersję.

Struktury danych

Typy danych takie jak liczby, napisy czy wartości logiczne są podstawowymi elementami używanymi przy budowaniu programów. W celu przechowywania ich większej ilości w uporządkowany sposób potrzebujemy jednak bardziej zaawansowanych struktur, umożliwiających nam przetrzymywanie i operację na wielu elementach jednocześnie.

Python posiada kilka wbudowanych w język struktur danych, są to:

- tuple - tuple
- listy - list
- słowniki - dict
- zbiory - set

Tuple

Tupla (zwana też krotką) jest niemutowalną kolekcją służącą do przechowywania wielu elementów.

Najprostszym sposobem utworzenia tupli jest użycie nawiasów okrągłych () pomiędzy którymi umieszcza się elementy kolekcji rozdzielone znakiem przecinka ,. Poniżej przykład utworzenia tupli zawierającej 3 liczby całkowite:

```
1 (1, 2, 3)
```

Tupla, tak samo jak każda inna kolekcja w Python'ie, może przechowywać jednocześnie elementy różnych typu:

```
1 ('abc', 1, 99.99)
```

Tupla zawierająca jeden element musi kończyć się znakiem przecinka:

```
1 ('abc',)
```

Stworzenie pustej tupli możliwe jest jedynie przy pomocy konstruktora `tuple()`.

Operator dostępu Przechowywania wielu elementów bez możliwości pobrania pojedynczej wartości z kolekcji nie miałoby sensu. W Python'ie do tego celu służy operator dostępu do elementu - `[]`. Pomiędzy nawiasami kwadratowymi umieszczamy liczbę całkowitą oznaczającą indeks elementu, który chcemy pobrać. Elementy numerowane są od 0 (pierwszy element) aż do wartości mniejszej o jeden od liczby elementów w całej kolekcji (ostatni element).

Poniższa sesja interaktywnego środowiska pokazuje użycie operatora `[]`:

```
1 >>> uczestnicy = ('Jan', 'Piotr', 'Ola', 'Tomasz', 'Sylwia')
2 >>> uczestnicy[0]
3 'Jan'
4 >>> uczestnicy[1]
5 'Piotr'
6 >>> uczestnicy[4]
7 'Sylwia'
```

Dostęp do elementu o nieistniejącym indeksie spowoduje wygenerowanie wyjątku `IndexError` - wyjątki będą omawiane w jednym z kolejnych rozdziałów.

Operator `[]` umożliwia także indeksowanie elementów od tyłu kolekcji używając do tego liczb ujemnych. W tym wypadku element o indeksie `-1` będzie ostatnim elementem, `-2` przedostatnim, itd. Przykładowe użycie ujemnych indeksów wygląda następująco:

```
1 >>> uczestnicy = ('Jan', 'Piotr', 'Ola', 'Tomasz', 'Sylwia')
2 >>> uczestnicy[-1]
3 'Sylwia'
4 >>> uczestnicy[-2]
5 'Tomasz'
6 >>> uczestnicy[-5]
7 'Jan'
```

Wycinanie Operator dostępu [] może służyć także do tworzenia "wycinków" kolekcji na podstawie oryginalnego obiektu. W celu stworzenia podkolekcji ograniczonej dwoma indeksami, musimy pomiędzy nawiasami kwadratowymi podać dwa indeksy rozdzielone znakiem dwukropka :. Pierwszy indeks oznacza początek nowej kolekcji, a drugi - indeks o jeden większy niż jej koniec. Brak któregokolwiek z indeksów oznacza odpowiednio utworzenie kolekcji od początku lub do samego końca oryginału. Indeksy o ujemnej wartości mają takie same znaczenia jak przy pobieraniu pojedynczego elementu - oznaczają one numerowanie od tyłu.

Przykłady wycinania zostały przedstawione w poniższej sesji interaktywnej:

```
1 >>> uczestnicy = ('Jan', 'Piotr', 'Ola', 'Tomasz', 'Sylwia')
2 >>> uczestnicy[1:3]
3 ('Piotr', 'Ola')
4 >>> uczestnicy[2:4]
5 ('Ola', 'Tomasz')
6 >>> uczestnicy[:3]
7 ('Jan', 'Piotr', 'Ola')
8 >>> uczestnicy[2:]
9 ('Ola', 'Tomasz', 'Sylwia')
10 >>> uczestnicy[::-1]
11 ('Jan', 'Piotr', 'Ola', 'Tomasz')
12 >>> uczestnicy[-2:]
13 ('Tomasz', 'Sylwia')
```

Podczas wycinania kolekcji możemy ustalić także "krok" wg. którego będą pobierane elementy w celu utworzenia nowego obiektu. Dzięki temu parametrowi możemy stworzyć kolekcję z np. co drugiego elementu oryginału. Ujemny krok oznacza rozpoczęcie tworzenia nowej kolekcji od końca. Parametr ten podaje się jako trzeci argument pomiędzy nawiasami kwadratowymi [], także oddzielony znakiem dwukropka ::.

Przykładowe zastosowanie wycinania wraz z podanym krokiem:

```
1 >>> uczestnicy = ('Jan', 'Piotr', 'Ola', 'Tomasz', 'Sylwia')
2 >>> uczestnicy[0:4:2]
3 ('Jan', 'Ola')
4 >>> uczestnicy[4:0:-1]
5 ('Sylwia', 'Tomasz', 'Ola', 'Piotr')
6 >>> uczestnicy[::-3]
7 ('Jan', 'Tomasz')
8 >>> uczestnicy[::-1]
9 ('Sylwia', 'Tomasz', 'Ola', 'Piotr', 'Jan')
```

Funkcja len() Bardzo przydatną funkcją przy pracy z kolekcjami jest `len()`. Dzięki niej możemy poznać liczbę elementów utworzonej kolekcji, którą musimy podać jako jedyny parametr funkcji. Przykładowe użycie `len()`:

```
1 >>> len(uczestnicy)
2 5
```

Operator zawierania W celu sprawdzenia obecności danego elementu w kolekcji należy skorzystać z operatora zawierania `in` lub jego zanegowanej wersji `not in`:

```
1 >>> 'Jan' in uczestnicy
2 True
3 >>> 'Krzysztof' not in uczestnicy
4 True
```

Listy

Lista jest mutowalną kolekcją. W odróżnieniu od niemutowalnej tupli, długość listy jak i jej zawartość mogą zostać zmienione po jej utworzeniu.

Podstawowym sposobem utworzenia listy jest użycie nawiasów kwadratowych `[]` pomiędzy którymi znajdują się elementy kolekcji rozdzielone znakiem przecinka `,`. W następującym przykładzie utworzona została 5-cio elementowa lista zawierająca liczby naturalne:

```
1 [10, 20, 50, 100, 200]
```

Pustą listę możemy utworzyć na dwa sposoby - poprzez nawiasy kwadratowe `[]` bez żadnych elementów kolekcji lub konstruktor `list()`.

Tak samo jak w przypadku tupli funkcja `len()` zwraca liczbę elementów znajdujących się w kolekcji:

```
1 >>> len([10, 20, 50, 100])
2 5
```

Analogicznie do typu `tuple` dla listy zachowuje się także operator dostępu do elementu `[]`, wycinanie oraz operator zawierania `in`:

```
1 >>> liczby = [10, 20, 50, 100]
2 >>> liczby[0]
3 10
4 >>> liczby[-1]
5 100
6 >>> liczby[1:3]
7 [20, 50]
8 >>> 100 in liczby
9 True
10 >>> 10 not in liczby
11 False
```

Tym czym lista różni się od tupli jest jej mutowalność - stan listy może być zmieniany po jej utworzeniu.

Do zmiany pojedynczego elementu listy używamy operatora przypisania:

```
1 >>> liczby = [10, 20, 50, 100]
2 >>> liczby[0] = 5
3 >>> liczby
4 [5, 20, 50, 100]
```

W celu dodania elementu na koniec kolekcji należy skorzystać z metody `append()` na obiekcie listy:

```
1 >>> liczby = [10, 20, 50, 100]
2 >>> liczby.append(200)
3 >>> liczby
4 [10, 20, 50, 100, 200]
```

Dodanie elementu pod wskazanym indeksem umożliwia metoda (tzn. funkcja wykonywana na rzecz obiektu - o metodach będziemy mówić więcej w rozdziale dotyczącym programowania obiektowego) `insert()` przyjmująca dwa parametry - indeks pod którym ma zostać umieszczony nowy element oraz wartość danego elementu. Przykład:

```
1 >>> liczby = [10, 20, 50, 100]
2 >>> liczby.insert(0, 5)
3 >>> liczby
4 [5, 10, 20, 50, 100]
```

Możliwe jest także wykorzystanie wycinania w celu zastąpienia zadanego zakresu elementami z innej kolekcji:

```
1 >>> liczby = [10, 20, 50, 100]
2 >>> liczby[1:3] = [1, 2]
3 >>> liczby
4 [10, 1, 2, 100]
```

Usunięcie elementu spod wskazanego indeksu realizowanie jest przy pomocy wyrażenia de l:

```
1 >>> liczby = [10, 20, 50, 100]
2 >>> del liczby[0]
3 >>> liczby
4 [20, 50, 100]
```

Zbiory

Zbiór jest mutowalną kolekcją przechowującą jedynie unikalne wartości. W języku Python zbiory tworzy się przy pomocy listy elementów pomiędzy nawiasami klamrowymi {}. Poniżej przykładowy zbiór liczb całkowitych:

```
1 {10, 20, 50, 100}
```

Pusty zbiór tworzymy przy użyciu konstruktora `set()`.

Zbiór może zawierać jedynie elementy unikalne, dlatego też powielenie któregokolwiek z elementów nie wpływa na zwiększenie jego długości:

```
1 >>> liczby = {10, 20, 50, 100, 10}
2 >>> liczby
3 {100, 10, 20, 50}
4 >>> len(liczby)
5 4
```

Zbiór nie zachowuje oryginalnej kolejności przechowywanych elementów, dlatego też w przypadku zbiorów nie jest możliwe wykorzystanie operatora dostępu do elementu `[]` oraz wycinania. Korzystając jednak z konstruktorów `list()` lub `tuple()` możemy w łatwy sposób przekonwertować zbiór do kolekcji wspierającej indeksowanie.

Funkcja `len()` oraz operator zawierania `in` działają w analogiczny sposób do pozostałych kolekcji.

Zaletą typu `set` jest obsługa standardowych operatorów dla zbiorów matematycznych, tzn.:

- suma - `|`
- różnica - `-`
- iloczyn - `&`
- różnica symetryczna - `^`

W poniżej sesji interaktywnego środowiska zostało zaprezentowana użycie operatorów dla typu `set`:

```
1 >>> a = {1, 2, 3}
2 >>> b = {2, 3, 4}
3 >>> a | b
4 {1, 2, 3, 4}
5 >>> a - b
6 {1}
7 >>> a & b
8 {2, 3}
9 >>> a ^ b
10 {1, 4}
```

Dodanie elementu do zbioru realizowane jest poprzez metodę `add`:

```
1 >>> liczby = {10, 20, 50, 100}
2 >>> liczby.add(200)
3 >>> liczby
4 {10, 20, 50, 100, 200}
```

Słowniki

Słownik jest mutowalną strukturą danych służącą do przechowywania asocjacji (powiązań) pomiędzy kluczem a zadaną wartością. W odróżnieniu od tupli i list, indeksami w słowniku są dowolne niemutowalne wartości - tzn. liczby, napisy czy też tuple (zawierające jedynie niemutowalne elementy).

Słowniki tworzy się, tak samo jak zbiory, przy wykorzystaniu nawiasów klamrowych - {}. W odróżnieniu od zbiorów, elementami słownika jest lista par klucz-wartość w postaci klucz: wartość rozdzielonych znakiem przecinka ,. W ramach jednego słownika klucze muszą być unikalne. Przykład utworzenia słownika:

```
1 {'jan': 4, 'ola': 5, 'dominik': 3}
```

Stworzenie pustego słownika możemy osiągnąć poprzez użycie pustych nawiasów {} lub konstruktora dict().

W celu pobrania elementu odpowiadającego zadanemu kluczowi używamy operatora dostępu do elementu []:

```
1 >>> oceny = {'jan': 4, 'ola': 5, 'dominik': 3}
2 >>> oceny['jan']
3 4
```

Dodanie nowego elementu do słownika odbywa się także poprzez użycia operatora [] oraz operatora przypisania = jako klucza wystarczy użyć obiektu nie występującego jeszcze w słowniku:

```
1 >>> oceny = {'jan': 4, 'ola': 5, 'dominik': 3}
2 >>> oceny['tomasz'] = 2
3 >>> oceny
4 {'dominik': 3, 'jan': 4, 'ola': 5, 'tomasz': 2}
```

Funkcja len(), operator zawierania in oraz wyrażenie del działają w analogiczny sposób do pozostałych kolekcji:

```
1 >>> oceny = {'jan': 4, 'ola': 5, 'dominik': 3}
2 >>> len(oceny)
3 3
4 >>> 'jan' in oceny
5 True
6 >>> del oceny['dominik']
7 >>> oceny
8 {'jan': 4, 'ola': 5}
```

Napisy

Napisy były omawiane już jako jeden z typów podstawowych w języku Python. Możemy traktować je jednak także jako bardziej złożoną strukturę - niemutowalną kolekcję, której elementami są poszczególne znaki wchodzące w skład napisu.

Typ `str` wspiera operator dostępu do elementu `[]`, wycinanie, operator zawierania oraz funkcję `len()` analogicznie do innych kolekcji:

```
1 >>> napis = 'test'
2 >>> len(napis)
3 4
4 >>> napis[0]
5 't'
6 >>> napis[1:3]
7 'es'
8 >>> 't' in napis
9 True
10 >>> 'tes' in napis
11 True
```

Instrukcja warunkowa

Instrukcja warunkowa jest jednym z najważniejszych i fundamentalnych elementów składniowych prawie każdego języka programowania. Dzięki instrukcji warunkowej możemy zdecydować o wykonaniu lub pominięciu wybranego fragmentu naszego programu.

W języku Python do zapisu instrukcji warunkowej używamy słowa kluczowego `if` po którym następuje warunek logiczny zakończony znakiem dwukropka `:`. Jeżeli warunek ten przyjmuje wartość `True` wykonany zostanie blok kodu znajdujący się tuż pod instrukcją warunkową.

W celu utworzenia bloku kodu w języku Python wykorzystywane są tzw. wcięcia, czyli białe znaki (spacja lub tabulacja) występujące na początku linii. Dobrą praktyką stosowaną przez większość programistów języka Python jest używanie 4 znaków spacji w celu wcięcia kodu o jeden poziom. Większość popularnych edytorów kodu oraz IDE (w tym PyCharm) umożliwiają wstawienie 4 znaków spacji poprzez jednokrotne użycia klawisza Tab.

Poniższy program przedstawia użycie instrukcji warunkowej. Jej warunek logiczny jest prawdziwy, dlatego następujący po niej blok kodu zostanie wykonany:

```
1 a = 5
2 if a < 10:
3     print('a jest mniejsze od 10')
```

Jeżeli warunek instrukcji `if` w wyniku daje wartość `False`, jej blok nie zostanie wykonany:

```
1 a = 5
2 if a == 4:
3     print('a jest rowne 4')
```

W celu dostarczenia bloku kodu, który wykona się w przypadku nie spełnienia zadanego warunku logicznego, używamy słowa kluczowego **else** wciętego na tym samym poziomie co słowo kluczowe **if** rozpoczynające instrukcję warunkową. Alternatywny blok kodu umieszczamy odpowiednio wcięty na końcu całej instrukcji. Poniżej zaprezentowano przykład użycia słowa kluczowego **else**:

```
1 a = 5
2 if a > 10:
3     print('a jest wieksze od 10')
4 else:
5     print('a jest mniejsze od lub rowne 10')
```

Częstym zadaniem jest konieczność sprawdzenia wielu warunków jednocześnie oraz wykonania bloku kodu tylko jednego z nich. W tym celu konieczne jest powiązanie wielu instrukcji warunkowych w jedną korzystając z słowa kluczowego **elif**. Podejście to różni się od wielu następujących po sobie oddzielnych instrukcji warunkowych - korzystając z słowa kluczowego **elif** mamy pewność, że w ramach naszej instrukcji wykona się maksymalnie jeden blok kodu.

```
1 a = 5
2 if a > 7:
3     print('a jest wieksze od 7')
4 elif a > 3:
5     print('a jest wieksze od 3')
6 elif a > 1:
7     print('a jest wieksze od 1')
8 else:
9     print('a jest mniejsze od lub rowne 1')
```

Po uruchomieniu powyższego programu na konsolę zostanie wypisany komunikat **a jest wieksze od 3**. Pomimo, że 3 warunek także jest spełniony ($a > 1$) to następujący po nim blok kodu nie zostanie wykonany.

Jeżeli warunek logiczny pętli nie jest wartością logiczną lub nie jest operacją, której rezultatem jest taka wartość, zostanie on poddany automatycznej konwersji przy pomocy konstruktora **bool()**. Częstym sposobem wykorzystania tego zachowania jest użycie instrukcji warunkowej do sprawdzenia występowania elementów w kolekcji lub też znaków w napisie:

```
1 lista = []
2 if lista:
3     print('lista posiada elementy')
4 else:
5     print('lista jest pusta')
```

Instrukcja pass

Ze względów składniowych konieczne jest utworzenie wciętego bloku kodu po nagłówku instrukcji warunkowej, pętli, funkcji, klasy, etc. Jeżeli chcemy aby ciało takiego bloku pozostało puste (np. tymczasowy brak implementacji), możemy do tego celu wykorzystać instrukcję `pass`. Instrukcja ta nie wykonuje żadnej operacji.

Poniżej przedstawiono instrukcję warunkową z następującym po niej "pustym" blokiem:

```
1 if True:  
2     pass
```

Pętle

Pętle służą do wykonywania zadanego fragmentu kodu wielokrotnie. W składni języka Python mamy dostępne dwie pętle:

- pętla `while`
- pętla `for`

Pętla while

Pętla `while` swoją składnią przypomina najprostszą wersję instrukcji warunkowej, w której słowo kluczowe `if` zastąpiono słowem kluczowym `while`. Różnica pomiędzy pętlą `while` a instrukcją warunkową polega na wykonywaniu bloku pętli wielokrotnie, aż do momentu kiedy jej warunek logiczny przestanie być prawdziwy.

Poniższy przykład przedstawia program, którego zadaniem jest wypisanie liczb od 0 do 10:

```
1 liczba = 0  
2 while liczba <= 10:  
3     print(liczba)  
4     liczba = liczba + 1
```

Blok powyższej pętli `while` wykonuje się 10 razy, aż wartość zmiennej `liczba` osiągnie wartość 11 przez co nie zostanie spełniony warunek `liczba <= 10` i pętla zakończy swoje działanie.

Pętla for

Pętla `for` służy do przechodzenia (tzw. iteracji) po kolejnych elementach w zdanej kolekcji. Jej nagłówek, poza słowem kluczowym `for`, zawiera nazwę zmiennej do której będą przypisywane

kolejne element oraz kolekcję po której odbędzie się iteracja. Oba elementy rozdzielone są słowem kluczowym `in`.

Przykład przedstawiony poniżej pokazuje iterację po wszystkich elementach listy zawierającej napisy:

```
1 uczestnicy = ['Jan', 'Sylwia', 'Adam', 'Ola']
2 for uczestnik in uczestnicy:
3     print(uczestnik)
```

Podczas każdej iteracji powyższej pętli `for` do zmiennej `uczestnik` będą przypisywane kolejne elementy listy `uczestnicy`. Blok kodu następujący po nagłówku pętli `for` zostanie wykonany 4 razy – tyle elementów znajduje się w zadanej kolekcji.

Wyrażenie `break` i `continue`

Zarówno wewnątrz pętli `while` jak i pętli `for` możemy użyć dwóch dodatkowych wyrażeń w celu kontroli iteracji naszych pętli: `break` oraz `continue`

Wyrażenie `break` służy do natychmiastowego przerwania iteracji pętli. W poniższym przykładzie pętla `for` wykonuje się do momentu gdy nie natrafi na element listy o zadanym warunku logicznym – liczba > 10 :

```
1 liczby = [1, 3, 4, 15, 2, 7]
2 for liczba in liczby:
3     print(liczba)
4     if liczba > 10:
5         break
```

Wyrażenie `continue` służy do pominięcia dalszej części kodu wewnątrz pętli i rozpoczęcie kolejnej iteracji. Jednym ze sposobów wypisanie wszystkich liczb większych od 10 w zadanej kolekcji mógłby być poniższy zapis wykorzystujący wyrażenie `continue`:

```
1 liczby = [2, 12, 4, 15, 1]
2 for liczba in liczby:
3     if liczba <= 10:
4         continue
5     print(liczba)
```

Blok `else`

Obie pętle języka Python posiadają możliwość dołączenia bloku `else` na końcu pętli. Zostanie on wykonany jednorazowo po zakończeniu pętli, nawet jeżeli nie została wykonana ani jedna iteracja.

Blok **else** nie zostanie wykonany jedynie w przypadku zakończenia pętli z użyciem wyrażenia **break**.

Z zachowania tego możemy skorzystać w przypadku użycia pętli do znalezienia elementu o zadanych warunkach w kolekcji:

```
1 liczby = [1, 5, 8, 9, 7]
2 for liczba in liczby:
3     if liczba > 10:
4         print('znaleziono liczbe wieksza od 10')
5         break
6 else:
7     print('nie znaleziony liczby wiekszej od 10')
```

W powyższym przypadku blok występujący po instrukcji **else** zostanie wykonany tylko wtedy, kiedy iteracja pętli **for** zakończy się w naturalny sposób, tzn. bez użycia wyrażenia **break**.

Funkcja range()

Często wykorzystywaną funkcją w kontekście pętli jest funkcja **range()** służąca do wygenerowania sekwencji liczb naturalnych z zadanego zakresu. Funkcję **range()** możemy wywołać dwojako:

- z jednym parametrem - oznaczającym liczbę o jeden większą niż ostatni element w tworzonej sekwencji, pierwszym elementem sekwencji będzie liczba 0,
- z dwoma lub trzema parametrami - pierwszy parametr oznacza pierwszy element sekwencji, drugi - element o jeden większy niż ostatni element, trzeci - krok używany do tworzenia sekwencji (tak samo jak przy wycinaniu).

Poniższy przykład wypisze na konsolę liczby od 0 do 9:

```
1 for liczba in range(10):
2     print(liczba)
```

W celu wypisania co drugiej liczby z przedziału 4-20 musielibyśmy użyć następującego wywołania funkcji **range()**:

```
1 for liczba in range(4, 21, 2):
2     print(liczba)
```

Wyrażenie listowe i inne

Składnia języka Python umożliwia w prosty sposób tworzenie nowych list na podstawie już istniejących kolekcji przy wykorzystaniu wyrażenia listowego (ang. *list comprehensions*). Jest ono połączeniem pętli **for** wraz z możliwą transformacją aktualnego elementu w jednej linii kodu.

Poniższy przykład przedstawia utworzenie listy zawierającej kwadraty liczb z przedziału 0-9:

```
1 kwadraty = []
2 for liczba in range(10):
3     kwadraty.append(liczba ** 2)
```

Ta sama operacja może zostać zapisana w postaci jednej linii przy użyciu wyrażenia listowego:

```
1 kwadraty = [liczba ** 2 for liczba in range(10)]
```

Ten skrócony zapis tworzenia nowej kolekcji jest możliwy także dla wszystkich innych podstawowych struktur danych - tupli, słownika oraz zbioru. W przypadku zapisu wykorzystującego składnię tupli, tzn. nawiasy okrągłe () wynikiem nie jest jednak obiekt typu tuple a generator.

```
1 kwadraty_generator = (liczba ** 2 for liczba in range(10))
2 kwadraty_zbior = {liczba ** 2 for liczba in range(10)}
3 kwadraty_słownik = {liczba: liczba ** 2 for liczba in range(10)}
```

Funkcje

Funkcja jest wydzielonym fragmentem naszego programu, którego zadaniem jest przeprowadzenie obliczeń na podstawie zadanych parametrów oraz zwrócenie wyniku lub wykonanie jakiejś akcji.

Przykładem funkcji, którą już poznaliśmy, jest funkcja `len()` - przyjmuje ona jeden parametr (kolekcję), a jej wynikiem jest liczba całkowita odpowiadająca długości zadanej kolekcji. Innym przykładem jest funkcja `print()` - jako parametr przyjmuje ona komunikat, który ma zostać wypisany na konsolę. W przeciwieństwie do funkcji `len()`, funkcja `print()` nie zwraca żadnego wyniku.

W celu użycia (tzw. wywołania) funkcji musimy podać jej nazwę oraz listę argumentów rozdzielonych znakiem przecinka , pomiędzy nawiasami okrągłymi (). Jeżeli funkcja nie przyjmuje żadnych argumentów (lub przyjmuje jedynie argumenty opcjonalne) używamy pustych nawiasów () .

Zdefiniowanie własnej funkcji odbywa się poprzez użycie słowa kluczowego `def`, po którym następuje nazwa funkcji, lista parametrów funkcji pomiędzy nawiasami () zakończona znakiem dwukropka : oraz blok kodu stanowiący jej ciało. W celu zwrócenia wyniku z funkcji wykorzystywane jest słowo kluczowe `return` po którym następuje zwracana wartość. Wyrażenie `return` może wystąpić w ciele funkcji wielokrotnie.

Poniższy przykład przedstawia definicję funkcji o nazwie `pole_trojkata`, która przyjmuje dwa parametry i zwraca w wyniku obliczone pole trójkąta:

```
1 def pole_trojkata(podstawa, wysokosc):
2     return 1 / 2 * podstawa * wysokosc
```

Wywołanie tej funkcji w dalszej części programu mogłoby wyglądać następująco:

```
1 wynik = pole_trapezu(10, 5)
2 print(f'Pole zadanej trapezu to {wynik}')
```

Funkcja nie musi przyjmować żadnego argumentu oraz nie musi także zwracać wyniku:

```
1 def hello_world():
2     print('Hello World!')
```

Argumenty z wartością domyślną Argumenty funkcji mogą przyjmować także wartości domyślne - w tym celu należy użyć znaku = podczas definiowania argumentu w nagłówku funkcji.

```
1 def wypisz_hello(name='Jan'):
2     print(f'Hello {name}!')
```

Funkcja wypisz_hello może zostać wywołana na dwa sposoby: bez żadnego argumentu (wtedy name przyjmie wartość 'Jan') lub z jednym argumentem.

```
1 wypisz_hello()
2 wypisz_hello('Tomek')
```

Argumenty kluczowe Wywołanie funkcji z długą listą przyjmowanych argumentów może wyglądać nieczytelnie. W tym celu składnia języka Python zezwala na wywoływanie funkcji wraz z bezpośrednim podaniem nazw argumentów. Tak użyte argumenty nazywamy argumentami kluczowymi.

W poniższym przykładzie funkcja o nazwie pole_trapezu przyjmuje 3 argumenty. Jej wywołanie z wykorzystaniem argumentów kluczowych wyglądałoby następująco:

```
1 pole_trapezu(pierwsza_podstawa=10, druga_podstawa=5, wysokosc=8)
```

Wywołanie to jest o wiele bardziej czytelne niż równoważne mu pole_trapezu(10, 5, 8) - z powyższego przykładu od razu widać do których argumentów przypisane zostaną wartości, nawet bez znajomości definicji funkcji. Użycie argumentów kluczowych nie wymaga zachowania oryginalnej kolejności listy argumentów, jak ma to miejsce podczas wywoływania z użyciem argumentów niekluczowych. Dopuszczalne jest także mieszanie obu technik, tzn. przekazanie części argumentów jako kluczowe, a części jako niekluczowe. Argumenty niekluczowe muszą jednak występować przed kluczowymi i muszą one zachować oryginalną kolejność listy argumentów:

Zmienna liczba argumentów Funkcje mogą także przyjmować zmienną liczbę argumentów. W celu zaimplementowania takiej funkcji korzystamy ze znaku * do zebrania argumentów niekluczowych oraz podwójnego znaku ** w celu przechwycenia argumentów kluczowych. Argument

oznaczony znakiem `*` będzie tuplą argumentów niekluczowych, a argument oznaczony `**` stanie się słownikiem argumentów kluczowych.

Poniższa funkcja wypisuje na konsolę wszystkie argumenty przekazane do funkcji podczas jej wywołania:

```
1 def wypisz_argumenty(*args, **kwargs):
2     print(f'Argumenty niekluczowe: {args}')
3     print(f'Argumenty kluczowe: {kwargs}')
```

Zwyczajowo tuplę z argumentami niekluczowymi nazywa się `args`, a słownik z argumentami kluczowymi `kwargs`. Na liście argumentów `*args` musi wystąpić zawsze przed `**kwargs`, które z kolei musi wystąpić zawsze jako ostatni argument.

Programowanie Obiektowe

Często używanym terminem w kontekście klasyfikacji języków programowania jest pojęcie paradygmatu programowania. Termin ten określa sposób projektowania i tworzenia oprogramowania oraz definiuje podstawowe elementy używane przy implementacji danego rozwiązania jak i sposób ich interakcji.

Jednym z najpopularniejszych paradygmantów programowania jest programowanie proceduralne. W podejściu tym głównym elementem kodu, na który kładziony jest największy nacisk, są funkcje (nazywane też procedurami). Wykorzystywane są one do dzielenia kodu na mniejsze fragmenty, zamkające w sobie rozwiązania poszczególnych składników problemu. Interakcja funkcji w kodzie proceduralnym polega na ich wzajemnym wywoływaniu się.

Paradygmat programowania proceduralnego nie sprawdza się jednak w przypadku projektowania bardziej skomplikowanych rozwiązań. Modelowanie złożonych problemów w oparciu jedynie o funkcje jest zadaniem bardzo trudnym.

Paradygmatem próbującym ułatwić projektowanie tego typu rozwiązań jest paradygmat programowania obiektowego. Technika ta polega na opieraniu tworzonych programów o interakcję obiektów, czyli bytów posiadających zarówno swój stan jak i zestaw czynności, które mogą zostać na nich wykonane.

Python jest językiem wspierającym wiele paradygmantów programowania. Możliwość definiowania własnych funkcji pozwala nam tworzyć rozwiązania w oparciu o paradygmat programowania proceduralnego, tworzenie własnych typów zapewnia wsparcie paradygmatu programowania obiektowego.

Klasy

Klasy pozwalają nam definiować własne typy posiadające zarówno swój stan jak i zestaw zachowań. Na podstawie klas możemy tworzyć obiekty, nazywane instancjami danej klasy.

W celu zdefiniowania klasy w języku Python wykorzystane jest słowo kluczowe `class` po którym następuje nazwa tworzonej klasy oraz blok kodu zawierający jej ciało.

Dobrą praktyką jest nazywanie klas wykorzystując "wielbłędzi" system notacji. Polega on na rozpoczęciu każdego słowa składającego się na nazwę klasy od dużej litery, np. `ToJestNazwaKlasy`.

Poniższy przykład pokazuje definicję klasy o nazwie `Samochod` z pustym ciałem:

```
1 class Samochod:  
2     pass
```

Tak utworzona definicja klasy pozwala nam na tworzenie nowych instancji na jej podstawie. W tym celu używamy nazwy klasy oraz nawiasów okrągłych () tak samo jak w przypadku wywoływania funkcji:

```
1 moj_samochod = Samochod()
```

Tak jak liczba 1 jest instancją klasy `int`, tak samo wartość przypisana do zmiennej `moj_samochod` jest instancją klasy `Samochod`.

Przedstawiona klasa jest bardzo prosta - nie posiada ona żadnych zachowań ani nie przechowuje żadnego stanu. W celu rozbudowania naszej klasy musimy zaimplementować w niej metody.

Metody

Metodą nazywamy funkcję zdefiniowaną w obrębie klasy. Dzięki metodom możemy do naszych klas dodać specyficzne dla nich zachowania, ale także wpływać na proces tworzenia ich instancji.

Wszystkie metody (poza metodami statycznymi oraz klasowymi - więc o nich w dalszej części tego rozdziału) zawsze jako swój pierwszy argument przyjmują instancję na rzecz której są wywoływane. Argument ten zwyczajowo nazywany jest `self`, chociaż jest to tylko konwencja a nie wymóg składniowy.

W poniższym przykładzie zaimplementowano klasę wraz z jedną metodą:

```
1 class Samochod:  
2     def wypisz_specyfikacje(self):  
3         print('To jest samochód!')
```

Taka definicja klasy pozwala nam na następujące wykorzystanie nowej metody:

```
1 moj_samochod = Samochod()
2 moj_samochod.wypisz_specyfikacje()
```

Wywoływanie metody na rzecz danej instancji odbywa się poprzez użycia znaku kropki . pomiędzy instancją i metodą, którą chcemy wywołać. W przypadku metod nie musimy przekazywać explicitie jej pierwszego argumentu - język Python sam przypisze do niego instancję na rzecz której wywoływana jest metoda. W powyższym przykładzie wartość argumentu self w metodzie wypisz_specyfikacje jest równa wartości zmiennej moj_samochod.

Metoda specjalna __init__

W celu przechowywania danych powiązanych z konkretną instancją obiektu konieczne jest wykorzystanie metody specjalnej __init__. Metoda ta wywoywana jest jako pierwsza metoda po utworzeniu instancji naszej klasy i przyjmuje ona wszystkie argumenty użyte podczas tworzenia obiektu.

Częstą praktyką jest wykorzystanie metody specjalnej __init__ do tworzenia atrybutów (nazywanych także polami) naszej instancji. Atrybuty wykorzystywane są do przechowywania stanu przypisanego do danego obiektu. W celu utworzenia nowego atrybutu używamy operatora przypisania =.

Poniższa definicja klasy posiada implementację metody __init__ przyjmującej jeden argument (poza self) oraz drugą metodę, która wykorzystuje utworzony wcześniej atrybut:

```
1 class Samochod:
2     def __init__(self, nazwa):
3         self.nazwa = nazwa
4
5     def wypisz_informacje(self):
6         print(f'To jest samochód {self.nazwa}!')
```

Utworzenie instancji obiektu zdefiniowanej klasy wymaga teraz podania jednego dodatkowego parametru:

```
1 moj_samochod = Samochod('Porsche')
```

Dostęp do nowo utworzonego atrybutu możliwy jest poprzez wykorzystanie znaku . - zarówno z poza jak i z wnętrza ciała metod.

```
1 moj_samochod = Samochod('Porsche')
2 print(moj_samochod.nazwa)
3 moj_samochod.wypisz_informacje()
```

Nazwa atrybutu oraz nazwa argumentu przyjmowanego przez metodę specjalną __init__ nie muszą być takie same, jednak stosowanie takiego nazewnictwa jest dobrą praktyką.

Metody statyczne

Jeżeli nasza metoda nie wykorzystuje instancji na rzecz której jest wywoływana (tzn. nie używa argumentu `self`) możemy taką metodę oznaczyć jako metodę statyczną.

Do tego celu wykorzystujemy wbudowany w język Python dekorator `@staticmethod` umieszczany przed wybraną metodą. Dekoratorami nazywamy funkcje, które zmieniają zachowanie innych funkcji. Ich użycie polega na umieszczeniu ich nad nagłówkiem funkcji, rozpoczynając ich nazwę od znaku `@`.

Do metod oznaczonych dekoratorem `@staticmethod` nie przekazywana jest instancja obiektu na rzecz którego metoda ta została wywołana. Metody takie mogą być wywoływanie także na rzecz klasy.

```
1 class Samochod:
2     @staticmethod
3     def wypisz_jednostke_napedowa():
4         print('Samochod napedza silnik.')
```

Powyższą metodę statyczną można wywołać na dwa sposoby: wykorzystując klasę oraz wykorzystując instancję klasy. Poniższy przykład obrazuje oba sposoby:

```
1 Samochod.wypisz_jednostke_napedowa()
2
3 moj_samochod = Samochod()
4 moj_samochod.wypisz_jednostke_napedowa()
```

Metody klasowe

Metodami klasowymi nazywamy metody oznaczone dekoratorem `@classmethod`. Tak samo jak w przypadku zwykłych metod, przekazywany jest do nich automatycznie pierwszy argument, ale nie jest nim instancja, a klasa na rzecz której wywoływana jest dana metoda.

Pierwszy argument metod klasowych zwyczajowo nazywany jest `cls`. Poniższy przykład obrazuje użycie metody klasowej do utworzenia instancji danej klasy:

```
1 class Samochod:
2     def __init__(self, nazwa):
3         self.nazwa = nazwa
4
5     @classmethod
6     def utwierz_super_samochod(cls):
7         return cls('Porsche')
```

Metody klasowe, tak samo jak metody statyczne, mogą być wywoływanie zarówno na rzecz klasy jak i jej instancji:

```
1 super_samochod = Samochod.utworz_super_samochod()
2
3 moj_samochod = Samochod()
4 super_samochod = moj_samochod.utworz_super_samochod()
```

Dynamiczne atrybuty

W celu utworzenia atrybutu, którego wartość wyliczana jest dynamicznie na podstawie stanu danej instancji, konieczne jest skorzystanie z dekoratora `@property`. Metody oznaczone tym dekoratorem wywoływane są bez użycia nawiasów okrągłych (), przez co przypominają zwykłe atrybutu.

Poniższa definicja klasy posiada metodę oznaczoną dekoratorem `@property`, która oblicza wartość atrybutu na podstawie innych pól:

```
1 class Samochod:
2     def __init__(self, waga_w_kg):
3         self.waga_w_kg = waga_w_kg
4
5     @property
6     def waga_w_tonach(self):
7         return self.waga_w_kg / 1000
```

Wykorzystanie zdefiniowanego atrybutu mogłoby wyglądać następująco:

```
1 samochod = Samochod(1500)
2 print(samochod.waga_w_tonach)
```

Pozostałe metody specjalne

Poza metodą specjalną `__init__` istnieje szereg innych metod, które możemy zaimplementować w naszej klasie nadając jej przez to specjalne zachowania. Wszystkie z nich rozpoczynają się i kończą podwójnym znakiem podkreślenia `__`.

Przykładem takiej metody może być metoda specjalna `__str__`. Jest ona wywoływana w momencie konwersji naszego obiektu na napis.

Poniższy przykład obrazuje wykorzystanie metody `__str__` do wypisania informacji o instancji:

```
1 class Samochod:
2     def __init__(self, nazwa):
3         self.nazwa = nazwa
4
```

```
5     def __str__(self):
6         return f'To jest samochod {self.nazwa}'
7
8 moj_samochod = Samochod('Porsche')
9 print(moj_samochod)
```

Podczas wywołania funkcji `print()` następuje automatyczna konwersja naszego obiektu na napis, co skutkuje wywołaniem metody `__str__` na rzecz naszej instancji.

Dzięki implementacji metod specjalnych możemy upodobnić nasze własne typy do typów wbudowanych, zapewniając obsługę m.in.:

- operatorów porównania (np. `x.__lt__(y)` odpowiada wywołaniu `x < y`)
- operatorów arytmetycznych (np. `x.__add__(y)` odpowiada wywołaniu `x + y`)
- operatorów używanych w kontekście kolekcji (np. `x.__getitem__(y)` odpowiada wywołaniu `x[y]`)

Enkapsulacja

Jednym z ważnych aspektów programowania obiektowego jest możliwość ukrywania wewnętrznych detali implementacyjnych naszej klasy przed jej użytkownikami. Dzięki temu klasy mogą w łatwy sposób rozdzielić swoje wewnętrzne szczegóły działania od zewnętrznego interfejsu określającego ich sposób interakcji. Podejście to nazywa się enkapsulacją (lub też hermetyzacją).

Język Python nie posiada żadnych mechanizmów składniowych wspierających enkapsulację a jedynie ogólnie przyjęte konwencje. Z jednej strony podejście to zapewnia dużą elastyczność ale nakłada też na programistę wymóg scistego trzymania się ustalonych zasad, nie pilnowanych przez interpretera języka.

Przyjętą w języku Python konwencją oznaczania atrybutów i metod jako prywatne, jest rozpoczęwanie ich nazwy od znaku podkreślenia `_`. Korzystanie z tego typu atrybutów i metod powinno odbywać się jedynie z wnętrza innych metod danej klasy.

Poniższy przykład pokazuje klasę wraz z jednym atrybutem oraz jedną metodą prywatną oraz ich wykorzystanie w innej metodzie należącej do klasy.

```
1 class Sejf:
2     def __init__(self, dane, haslo):
3         self._dane = dane
4         self._haslo = haslo
5
6     def pobierz_dane(self, haslo):
7         if haslo == self._haslo:
8             return self._dane
9         else:
```

```
10         self._zniszcz_sejf()
11         return None
12
13     def _zniszcz_sejf(self):
14         self._dane = None
15         self._haslo = None
```

W naszym przykładzie interfejsem, czyli widocznym dla użytkowników sposobem komunikacji z naszą klasą, jest metoda inicjalizująca instancję `__init__` oraz metoda `pobierz_dane`. Nie chcemy aby użytkownik naszej klasy miał bezpośredni dostęp do naszych danych (atrybut `_dane` i `_haslo`) oraz metody prywatnej (metoda `_zniszcz_dane`).

Dziedziczenie

Dziedziczenie umożliwia nam rozszerzanie zachowań istniejących klas poprzez tworzenie nowych typów na ich podstawie.

W celu utworzenia nowego typu bazującego na już istniejącej klasie, należy nazwę klasy bazowej (`KlasaA`) wpisać pomiędzy nawiasy okrągłe tuż po nazwie nowo tworzonej klasy (`KlasaB`) w jej nagłówku.

```
1 class KlasaA:
2     pass
3
4 class KlasaB(KlasaA):
5     pass
```

Instancje klasy dziedziczącej (`KlasaB`) będą posiadać wszystkie atrybuty oraz metody z klasy nadzędnej (`KlasaA`) oraz dodatkowe atrybuty i metody zdefiniowane w klasie dziedziczącej.

Poniższy przykład obrazuje prostą hierarchię klas oraz ich użycie:

```
1 class Samochod:
2     def jedz(self, nazwa_celu):
3         print(f'Jade do celu {nazwa_celu}')
4
5 class ElektrycznySamochod(Samochod):
6     def laduj(self):
7         print('Proces ładowania rozpoczęty')
8
9
10 elektryczny = ElektrycznySamochod()
11 elektryczny.laduj()
12 elektryczny.jedz('New York')
```

Jak widać obiekt stworzony przy wykorzystaniu klasy dziedziczącej posiada zarówno metodę z klasy nadzędnej jak i metodę zdefiniowaną w klasie podrzędnej.

Nadpisywanie metod i funkcja super()

Jednym z częstych powodów dziedziczenia po danej klasie jest chęć zmiany jej zachowania, które zostało już jednak zdefiniowane w jednej z metod klasy nadrzędnej.

Python umożliwia dostarczenie nowej implementacji metody w klasie dziedziczącej - metodę taką wystarczy nazwać tak samo jak nazywa się ona w klasie bazowej.

W celu uzyskania dostępu do oryginalnej wersji nadpisywanej metody należy skorzystać z funkcji `super()`. Obiektem zwracanym przez funkcję `super()` jest instancja danego obiektu ale w kontekście klasy bazowej.

```
1 class Samochod:
2     def jedz(self, nazwa_celu):
3         print(f'Jade do celu {nazwa_celu}')
4
5 class ElektrycznySamochod(Samochod):
6     def jedz(self, nazwa_celu):
7         print('Obliczanie naladowania baterii')
8         super().jedz(nazwa_celu)
```

W powyższym przykładzie metoda w klasie podrzędnej rozszerza zachowanie metody z klasy nadrzędnej. Korzysta ona jednak także z implementacji z klasy bazowej, wykorzystując do tego metodę `super()`.

Wielodziedziczenie

Klasy w języku Python mogą dziedziczyć z wielu klas bazowych. W celu podania listy klas nadrzędnych, należy rodzielić je znakiem , w nagłówku funkcji pomiędzy nawiasami okrągłymi.

```
1 class KlasaA:
2     pass
3
4 class KlasaB:
5     pass
6
7 class KlacaC(KlasaA, KlasaB):
8     pass
```

Przeszukiwanie klas bazowych w celu znalezienia atrybutu lub metody odbywa się od lewej do prawej strony. Ma to szczególne znaczenia jeżeli dwie lub więcej z klas bazowych posiadają tak samo nazywający się atrybut lub metodę.

Wyjątki

Mechanizm wyjątków, jak sama nazwa wskazuje, służy do sygnalizowania wyjątkowych sytuacji podczas działania programu. Wyjątki dają możliwość łatwego przekierowania wykonania programu w miejsce ich obsługi.

W kontekście mechanizmu wyjątków występują dwa ważne pojęcia:

- generowanie wyjątków, nazywane też “rzucaniem” wyjątków,
- obsługa wyjątków, nazywana też “łapaniem” wyjątków.

Oba z tych terminów mają swoje odzwierciedlenie w składni języka Python.

Generowanie wyjątków

Wyjątek może zostać wygenerowany podczas wykonania naszego programu z wielu powodów. Jego źródłem może być niedozwolona operacja, próba dostępu do nieistniejących danych lub też może zostać on wygenerowany z wykorzystaniem wyrażenia `raise`.

Prostym sposobem na wygenerowanie wyjątku jest próba podzielenia dowolnej liczby przez 0. Operacja ta jest operacją niedozwoloną i spowoduje wygenerowania wbudowanego wyjątku `ZeroDivisionError`.

```
1 >>> 1 / 0
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4     ZeroDivisionError: division by zero
```

Innym sposobem na wygenerowanie wyjątku jest wykorzystanie wyrażenia `raise` po którym musi nastąpić utworzenie instancji wyjątku.

Poniższy przykład pokazuje wykorzystanie wbudowanego w język wyjątku `ValueError` do zakomunikowanego błędnych argumentów przekazanych do funkcji:

```
1 def pole_trojkota(podstawa, wysokosc):
2     if podstawa <= 0 or wysokosc <= 0:
3         raise ValueError('argumenty musza byc dodatnie')
4     return 1 / 2 * podstawa * wysokosc
```

Po wygenerowaniu wyjątku w naszym programie interpreter języka Python szuka najbliższego miejsca gdzie wyjątek ten może zostać obsłużony. Jeżeli kod obsługi wyjątku nie zostanie znaleziony to wykonanie programu zostanie zakończone.

Obsługa wyjątków

Obsługa wyjątków możliwa jest dzięki wyrażeniu **try**. Wyrażenie to składa się z bloku kodu występującego po słowie kluczowym **try** oraz jednej lub więcej sekcji **except** służących do łapania wyjątków, umieszczonych na końcu całego wyrażenia.

Najprostszy sposób złapania wyjątku wygląda następująco:

```
1 try:  
2     wynik = 1 / 0  
3 except:  
4     print('Blad obliczen')
```

Pomimo wygenerowania wyjątku wynikającego z dzielenia przez 0, nasz program nie zakończy swojego działania, a jego sterowanie zostanie przeniesione do najbliższej sekcji **except** gdzie możemy zareagować na pojawienie się wyjątkowej sytuacji.

Pozostawienie sekcji **except** w tej formie spowoduje przechytywanie wszystkich wyjątków jakie wystąpią w bloku wyrażenia **try**. Tego typu podejście nie jest dobrą praktyką, ponieważ potencjalnie ukrywa ono przed nami wszystkie sytuacje wyjątkowe - nawet te których nie chcemy obsługiwać.

W celu przechwycenia wyjątku jedynie wskazanego typu należy podać klasę wyjątku w nagłówku sekcji **except**:

```
1 try:  
2     wynik = 1 / 0  
3 except ZeroDivisionError:  
4     print('Blad obliczen')
```

Jeżeli chcemy obsługiwać większą liczbą wyjątków możemy dodać do naszego wyrażenia **try** kolejne sekcje **except**:

```
1 try:  
2     wynik = 1 / slownik['dzielnik']  
3 except ZeroDivisionError:  
4     print('Blad dzialenia przez zero')  
5 except KeyError:  
6     print('Brak wartosci w slowniku')
```

Korzystając ze słowa kluczowego **as** możemy podczas obsługi wyjątku dostać się do rzuconej instancji:

```
1 try:  
2     wynik = 1 / slownik['dzielnik']  
3 except KeyError as wyjatek:  
4     print(f'Brak wartosci w slowniku - argumenty wyjatku: {wyjatek.args  
}')
```

W przypadku gdy jedna sekcja `except` powinna obsługiwać wyjątki wielu typów, musimy dostarczyć ich tuplę do danej sekcji:

```
1 try:
2     wynik = 1 / slownik['dzielnik']
3 except (ZeroDivisionError, KeyError):
4     print('Wystapil blad')
```

Gdy podczas łapania wyjątku sekcja `except` nie potrafi w satysfakcyjny sposób zakończyć obsługi wyjątku, możliwe jest rzucenie dalej złapanego wyjątku. W tym celu wystarczy użyć pustej instrukcji `raise`:

```
1 try:
2     wynik = 1 / 0
3 except ZeroDivisionError:
4     print('Blad obliczen, nie wiem co zrobic')
5     raise
```

Sekcja `finally`

Sekcja `finally` umieszczana jest na końcu wyrażenia `try` i zostanie ona wykonana zawsze bez względu na to czy wyjątek zostanie złapany czy nie lub czy został on w ogóle wygenerowany.

```
1 try:
2     wynik = 1 / 0
3 except ZeroDivisionError:
4     print('Blad dzielenia przez zero')
5 finally:
6     print('Koniec obliczen')
```

Definiowanie wyjątków

Definiowanie własnych wyjątków polega na tworzeniu klas dziedziczących po wbudowanym wyjątku `Exception`. Są to zwykłe klasy, które mogą posiadać własne atrybuty i metody.

Wyjątki najczęściej nie są jednak skomplikowanymi bytami dlatego częstą praktyką jest tworzenie pustych klas wyjątków z wykorzystaniem słowa kluczowego `pass`.

Poniższy przykład przedstawia utworzenie dwóch wyjątków: prostego wyjątku nie posiadającego żadnych dodatkowych argumentów oraz wyjątku bardziej rozbudowanego, zawierającego dwa argumenty.

```
1 class WlasnyWyjatek(Exception):
2     pass
```

```
3
4 class BlednyArgument(Exception):
5     def __init__(self, nazwa_argumentu, bledna_wartosc):
6         self.nazwa_argumentu = nazwa_argumentu
7         self.bledna_wartosc = bledna_wartosc
```

Wyjątki zdefiniowane samodzielnie generowane są tak samo jak wyjątki wbudowane - poprzez wykorzystanie instrukcji `raise` wraz z instancją wyjątku.

Wbudowane wyjątki

Język Python posiada wiele wbudowanych wyjątków generowanych przez interpretera Pythona lub też bibliotekę standardową.

Najpopularniejsze z nich to:

- `KeyError` - brak klucza w słowniku
- `IndexError` - brak indeksu w kolekcji
- `SyntaxError` - błąd składniowy
- `NameError` - brak użytego identyfikatora (np. zmiennej)
- `AttributeError` - brak atrybutu w obiekcie

Organizacja kodu

Wraz z rozwojem projektu zawsze wzrasta liczba linii kodu naszego programu. Rozwijanie dużych aplikacji przy wykorzystaniu tylko jednego pliku z kodem źródłowym jest wręcz niemożliwe. W języku Python z pomocą przychodzą moduły oraz mechanizm importowania.

Moduły

Moduł jest zwykłym plikiem z kodem źródłowym o rozszerzeniu `*.py`. Zawiera on definicje klas i funkcji oraz inne dowolne instrukcje języka Python. Dzięki wyrażeniu `import` możliwe jest jego zaimportowanie w innym module, głównym pliku programu lub też sesji interaktywnej interpretera w celu wykorzystania zdefiniowanych w nim elementów. Dzięki takiemu rozwiązaniu możemy nasze aplikacje dzielić na mniejsze fragmenty.

W celu stworzenia modułu należy utworzyć plik o takiej samej nazwie jak nasz moduł z rozszerzeniem `*.py`.

```
1 def pole_trojkata(podstawa, wysokosc):
2     return 1 / 2 * podstawa * wysokosc
3
4 def pole_kwadratu(bok):
5     return bok * bok
```

Jeżeli powyższy przykład zostanie zapisany w pliku o nazwie `geometria.py` możliwe będzie zimportowanie modułu `geometria` oraz wykorzystanie zdefiniowanych w nim funkcji:

```
1 >>> import geometria
2 >>> geometria.pole_trojkata(4, 5)
3 10.0
```

Dostęp do elementów z danego modułu odbywa się poprzez użycie znaku kropki `.` tak jak ma to miejsce w przypadku atrybutów lub metod obiektów.

Instrukcja `import` posiada także inną formę wykorzystującą słowo kluczowe `from` - umożliwia ona zimportowanie wybranego elementu zdefiniowanego w zadanym module.

```
1 >>> from geometria import pole_trojkata
2 >>> pole_trojkata(4, 5)
3 10.0
```

Jeżeli konieczne jest zimportowanie większej liczby elementów z jednego modułu, możemy dostarczyć instrukcji `import` wiele nazw rozdzielając je znakiem przecinka `,`:

```
1 >>> from geometria import pole_trojkata, pole_kwadratu
```

Istnieje także możliwość zimportowania wszystkich symboli z modułu - w tym celu wykorzystywany jest symbol `*`.

```
1 >>> from geometria import *
2 >>> pole_trojkata(4, 5)
3 10.0
```

Korzystanie z tej formy instrukcji `import` nie jest jednak dobrą praktyką - napotykając taką instrukcję w kodzie nie wiemy dokładnie jakie symbole są do niego importowane. Takie podejście może prowadzić do bardzo trudno wykrywalnych błędów.

Poszukiwanie modułów

W celu zlokalizowania modułu interpreter języka Python przeszukuje katalogi zdefiniowane w liście `path` modułu `sys`. Lista ta jest inicjalizowana zawartością zmiennej środowiskowej `PYTHONPATH` oraz ścieżkami zależnymi od sposobu instalacji interpretera w danym systemie operacyjnym. Na liście tej zawsze znajduje się katalog wewnętrz którego uruchomiono interpreter.

Listę tę można modyfikować podczas działania programu w celu rozszerzenia listy katalogów w których interpreter ma wyszukiwać moduły.

Zależności cykliczne

Dwa moduły nie mogą importować się wzajemnie - zarówno bezpośrednio (moduł A importuje moduł B, moduł B importuje moduł A) jak i pośrednio (moduł A importuje moduł C, który importuje moduł B i odwrotnie). Powstanie takiej cyklicznej zależności powoduje wygenerowanie wyjątku podczas importowania. Problem cyklicznej zależności świadczy najczęściej o złym podziale kodu na moduły.

Pakiety

Pakiety służą do grupowania wielu modułów pod jedną nazwą. Rozwiązywanie takie umożliwia nam wprowadzenie czytelniejszej struktury podziału naszego projektu oraz zapobiega problemowi kolizji nazw z innymi modułami w przypadku wykorzystania powtarzającej się nazwy modułu.

W Pythonie pakietem nazywamy katalog zawierający plik `__init__.py` oraz ewentualnie inne moduły i zagnieżdżone pakiety.

Poniższa struktura katalogów obrazuje przykładową hierarchię pakietów i modułów w bibliotece zajmującej się obliczeniami matematycznymi:

```
1 matematyka/
2 |-- __init__.py
3 |-- algebra/
4 |   |-- __init__.py
5 |   |-- macierze.py
6 |   |-- wykresy.py
7 |   \-- ...
8 |-- analiza/
9 |   |-- __init__.py
10 |   |-- calki.py
11 |   |-- wykresy.py
12 |   \-- ...
13 |-- statystyka/
14 |   |-- __init__.py
15 |   |-- wariancje.py
16 |   \-- ...
17 \-- pierwiastek.py
```

Struktura ta składa się z głównego pakietu `matematyka` w którego skład wchodzą 3 pakiety (`algebra`, `analiza`, `statystyka`) oraz jeden moduł `pierwiastek`. Wewnętrzne pakiety zawierają kolejne moduły (`macierze`, `calki`, `wykresy`, etc.) - nazwy niektórych z nich są takie same, ale nie jest to problemem, ponieważ umieszczone są one w osobnych pakietach.

W celu zimportowania modułu macierze z pakietu algebra, wchodzącego w skład pakietu matematyka, musimy podać pełną ścieżkę modułu, rozdzielając nazwy pakietów znakiem kropki:

```
1 import matematyka.algebra.macierze
```

Tej samej ścieżki musimy użyć jeżeli chcemy wykorzystać funkcję, klasę czy też zmienną zdefiniowaną w importowanym module:

```
1 m = matematyka.algebra.macierze.Macierz()
```

Alternatywną metodą importowania modułu z pakietu jest wykorzystanie składni `from ... import ...`:

```
1 from matematyka.algebra import macierze
```

W takim wypadku dostęp do elementów zdefiniowanych w module macierze wymaga podania tylko jego nazwy, a nie pełnej ścieżki:

```
1 m = macierze.Macierz()
```

Tak samo jak w przypadku prostych modułów, możliwe jest także zimportowanie jedynie wybranego elementu z wskazanego modułu:

```
1 from matematyka.algebra.macierze import Macierz
```

Importowanie względne

W przypadku gdy jeden z modułów pakietu potrzebuje zależności znajdującej się w innym module wchodzącym w skład tego samego pakietu (bez względu na poziom zagnieżdżenia), możliwe jest jego zimportowanie przy użyciu ścieżki względnej. W tym celu zamiast nazwy pakietu wykorzystywany jest znak kropki. Pojedyncza kropka oznacza ten sam pakiet, dwie kropki - pakiet o jeden poziom wyżej w hierarchii, trzy kropki - pakiet o dwa poziomy wyżej, itd.

Jeżeli w module matematyka.algebra.wykresy konieczne jest wykorzystanie modułu matematyka.algebra.macierze, to możemy go zimportować w następujący sposób:

```
1 from . import macierze
```

Jeżeli moduł matematyka.algebra.wykresy potrzebowałby skorzystać z modułu matematyka.pierwiastek musielibyśmy skorzystać z dwóch znaków kropki, oznaczających odniesienie do pakietu o jeden poziom wyżej:

```
1 from .. import pierwiastek
```

Po znakach kropki może wystąpić także nazwa pakietu. W przypadku konieczności wykorzystania modułu matematyka.algebra.macierze w module matematyka.analiza.calki możemy wykorzystać następującą instrukcję `import`:

```
1 from ..algebra import macierze
```

Elementy komunikacji w sieci internetowej

Adres URL

Wiesz już mniej więcej jak działa Internet. Zanim zaczniemy właściwą część zastanówmy się jeszcze przez chwilę co to jest adres URL i jak jest zbudowany. Aplikacje internetowe opierają się w dużej mierze o takie adresy. Usystematyzujmy wiedzę o tym elemencie. **URL** (Uniform Resource Locator) to standard opisujący ujednolicony format adresowania, czyli określenia lokalizacji pewnych zasobów. Jest on stosowany w sieciach komputerowych, w szczególności w Internecie. Na ogół kojarzymy go ze stronami WWW, ale przy pomocy URL adresujemy wszelkie zasoby dostępne w Internecie. Budowę URL najogólniej opisać można następująco:

```
1 <schemat>:<ścieżka hierarchiczna>
```

Schematy są różne, bardzo często są to nazwy protokołów np: `http`, `https`, `ftp`, `telnet`, `mailto`, `file`. W przeglądarkach na ogół są to `http` i `https`. Czasem, gdy chcemy w przeglądarce otworzyć plik z dysku może to być też `file`. Po schemacie na ogół występuje host i ścieżka dostępu. Przykład takiego URL:

```
1 https://www.alx.pl/tech/python/
2 \_\_ / \_\_\_\_\_\_ / \_\_\_\_\_\_
3 | | | | | | | |
4 protokół host sz cieka do zasobu
```

Bardziej złożony URL może zawierać jeszcze takie elementy jak port, zapytanie (query string), czy odniesienie do fragmentu dokumentu:

```
1 http://www.costam.pl:5555/kat/plik?par1=wartosc1&par2=wartosc2#fragment
2 \_\_ / \_\_\_\_\_\_ / \_\_ / \_\_\_\_\_\_ / \_\_\_\_\_\_ / \_\_\_\_\_\_
3 | | | | | | | |
4 schemat host port sz cieka zapytanie
      fragment
5 protokół nazwa, do pliku (query string)
6 lub adres (pali)
7 ip
```

Aplikacje mogą być udostępniane na różnych portach. Dzięki temu na jednym serwerze może być

udostępnionych wiele aplikacji. Wiele z nich ma swoje domyślne porty. Np. serwer bazy danych PostgreSQL domyślnie udostępnia swoje usługi na porcie 5432. Serwer deweloperski Django domyślnie uruchamia się na porcie 8000, Flask na porcie 5000.

W literaturze można też spotkać określenie URI (Uniform Resource Identifier). Jest to pewne uogólnienie URL. Obejmuje ono także URN (Uniform Resource Name), czyli ujednolicony format nazw. Przykładem takiego URN może być np. uuid, czy nr ISSN (International Standard Serial Number) - ośmio cyfrowy, unikalny identyfikator wydawnictw ciągłych. Dalej możesz spotkać się z tym, że określenia URI i URL mogą być stosowane zamiennie. Jeszcze kilka praktycznych informacji. Bardziej dla przyomnienia. Lokalny komputer ma specjalny adres IP: 127.0.0.1 - stosowana jest też nazwa hosta: localhost. Jeśli uruchomimy na naszym komputerze jakieś aplikacje - np. napisaną we Flask i udostępnioną na porcie 5000 i napisaną w Django - udostępnioną na porcie 8000, to mogą one być dostępne jednocześnie pod takimi adresami:

- Django:
 - <http://localhost:8000>

```
1 http://127.0.0.1:8000
```

- Flask:
 - <http://localhost:5000>
 - <http://127.0.0.1:5000>

Jeśli port nie jest podany, to w przeglądarce przyjmuje domyślną wartość: 80. Większość stron internetowych wystawiana jest właśnie na tym porcie.

Protokół HTTP

Komunikacja komputerów to nie jest wesoła pogawędka przekupek na targu. Oj nie. Przypomina to raczej sztywną, wysoce sformalizowaną komunikację międzynarodową. Nie można ot tak zadzwonić do prezydenta jakiegoś kraju i sobie z nim coś uzgadniać. W takich sytuacjach niezbędne jest stosowanie się do różnego rodzaju protokołów, czyli reguł zachowania, komunikacji. Tak właśnie jest w przypadku komputerów. Jest tam cała masa różnego rodzaju protokołów. Każdy z nich ma jakieś swoje zastosowanie. Często też protokoły świadczą sobie wzajemnie jakieś usługi.

Wróćmy do tych dyplomatycznych protokołów. Możemy sobie wyobrazić taką sytuację, w której jedna ważna osoba - nazwijmy ją Pi musi porozmawiać z drugą ważną osobą - Fi. Pi i Fi stoją na szczytach ważnych organizacji i mają bardzo zapełnione kalendarze. Dbaniem o kalendarz zajmują się ich sekretarki. Pi zleca więc sekretarce - w ich własnym protokole zadanie ustalenia rozmowy z Fi. Sekretarka

Pi komunikuje się w tym celu z sekretarki Fi i uzgadniają dogodny dla Pi i Fi termin rozmowy. Ustawiają spotkanie w kalendarzach a gdy przychodzi na nie pora zestawiają połączenie i udostępniają je Pi i Fi. Zobaczmy pewne analogie w świecie komputerów.

Aplikacje komputerowe często komunikują się ze sobą na przy pomocy specjalnego protokołu - HTTP (Hypertext Transfer Protocol) lub jego szyfrowaną odmianą HTTPS (Hypertext Transfer Protocol Secure). Przykładem takich aplikacji mogą być np. przeglądarka internetowa oraz serwer HTTP. To właśnie w przeglądarce mogłeś/mogłaś zetknąć się już z nazwą wspomnianych protokołów. Zaczynają się od nich - nieprzypadkowo - adresy stron internetowych, np: <https://alx.com>. Po wpisaniu takiego adresu przeglądarka wysyła do serwera zapytanie/żądanie (czyli znany nam już request). Serwer procesuje takie requesty na podstawie zdefiniowanych w nim reguł i przygotowuje odpowiedź (response). Może to być statyczna strona HTML, inny statyczny plik - np. obrazek. Może to też być jakaś odpowiedź przygotowana przez inną aplikację. Wtedy serwer HTTP przesyła taki request do tej aplikacji. Aplikacja zwraca odpowiedź, które przekazuje dalej serwer. Komunikacja z takim serwerem jak już zapewne się domyślasz odbywa się przy pomocy protokołu HTTP. Zanim poznamy go lepiej określmy dokładniej jego miejsce w kontekście komunikacji sieciowej. W komunikacji tej używanych jest bowiem wiele różnych protokołów. Jak wspomniałem jedne protokoły korzystają z usług świadczonych przez inne. I tak np. w HTTPS dane szyfrowane są przy pomocy protokołu - TLS (Transport Layer Security). Protokół TCP (Transmission Control Protocol) korzysta natomiast z usług protokołu IP (Internet Protocol).

W procesie komunikacji między komputerami protokołów jest znacznie więcej. Ponadto rozróżnia się też różne warstwy. Jest to tzw. model OSI. I tak np. HTTP jest protokołem w warstwie aplikacji tego modelu, TLS to protokół używany w warstwie prezentacji. TCP jest protokołem z warstwy transportowej a IP jest protokołem komunikacyjnym warstwy sieciowej. Takich warstw jest więcej. Model OSI (Open Systems Interconnection), który jest standardem opisującym strukturę komunikacji w sieci komputerowej wyróżnia ich 7. Poniżej je przedstawiamy. W nawiasach podano przykłady protokołów, standardów i urządzeń, które należą do danej warstwy.

7 - warstwa aplikacji (np. HTTP, FTP, SMTP, DNS, DHCP, ...) 6 - warstwa prezentacji (np. TLS, SSL, MIME, ...) 5 - warstwa sesji (PPTP, NetBIOS, L2TP, ...) 4 - warstwa transportowa (TCP, UDP, SCTP, ...) 3 - warstwa sieciowa (IP, AppleTalk, router) 2 - warstwa łącza danych (SDLC, PPP, switch) 1 - warstwa fizyczna (RS-232, Bluetooth)

Dane wędrując z jednego komputera na drugi muszą odbyć najpierw podróż poprzez te warstwy w dół zmieniając przy okazji swój format w procesie nazywanym enkapsulacją. Czyli dane przechodzą w takim modelu drogą od góry (od aplikacji) w dół do warstwy fizycznej. Kiedy nasz komputer dostaje dane przechodzą one ten cały proces w drugą stronę. To jak z naszym dyplomatycznym protokołem - żądanie wychodzi od szefa, wędruje w dół do sekretarki, od niej do telefonistki, potem jest warstwa fizyczna - czyli linia telefoniczna, na której drugim końcu jest kolejna telefonistka, która przekaże

sprawy sekretarce itd itd. Jest to wielkie uproszczenie, ale pozwala zrozumieć pewną naturę tego procesu. W szczegółach - w każdej z warstw czeka na nas wiele problemów do rozwiązania. Jest to skomplikowany proces. Szczęśliwie jednak na co dzień nie musimy się tym przejmować, ponieważ proces ten może być dla nas niewidoczny. Wystarczy, że wpiszemy odpowiedni adres i przekażemy tam odpowiednie dane.. coś do nas wróci. Gorzej gdy nie wraca :)

Wróćmy do protokołu HTTP.

Obecnie najpowszechniej używane są wersją protokołu:

- HTTP/1.1 opisana w dokumencie <https://tools.ietf.org/html/rfc7230>
- HTTP/2 - <https://tools.ietf.org/html/rfc7540>

Trwają też prace nad wersją trzecią.

Główne składowe tego protokołu to: metody, nagłówki oraz kody HTTP. Co ważne jest to protokół bezstanowy. Oznacza to, że kolejne zapytania i odpowiedzi nie wiedzą nic o poprzednich. Są więc traktowane jako niezależne i jako takie powinny zawierać komplet informacji potrzebnych do ich zrealizowania. Pewnym obejściem tej bezstanowości są ciasteczka (cookies), czyli małe fragmenty tekstu, w których np. mogą być przechowywane dane sesji po zalogowaniu się użytkownika do aplikacji - tak by aplikacja wiedziała, że użytkownik jest zalogowany. Mogą to też być w zasadzie dowolne inne dane. Mechanizmem cookies nie będziemy się tu jednak zajmować.

Komunikacja w protokole odbywa na zasadzie wymiany żądań i odpowiedzi. Zbudowane są one według pewnego schematu. Najogólniej mówiąc żądania i odpowiedzi składają się mogą z 3 części:

NAZWA	OPIS
HTTP Header	Nagłówek a właściwie całą sekcja nagłówków. Sekcja obowiązkowa. Obowiązkowo musi tu pojawić informacja o metodzie.
Body	Ciało. W przypadku niektórych metod wymagane, dla innych nie. Jeśli np. na dane żądanie zostanie zwrócony dokument html z treścią strony to będzie właśnie tutaj.
Trailer	Dodatkowe nagłówki - przydatne np. dla zachowania integralności danych. Tutaj też jest miejsce np. na podpis cyfrowy. Prezentowany tu też może być status przetwarzania końcowego

Jak wygląda obsługa żądania?

W pierwszym kroku wyszukiwany jest adres serwera. Klient wysyła żądanie (DNS Query) do serwera DNS (Domain Name System), by ustalić adres IP serwera. Serwery DNS mogą być różne. Mogą

pochodzić od dostawcy internetu, ale może to być też ogólnodostępna usługa - jak ta spod adresu 8.8.8.8, którą udostępnia Google. Tłumaczą one nazwy domen - np. witryna <https://www.nasa.gov> w momencie pisania tego tutoriala kryje się pod adresem IP 52.0.14.116. Przekonaj się czy tak jeszcze jest klikając w link: <https://52.0.14.116>. Adres IP to taki identyfikator sieci komputerowej w protokole IP. Czasem wewnątrz takiej sieci są jeszcze kolejne sieci.. więc tak naprawdę pakiety które sobie między tymi sieciami krążą są opakowywane w różne adresy (porównaj sobie z nagraniem: https://youtu.be/7_LPdttKXPc). Następnie ustanawiane jest połączenie TCP. Jeśli się powiedzie wysyłane jest żądanie HTTP. Potem następuje oczekiwania na odpowiedź. Odebranie pakietów i zakończenie połączenia TCP. Proste jak droga do pracy z "Nie lubię poniedziałku" - <https://youtu.be/xHCSf3ps03g>.

W dalszej części omówimy metody, nagłówki oraz kody HTML. Jak zobaczymy później znajomość metod przyda nam się przy obsłudze formularzy a jeszcze bardziej przy budowaniu RESTowego API, w którym istotną rolę odgrywają także inne elementy protokołu.

Metody

Protokół HTML udostępnia szereg metod, które określają rodzaj pożądanej akcji dla danego zasobu. Korzystaliśmy już z najpopularniejsze metody: GET, używana jest wtedy kiedy wchodzimy na jakąś stronę internetową poprzez wpisanie jej adresu lub kliknięcie w link w przeglądarce. Zróbcmy krótki przegląd takich metod i ich zastosowań.

Metoda	Zastosowanie
GET	Jest to żądanie reprezentacji określonego zasobu. Np. konkretnej strony internetowej. Konkretnego artykułu, czy listy przedmiotów. Choć da się w ten sposób przekazywać dane do serwera, to jednak nie powinniśmy w ten sposób tworzyć nowych zasobów.
HEAD	Metoda działa podobnie jak GET. W odpowiedzi nie jest jednak zwracane ciało a jedynie same nagłówki.
POST	Metoda ta służy do przesyłania danych do serwera. Przy jej pomocy możemy tworzyć, czy modyfikować istniejące zasoby. Metoda ta może więc powodować zmianę stanu serwera.
PUT	Metoda ta nadpisuje istniejący zasób nowymi danymi.
DELETE	Metoda służy do usuwania określonego zasobu.
CONNECT	Metoda ustanawia tunel do serwera identyfikowanego przez zasób docelowy

Metoda	Zastosowanie
PATCH	Metoda nadpisuje wybrane elementy określonego zasobu. Jest to w odróżnieniu od PUT częściowa modyfikacja zasobu
OPTIONS	Metoda opisuje komunikację z docelowym zasobem. Np. zwraca możliwe do wykonania metody
TRACE	Metoda wykonuje test pętli zwrotnej wiadomości wzdłuż ścieżki do docelowego zasobu

Najczęściej wykorzystywane metody to GET i POST. W przypadku RESTowych API także PUT, DELETE, PATCH. Dlatego warto zapoznać się szczegółowo z ich opisem. Np. w tym zasobie: <https://tools.ietf.org/html/rfc7231#section-4>.

Popatrzmy jeszcze trochę na to co dzieje się w przeglądarce. Żądanie pobrania strony najczęściej wywołuje cały szereg różnych zapytań, które odbywają się w tle. Np. pobranie plików szablonów stylów CSS, skryptów JavaScript, obrazków, czy różnych danych. Możemy się o tym przekonać włączając specjalne narzędzie w przeglądarce. Wpisz w niej adres np.: <https://www.w3resource.com/>. Po załadowaniu strony kliknij prawym przyciskiem myszy i wybierz "Zbadaj element". Może to być też kombinacja klawiszy Ctrl+Shift+E, lub wybór z menu: Narzędzia -> Dla twórców witryn -> Sieć. Kroki te wykonuję w przeglądarce Mozilla Firefox, w innych przeglądarkach powinny być dostępne analogiczne opcje. Możesz tu znaleźć bardzo pomocne narzędzia. Warto się z nimi zapoznać. W dolnej części ekranu (czasem gdzieś z boku) powinien pojawić się specjalny panel z zakładkami. Przejdz do zakładki "Sieć". Teraz możesz odświeżyć stronę (możesz nacisnąć F5, albo odpowiedni przycisk w przeglądarce) Powinieneś zobaczyć widok podobny do poniższego:

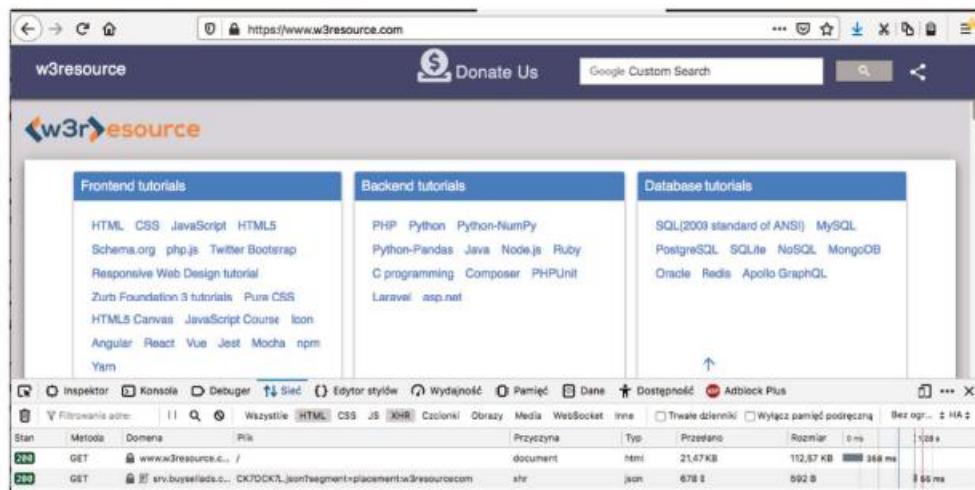


Figure 0.4: img

Na dole screenu widać kolumny. Pierwsza z nich licząc od lewej to "Stan". Widzimy tam kod HTML - to te 200 na zielonym tle. O kodach parę słów napiszemy później. Następna kolumna to "Metoda". W tym przypadku widzimy dwa zapytania typu GET. Dalej jest adres url zasobu. Widać tam jeszcze kilka innych interesujących informacji. Np. kolumna "Typ" pokazuje jakiego rodzaju dokument został tu zwrócony. Jeden z nich to dokument HTML, drugi jest w formacie JSON - spotkamy się z nim jeszcze w dalszej części.

Wspomnieliśmy już sobie o tym, że GET nie zawsze się sprawdza. Skoro przez wpisanie adresu w przeglądarce robimy zapytanie GET, to jak mogę zrobić zapytanie inną metodą? Jedną z możliwości jest użycie opcji przeglądarki. Jeśli na liście klikniemy interesujące nas zapytanie, to po prawej stronie (na ogół) powinien nam się pokazać panel z jego szczegółami. Jedną z dostępnych opcji będzie tam Edytuj i wyślij ponownie. Można tam wpisać też nazwę metody. Nie jest to jednak zbyt wygodny sposób. Zdecydowanie lepiej byłoby skorzystać z jakichś narzędzi dedykowanych do tego typu zadań. Jednym z takich narzędzi - dostępnym dla większości systemów operacyjnych jest Postman API Client (<https://www.postman.com/product/api-client>). W narzędziu tym można tworzyć nowe zapytania wpisując w odpowiednie pola metodę, adres, czy nagłówki.

W profesjonalnej wersji Pycharma dostępny jest klient http (<https://www.jetbrains.com/help/pycharm/http-client-in-product-code-editor.html>). Także w Visual Studio Code jako dodatek mamy REST Client (<https://marketplace.visualstudio.com/items?itemName=humao.rest-client>). Pozwalają one na tworzenie kolekcji zapytań w formie dokumentów. W takim dokumencie możemy mieć przygotowanych wiele różnych requestów. Oddzielam je od siebie tak jak w poniższym przykładzie, w

którym pokazano przykładowe zapytania GET i POST.

```
1 GET http://wp.pl
2
3 ####
4
5 GET http://localhost:5000/upper/?text=ala ma kota
6
7 ####
8
9 HEAD http://api.nbp.pl/api/exchangerates/tables/C
10
11 ####
12
13 OPTIONS http://api.nbp.pl/api/exchangerates/tables/C
14
15 ####
16
17 GET /api/exchangerates/tables/C HTTP/1.1
18 Host: api.nbp.pl
19 Content-Type: application/xml
20
21 ####
22
23 GET http://api.nbp.pl/api/exchangerates/tables/C HTTP/1.1
24 Host: api.nbp.pl
25 Content-Type: application/json
26
27 ####
28
29 POST /todos/ HTTP/1.1
30 Host: localhost:8000
31 User-Agent: Mozilla/5.0
32 Content-Length: 30
33 Content-Type: application/x-www-form-urlencoded
34
35 title=julius&description=cezar&done=false
```

Jak widać na tym etapie najczęściej wystarczy wpisać nazwę metody - w tym przypadku GET a po nim adres URL. W praktyce zapytanie zawiera często więcej informacji. Dodaje się tam specjalne nagłówki. Przy niektórych metodach wstawia się też sekcję body zawierającą dane do przesłania. Powyżej jest to fragment:

```
1 title=julius&description=cezar&done=false
```

Z podobnych części zbudowane są odpowiedzi serwera:

```
1 GET http://localhost:5000/upper/?text=ala+ma+kota
2
3 HTTP/1.0 200 OK
```

```

4 Content-Type: text/html; charset=utf-8
5 Content-Length: 11
6 Server: Werkzeug/1.0.1 Python/3.7.4
7 Date: Fri, 10 Apr 2020 12:13:23 GMT
8
9 ALA MA KOTA
10
11 Response code: 200 (OK); Time: 19ms; Content length: 11 bytes

```

W pierwszym wierszu widzimy metodę i URL. Jak widać znaki spacji w query string zmienione zostały na symbol +. Dalej mamy szereg metadanych - nagłówków. Jest tam wskazana wersja protokołu, status odpowiedzi, jej długość, czy typ. Następnie jest część zawierająca dane - to tzw. ciało odpowiedzi (ang. *body*) oraz wiersz z podsumowaniem, w którym jeszcze raz widać status odpowiedzi (Response code), czas wykonania i objętość ciała odpowiedzi - 11 bajtów - czyli po jednym bajcie na każdy znak w 'ALA MA KOTA'. Porównajmy to jeszcze z tym co można zobaczyć w przeglądarce:

The screenshot shows the Network tab of a browser's developer tools. A single request is listed:

- URL:** localhost:5000/upper/?text=ala+ma+kota
- Metoda żądania:** GET
- Zdany adres:** 127.0.0.1:5000
- Kod statusu:** 200 OK
- Wersja:** HTTP/1.1

The response details show:

- Nagłówki odpowiedzi:** (153 B)
- HTTP/1.1 200 OK**
- Content-Type:** text/html; charset=utf-8
- Content-Length:** 11
- Server:** Werkzeug/1.0.1 Python/3.7.4
- Date:** Fri, 10 Apr 2020 12:39:04 GMT

The response body is displayed as "ALA MA KOTA".

Figure 0.5: img

Jak widzimy nagłówki odpowiedzi są w zasadzie identyczne. Dość duże różnice są w nagłówkach żądania. Są to rzeczy dodawane przez przeglądarkę

Narzędzie dostarczone z PyCharm dostarcza jeszcze wielu innych funkcjonalności. Można tam wprowadzać różne zmienne, parametry, np. po to, by testować je na różnych środowiskach. My skupimy się na najprostszych użyciach. W dalszych przykładach - szczególnie w części poświęconej REST będą pokazywane przykłady zapytań, które można w takim kliencie użyć

Kody HTML

W poprzednich przykładach widzieliśmy, że serwer zwracając odpowiedź umieszczał w niej między innymi liczbowy kod. To kod odpowiedzi. Te kody mają w protokole swoje znaczenie. Przydają się też wtedy, kiedy piszemy aplikacje, które sięgają po jakieś dane w Internecie. Taki kod może zaoszczędzić nam sporo pracy i warto wiedzieć co się pod nim kryje. Na pewno zdarzyło Ci się nieraz widzieć na stronie komunikat HTTP 404 Not Found. Często ma on różne zabawne formy czy grafiki. W większości projektów jest przygotowana specjalna strona HTML, która jest przygotowana właśnie na potrzeby obsługi tego kodu. Oznacza on, że strona po którą sięgasz nie istnieje. Może to być np. jakiś artykuł, czy film, który ktoś usunął ze swojej aplikacji. A może po prostu jakaś literówka. Koniec końców serwer nie umie znaleźć takiego zasoby i dlatego wyświetla taką informację. Innym częstym kodem jest HTTP 500 - oznacza on, że wystąpił jakiś błąd aplikacji. Pisząc nasze aplikacje nie przejmujemy się specjalnie HTTP 404. Ale 500-ka może spędzać sen z powiek. Takich kodów jest bardzo dużo. Można podzielić je na kilka grup, które tu poniżej opiszymy.

W protokole pierwszą informacją w odpowiedzi jest wersja protokołu HTTP, po niej następuje kod odpowiedzi, po nim zaś słowny opis tego kodu. Np:

```
1 HTTP/1.1 400 Bad Request
```

Kody HTTP można pogrupować w kilka najważniejszych kategorii. Odróżnia je od siebie pierwsza cyfra.

Kategoria	Info	Opis
1xx	Informational	Komunikaty dotyczące transferu.
2xx	Success	Informuje o tym, że żądanie klienta zostało obsłużone poprawnie
3xx	Redirection	Informuje o tym, że zostało zrobione przekierowanie na inny zasób.

Kategoria	Info	Opis
4xx	Client Error	Informuje o błędach spowodowanych przez użytkownika. Tzn. aplikacja działa poprawnie, ale klient zgłosił nieprawidłowe żądanie. Np. sięga po nie istniejący zasób (404), albo nie ma uprawnień (403), i wiele innych
5xx	Server Error	Informuje o błędach po stronie serwera. Np. ktoś nie obsługuje dzielenia przez zero. Albo wysyła w formularzu jakiś napis zamiast liczby i działanie nie może być wykonane.

Najczęściej używane kody to te z grupy Success, Client Error i Server Error, można by wymienić takie jak 200, 201, 202, 403, 404, 500. I tak np. kod 200 oznacza, że zasób został poprawnie zwrócony. Kod 201 powinno zwrócić po poprawnym utworzeniu zasobu. Kod 202 lub 204 wtedy, gdy zasób usunęliśmy. Kod 404 - gdy takiego zasobu nie ma. Kod 403, gdy nie mamy do niego uprawnień itd itd. Kodów jest cała masa i można o nich poczytać w dokumentacji, np tutaj: <https://developer.mozilla.org/en-US/docs/Web/HTTP>Status>

Nagłówki

Nagłówki HTTP to różnego rodzaju komendy stosowane w komunikacji między klientem (np. przeglądarką) a serwerem HTTP. Jedynym wymaganym nagłówkiem w zapytaniu HTTP jest nagłówek Host

Choć wiele klientów pozwala na zapis taki jak poniżej

```
1 GET http://api.nbp.pl/api/exchangerates/tables/C
```

to jednak tłumaczony jest on potem na mniej więcej taki:

```
1 GET http://api.nbp.pl/api/exchangerates/tables/C HTTP/1.1
2 Host: api.nbp.pl
```

również poprawnym byłby zapis:

```
1 GET /api/exchangerates/tables/C HTTP/1.1
2 Host: api.nbp.pl
```

W przypadku gdyby nagłówek Host zawierał inny adres niż to jest w URL, pierwszeństwo ma URL.

Nagłówki pozwalają na dodanie do zapytania wielu dodatkowych informacji. Możemy tam określić np. jaki typ MIME dokumentu jest preferowane przez klienta:

```
1 Accept: application/xhtml+xml, application/xml;q=0.9, text/xml;q=0.7,  
text/html;q=0.5, text/plain;q=0.3
```

W powyższym przykładzie parametr q przyjmuje wartości od 0 do 1 z krokiem 0.1. Określa on wagę preferencji. Nie podany oznacza wagę 1. Widać więc, że najwyższy priorytet ma typ application/xhtml+xml a najniższy zwykły tekst (text/plain).

Inny nagłówek określać może preferowany język:

```
1 Accept-Language: pl, en-us;q=0.7
```

W naszych pracach przydatny może być nagłówek Content-Type. Spróbuj porównać wyniki zwracane przez te dwa zapytania:

```
1 GET http://api.nbp.pl/api/exchangerates/tables/C  
2 Content-Type: application/xml  
3  
4 GET http://api.nbp.pl/api/exchangerates/tables/C  
5 Content-Type: application/json
```

Jak widać serwer w zależności od tego nagłówka zwraca dane w innym formacie.

https://pl.wikipedia.org/wiki/Lista_nag%C5%82%C3%B3wk%C3%B3w_HTTP

Django

Wstęp

Django jest jednym z najpopularniejszych frameworków webowych opartych o język Python.

Czym właściwie jest framework? Możemy o nim pomyśleć jako o zestawie pewnych narzędzi. To jednak będzie odrobinę za mało. Taki zestaw narzędzi w świecie programistów to po prostu biblioteka. Framework rozszerza ten zestaw o pewną metodologię. Mówi nie tylko czego używać, ale też jak organizować kod, by tworzyć aplikacje w sposób wygodny i efektywny.

Django to pewien zestaw narzędzi, a także zestaw reguł służących do tworzenia aplikacji internetowych. Takie zestawy narzędzi i reguł często nazywamy też frameworkami. Dzięki zorganizowaniu takich narzędzi i reguł w całość możliwe jest szybsze i łatwiejsze tworzenie różnego rodzaju aplikacji. Frameworki webowe do których zaliczane jest Django pomagają w tworzeniu aplikacji internetowych - tzn takich z których korzystamy poprzez przeglądarki WWW.

Źródła Django można dopatrzeć się 2003 roku, kiedy to Adrian Holovaty oraz Simon Willison, pracownicy Lawrence Journal-World newspaper poszukując sposobów na optymalizację tworzenia

aplikacji internetowych, wybrali język Python jako odpowiednie narzędzie. Dostrzegli oni pewne ograniczenia w istniejących już wtedy narzędziach napisanych w Pythonie i postanowili stworzyć własne. Początkowo postrzegali to narzędzie jako swoisty CMS i tak go nazywali. Po odejściu Simona jego miejsce zajął Jacob Kaplan - Moss, który wraz z Adrianem kontynuował rozwój aplikacji. Przekonali oni swoich pracodawców (World Company) do tego, by uwolnić ten kod na zasadach open source. Wtedy też Adrian - wielki miłośnik wspaniałego jazzmana i gitarzysty Django Reihnhardta wybrał na jego część nazwę dla projektu.

<https://www.youtube.com/watch?v=a1j8VLPasO0>

Dalszy rozwój projektu doprowadził nas do miejsca, w którym Django jest jednym z najpopularniejszych narzędzi do tworzenia aplikacji internetowych - przynajmniej w świecie Pythona. Django posiada szerokie portfolio wdrożeń, opiera się na nim wiele znanych aplikacji internetowych, wśród nich warto wspomnieć o YouTube, Dropbox, Quora, Instagram, czy Spotify. Można o tym przeczytać np. pod tym linkiem:

<https://www.shuup.com/blog/25-of-the-most-popular-python-and-django-websites/>

Oczywiście w aplikacjach o tej skali Django nie jest jedynym narzędziem. Niemniej jednak znaczco wpływało na ich rozwój, co warto odnotować.

Frameworki często mają jakieś swoje marketingowe hasła. W przypadku Django jest to "The web framework for perfectionists with deadlines.". Dobrze ono oddaje naturę tego narzędzia. Uzbrojone jest ono bowiem szeroki zestaw funkcjonalności, które pozwalają na zbudowanie w pełni funkcjonalnej aplikacji w stosunkowo krótkim czasie.

Tworząc aplikacje, webowe spotykamy się z często powtarzanymi zadaniami. np.: z koniecznością uwierzytelniania użytkowników, tworzenia i obsługi formularzy, wyświetlania i modyfikacji treści na stronach html w oparciu o dane pobierane z bazy danych lub innych źródeł. Jedną z cech Django jest podejście "Battery Included" - sugerujące, że zestaw ten zawiera wszystko, czego potrzebujemy, by zbudować taką aplikację. I tak w istocie jest. Wszelkie wymienione wcześniej zadania i wiele innych możliwe są do realizacji przy użyciu komponentów oferowanych przez Django. Mało tego przez lata rozwoju projekt ten doczekał się ogromnej społeczności, która wytworzyła szereg dodatkowych narzędzi, które możemy we własnych projektach wykorzystać. Django jest też intensywnie rozwijane. Regularnie wychodzą nowe wersje, pojawiają się poprawki związane z bezpieczeństwem. To wszystko sprawia, że Django jest świetny wyborem zarówno jako narzędzie do tworzenia różnego rodzaju aplikacji internetowych, ale też jako narzędzie do rozwoju ścieżki zawodowej.

Kurs wymaga już pewnej wiedzy. W tym miejscu zakładam, że już trochę znasz Pythona. Umiesz też poruszać się w swoim systemie przy wykorzystaniu interfejsu tekstowego. Jeśli pracujesz na Windowsie, może to być Command Prompt czy Windows Power Shell, na macu, czy linuxie będzie to po prostu terminal. Moje przykłady będą robione właśnie w terminalu. Jest to najpowszechniej

stosowane w środowisku programistów narzędzie i warto się do niego przyzwyczajać. W systemie Windows jest szereg możliwości, by takie narzędzie pozyskać. Może to być np. cygwin, babun. Można też pokusić się o użycie Windows Subsystem Linux - czyli opcję zainstalowania jakiejś dystrybucji Linuxa wprost w Windowsie. Jeszcze inną opcją jest użycie narzędzi takich jak VirtualBox i postawienie wirtualnej maszyny z jakimś Linusem. Taka wiedza na pewno zaowocuje w przyszłości czymś dobrym.

Instalacja, konfiguracja

Django to projekt open source. Jego kod źródłowy można znaleźć w repozytorium znajdującym się pod adresem:

<https://github.com/django/django>

Aby zainstalować Django wystarczy mieć zainstalowanego pythona - najlepiej w najnowszej wersji (w chwili pisania tego dokumentu jest to wersja 3.7.1), a następnie instalację go poprzez manager pakietów pip. Wcześniej jednak zalecamy utworzenie wirtualnego środowiska dla przygotowywanego projektu.

```
1 $ cd ~
2 $ mkdir workspace
3 $ cd workspace/
4 $ mkdir kurs_django
5 $ cd kurs_django/
6 $ python -m venv .venv
7 $ ls -all
8 total 0
9 drwxr-xr-x 3 alx staff 96 27 maj 23:17 .
10 drwxr-xr-x 4 alx staff 128 27 maj 23:16 ..
11 drwxr-xr-x 6 alx staff 192 27 maj 23:17 .venv
12 $ source .venv/bin/activate
13 (.venv) $ pip install django
```

Możemy sprawdzić jaką wersję zainstalowaliśmy poleceniem:

```
1 (.venv) $ python -c "import django;print(django.__version__)"
2 3.0.6
```

Na potrzeby pierwszego projektu to powinno wystarczyć. Django przychodzi z własnym serwerem WWW, który umożliwia podgląd prac już w czasie developmentu

Virtualenvwrapper

Czasem wygodniej jest użyć nakładki na virtualenv. Może to być `virtualenvwrapper`, który trzyma wszystkie środowiska w jednym miejscu i pozwala łatwo się między nimi przekształcać.

<https://virtualenvwrapper.readthedocs.io/en/latest/>

```
1 $ pip install virtualenvwrapper
2 ...
3 $ export WORKON_HOME=~/Envs
4 $ mkdir -p $WORKON_HOME
5 $ source /usr/local/bin/virtualenvwrapper.sh
6 $ mkvirtualenv env1
7 Installing
8 setuptools.....
9 .....
10 .....
11 .....done.
12 virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env1/bin/
     predeactivate
13 virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env1/bin/
     postdeactivate
14 virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env1/bin/
     preactivate
15 virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env1/bin/
     postactivate New python executable in env1/bin/python
16 (env1)$ ls $WORKON_HOME
17 env1 hook.log
```

W przypadku tej nakładki ustawiamy zmienną środowiskową `WORKON_HOME` na folder, w którym chcemy trzymać nasze środowiska. Chcąc zmienić środowisko wpisujemy po prostu polecenie `workon <nazwa>`. Np: będąc w środowisku `env2`, by zacząć pracę w `env1` posłużę się następującymi komendami:

```
1 (env2)$ workon env1
2 (env1)$ echo $VIRTUAL_ENV
3 /Users/dhellmann/Envs/env1
4 (env1)$
```

Więcej informacji tutaj:

<https://virtualenvwrapper.readthedocs.io/en/latest/>

Tworzymy projekt

Struktura kodu tworzonego w Django zorganizowana w ramach **projektu**. Pojedynczym projektem jest na przykład strona szkolenia prowadzonego przez Internet. Projekt składa się z aplikacji. Ap-

likacją jest na przykład blog, forum dyskusyjne, galeria zdjęć, system do przeprowadzania ankiet. Każdy projekt może składać się z wielu aplikacji. Każda aplikacja tworzy w zasadzie osobny „byt” i jej kod może być użyty w innych projektach. Każdy projekt żyje w jakimś katalogu. Katalogu tego nie umieszczamy zwykle w miejscu serwowanym przez serwer WWW. Skoro jesteśmy trochę w tematyce jazzowej nazwijmy projekt jazzy. Będziemy przechowywać go w swoim katalogu domowym. W jego skład będzie wchodzić na razie tylko jedna aplikacja - prosty blog. Na blogu będzie można zamieszczać wpisy i oznaczać je tagami.

Żeby stworzyć nowy projekt, wydajemy polecenie:

```
1 $ django-admin startproject jazzy
```

To polecenie tworzy katalog jazzy, a w nim kilka plików potrzebnych do działania projektu.

Możemy teraz wystartować własny serwer WWW będący częścią Django. W tym celu wchodzimy do katalogu z projektem i wydajemy polecenie:

```
1 $ python manage.py runserver
```

I teraz możemy wejść przeglądarką pod adres: <http://localhost:8000> i przekonać się, że nasz projekt, choć pusty, jest dostępny.

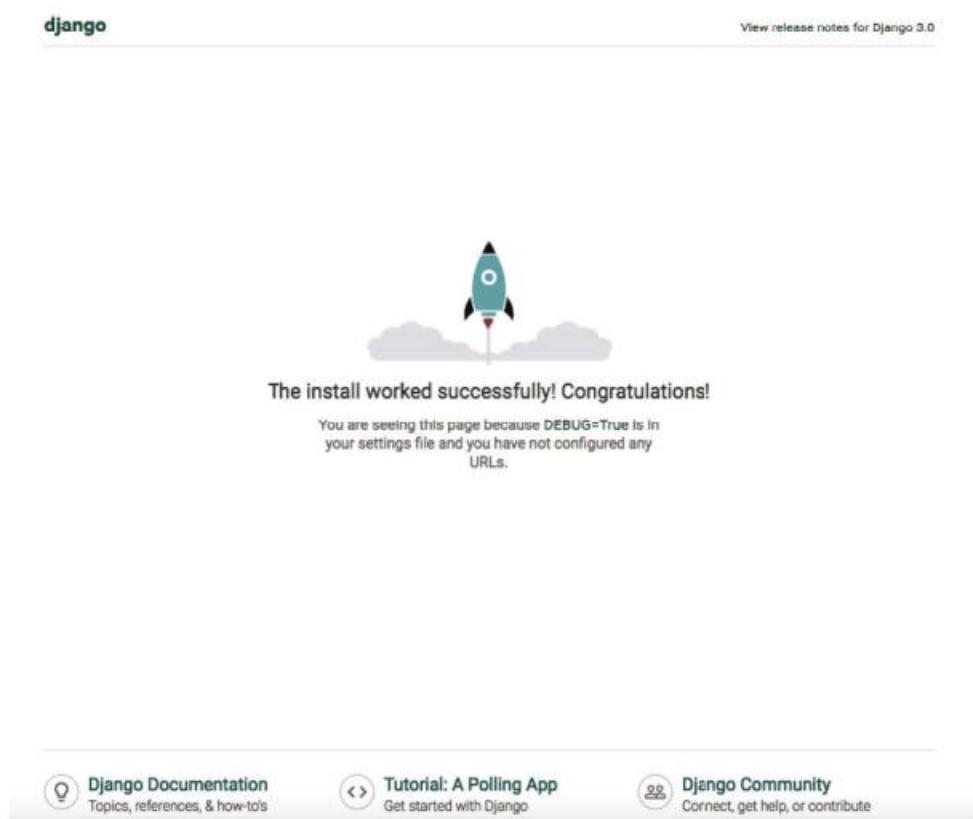


Figure 0.6: img

Django przychodzi do nas z systemem pomocy - jak większość porządkowych frameworków napisanych w Pythonie. Wywołanie samego polecenia `django-admin` pomoże nam znaleźć odpowiednią opcję jeśli ją zapomnimy:

```
1 (.venv) $ django-admin
2 Type 'django-admin help <subcommand>' for help on a specific subcommand
3 .
4 Available subcommands:
5 [django]
6   check
7   compilemessages
8   createtemplate
9   dbshell
```

```
10    diffsettings
11    dumpdata
12    flush
13    inspectdb
14    loaddata
15    makemessages
16    makemigrations
17    migrate
18    runserver
19    sendtestemail
20    shell
21    showmigrations
22    sqlflush
23    sqlmigrate
24    sqlsequencereset
25    squashmigrations
26    startapp
27    startproject
28    test
29    testserver
30 Note that only Django core commands are listed as settings are not
properly configured (error: Requested setting INSTALLED_APPS, but
settings are not configured. You must either define the environment
variable DJANGO_SETTINGS_MODULE or call settings.configure() before
accessing settings.).
```

Zgodnie z podpowiedzią, o zastosowaniu opcji możemy dowiedzieć się przy pomocy polecenia:

```
1 django-admin help <subcommand>
```

np:

```
1 (.venv) $ django-admin help startproject
```

Sprawdź samodzielnie wynik powyższego wywołania.

Po wykonaniu komendy `django-admin startproject jazzy` i przejściu do katalogu `jazzy` powiniśmy zobaczyć strukturę przypominającą tę:

```
1 $ tree
2 .
3   jazzy
4     __init__.py
5     settings.py
6     urls.py
7     wsgi.py
8     db.sqlite3
9     manage.py
```

Mamy więc główny katalog jazzy a w nim katalog o tej samej nazwie i plik manage.py. Pierwsze jazzy to główny katalog naszego projektu. Ten wewnętrzny zawiera podstawowe pliki naszego projektu. Na tym etapie w naszym projekcie nie ma jeszcze aplikacji.

Pliki projektu

manage.py

W głównym katalog jazzy znajdziemy plik manage.py. Plik ten to narzędzie, które pozwala nam sterować wieloma operacjami w naszym projekcie przy pomocy linii poleceń. To bardzo ważny plik. Możesz zerknąć w jego zawartość, ale jej nie modyfikuj. Będziemy korzystać z niego wielokrotnie. By dowiedzieć się co potrafi możesz wywołać w terminalu ten skrypt tak jak poniżej. Jest tu dużo podobieństw do django-admin prawda? Nie jest to przypadkowe.

```
1 (.venv) $ python manage.py
2
3 Type 'manage.py help <subcommand>' for help on a specific subcommand.
4
5 Available subcommands:
6
7 [auth]
8     changepassword
9     createsuperuser
10
11 [contenttypes]
12     remove_stale_contenttypes
13
14 [django]
15     check
16     compilemessages
17     createcachetable
18     dbshell
19     diffsettings
20     dumpdata
21     flush
22     inspectdb
23     loaddata
24     makemessages
25     makemigrations
26     migrate
27     sendtestemail
28     shell
29     showmigrations
30     sqlflush
31     sqlmigrate
32     sqlsequencereset
33     squashmigrations
```

```
34     startapp
35     startproject
36     test
37     testserver
38
39 [sessions]
40     clearsessions
41
42 [staticfiles]
43     collectstatic
44     findstatic
45     runserver
```

O django-admin i manage.py więcej można dowiedzieć się w dokumentacji:

<https://docs.djangoproject.com/en/3.2/ref/django-admin/>

Jaką rolę spełniają pozostałe pliki?

- jazzy/**init.py** - pusty plik, który mówi Pythonowi o tym, że katalog, w którym się znajduje powinien być potraktowany jako paczka - czyli pythonowy pakiet. O paczkach możesz przeczytać w oficjalnej dokumentacji Pythona <https://docs.python.org/3/tutorial/modules.html#tut-packages>
- jazzy/settings.py - Ustawienia i konfiguracja projektu. To bardzo ważny plik, w którym powinny znaleźć się wszelkie ustawienia konfiguracyjne. W miarę rozbudowy projektu będziemy dodawać tutaj nowe rzeczy.
- jazzy/urls.py - ten plik to swoisty spis treści stron dostępnych w naszej aplikacji. Odpowiada on za tzw. routing. Czyli przekierowanie odpowiednich żądań do odpowiednich funkcji, które je obsługują.
- jazzy/wsgi.py - ten plik odpowiada za współpracę z serwerami WWW, które obsługują standard WSGI. Przyjrzymy się temu podczas opisywania deploymentu
- jazzy/asgi.py - podobnie jak wyżej, tyle że obsługuje inny standard - ASGI to Asynchronous Server Gateway Interface. Obsługuje on żądania asynchroniczne. W ramach tego kursu nie będziemy się jednak szerzej tym tematem zajmować

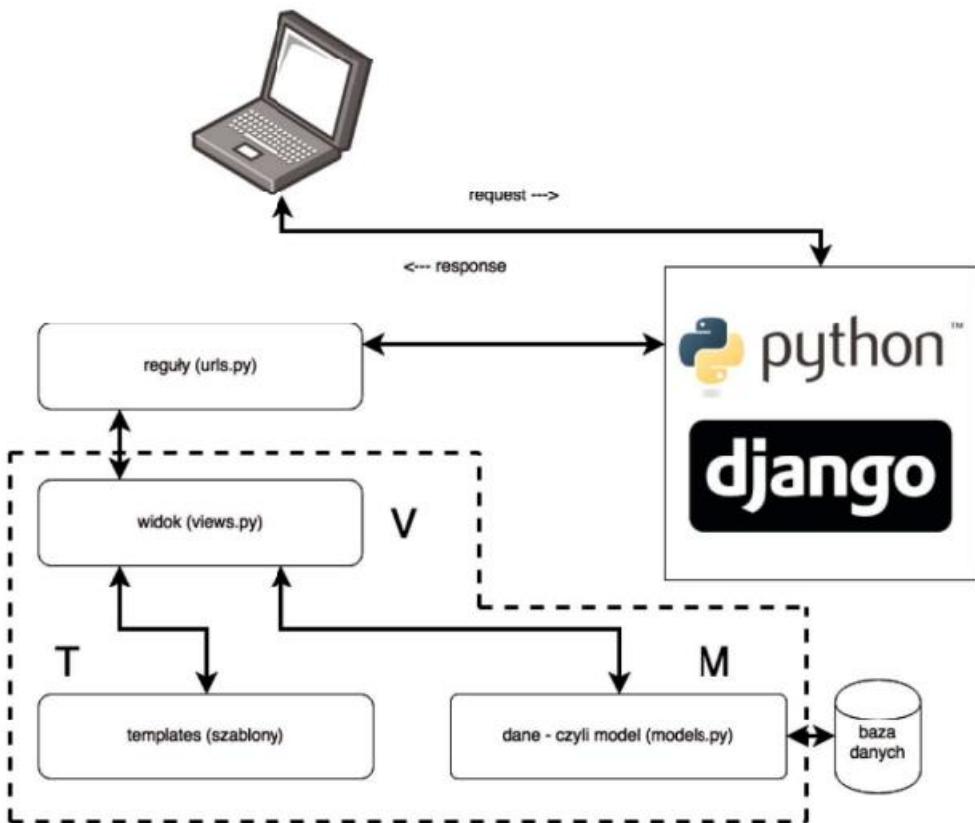
Jak działa aplikacja w Django

Aplikacje internetowe często opierają się na wzorcu MVC (Model-View-Controller). Więcej o tym wzorcu przeczytać można np. na wikipedii: <https://pl.wikipedia.org/wiki/Model-View-Controller>

Wzorzec zastosowany w Django jest odmianą MVC i określany jest skrótem MVT:

1 Model - View - Template

Oznacza to tu tyle, że same widoki ściślej powiązane są z adresami URL i decydują o tym co będzie zwracane - tym samym pełnią funkcję kontrolera. Szablony (template) odpowiadają natomiast za to w jaki sposób dane zostaną zaprezentowane - w kontekście MVC stają się więc widokiem. Poniższy diagram pokazuje uproszczony model komunikacji. Request z komputera klienckiego przekazywany jest do aplikacji Django, w której analizowany jest wzorzec adresu (urls.py) i wywoływana odpowiednia funkcja - nazywana widokiem. Widok może zaciągnąć dane z modelu, który pobiera dane - np. z bazy danych. Widok może też wyrenderować szablon w oparciu o te dane. Tworzony jest response, który odbywa tę samą drogę, ale w kierunku przeciwnym. Linią przerywaną zaznaczono obszar MVT

**Figure 0.7:** img

- **Modele** są to stworzone przez nas klasy odpowiadające tabelkom bazodanowym.

- **Widoki** najogólniej mówiąc są to stworzone przez nas funkcje, które potrafią pobrać dane z modeli, wygenerować na ich podstawie treść strony i zwrócić ją.

Kiedy do serwera WWW przychodzi żądanie, trafia ono do Django. Django analizuje żądanie i na podstawie napisanych przez nas **reguł** decyduje, który widok wywołać i jakie parametry mu przekazać. Wywołany widok generuje stronę, ta zostaje odesłana do klienta. Nasza aplikacja będzie miała tylko jeden model - wpisy (tagi i komentarze dodamy potem). Będą też dwa widoki: `index` będzie pokazywał listę wszystkich wpisów i treść najnowszego z nich. Wywoływać go będzie URL `http://localhost:8000/blog/` `detail` będzie pokazywał treść jednego wybranego wpisu. Wywoływać go będzie URL

```
1 http://localhost:8000/blog/wpisy/<numer_wpisu>.
```

Aplikacje Django

Słowo aplikacja, podobnie jak baza danych, może oznaczać co innego w zależności od kontekstu. Z punktu widzenia użytkownika cały projekt napisany w Django, czy innym frameworku może być nazywany aplikacją. Stąd np. aplikacje bankowe, pogodowe. Można o nich pomyśleć jak o pewnych całościach - portalach.

W Django aplikacja ma dodatkowe znaczenie. Nazywają się tak komponenty, z których zbudowany jest projekt. Część z tych aplikacji jest udostępniana już przez samo Django. Część użytkownik tworzy sam. Jeszcze inne aplikacje mogą być udostępnione przez innych użytkowników. Zainstalowane mogą być użyte w naszym projekcie.

Czas na nasze pierwsze własne aplikacje. Dzięki pierwszej z nich zobaczymy jak działa routing - czyli obsługa adresów wpisywanych przeglądarki i obsługę ich przez Django.

Zatrzymajmy na chwilę nasz serwer - wystarczy kombinacja przycisków `Ctrl + c`. Następnie wpiszmy polecenie:

```
(.venv) $ python manage.py startapp maths
```

Struktura po tej operacji będzie taka:

```
1 (.venv) $ tree
2 .
3   db.sqlite3
4   jazzy
5     __init__.py
6     asgi.py
7     settings.py
8     urls.py
```

```
9     wsgi.py|  
10    manage.py|  
11    maths|  
12        __init__.py|  
13        admin.py|  
14        apps.py|  
15        migrations|  
16            __init__.py|  
17            models.py|  
18            tests.py|  
19            views.py
```

Utwórzmy też od razu drugą aplikację blog. Posłuży nam ona do zapoznania się z modelami i pracą z nimi.

```
1 $ python manage.py startapp blog
```

Dostaniemy podobną strukturę - tym razem dla aplikacji blog

```
1 $ tree  
2 .|  
3   blog|  
4       admin.py|  
5       apps.py|  
6       __init__.py|  
7       migrations|  
8           __init__.py|  
9           models.py|  
10          tests.py|  
11          views.py|  
12   jazzy|  
13   ...|
```

Pliki aplikacji

W obu aplikacjach Django automatycznie utworzyło szkielet w postaci plików i katalogów

- admin.py - służy do opisu interakcji z Panelem Admina
- apps.py - pozwala na konfigurację pewnych metainformacji aplikacji - np. zmianę jej nazwy widocznej w panelu administratora
- 'migrations' - folder zawierający migracje - czyli opis zmian tabeli w bazie danych
- models.py - definicja modeli, czyli opis tabeli w bazie danych, oraz szereg przydatnych metod
- tests.py - miejsce na testy

- `views.py` - definicje widoków, czyli warstwy, która pobiera dane z modelu i renderuje odpowiedź HTTP

Dzięki takiej separacji oddzielamy od siebie różne warstwy naszej aplikacji. W miarę rozwoju w takich aplikacjach dochodzić mogą kolejne pliki - np:

- `forms.py` - definicje formularzy potrzebnych w aplikacji
- `urls.py` - routing na poziomie aplikacji
- `services.py` - warstwa w której często wydziela się logikę biznesową - tak by nie zawierać jej w modelach ani widokach.
- `serializers.py` - definicje serializerów - czyli warstwa odpowiedzialna za zamianę obiektów Pythona w bardziej trwałe, czy uniwersalne formy - takie jak np. `xml`, `json`, `yaml`

i tak naprawdę jeszcze wiele innych, gdy jest taka potrzeba. Często dobrym pomysłem jest wydzielanie modułu zamiast wciskanie na siłę kodu do jednego z powstałych plików

Routing i pierwsze widoki. (`urls.py`)

Jak widzieliśmy, po utworzeniu aplikacji doszedł nowy folder - `maths`, a w nim szereg plików. Będziemy je omawiać dokładniej, ale na razie skupimy się na pliku `views.py`. Ważny też dla nas będzie plik `urls.py` z katalogu `jazzy`. Zaczniemy od tego drugiego. Edytować pliki możesz w zasadzie w dowolnym edytorze tekstowym. Idealnie będzie, jeśli taki edytor wesprze Cię, kolorując składnię, dbając o poprawne wcięcia, czy podpowiadając. W większości z takich edytorów czy IDE można takie rzeczy skonfigurować. Pamiętaj, by trzymać się zaleceń z PEP8. Jeśli zastanawiasz się jaki edytor, czy środowisko wybrać - to rozważ użycie PyCharm, Visual Studio Code, Sublime, Atom. Możesz używać dowolnego narzędzia - zadbaj jednak o to, by Cię nie ograniczało a wspierało. Wróćmy do pliku `urls.py`. Obecnie ma taką treść:

```
1 """jazzy URL Configuration
2
3 The `urlpatterns` list routes URLs to views. For more information
4     please see:
5         https://docs.djangoproject.com/en/3.0/topics/http/urls/
6 Examples:
7     1. Add an import: from my_app import views
8         2. Add a URL to urlpatterns: path('', views.home, name='home')
9 Class-based views
10    1. Add an import: from other_app.views import Home
11        2. Add a URL to urlpatterns: path('', Home.as_view(), name='home')
12 Including another URLconf
13    1. Import the include() function: from django.urls import include,
14        path
14    2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
```

```
15 """
16 from django.contrib import admin
17 from django.urls import path
18
19 urlpatterns = [
20     path('admin/', admin.site.urls),
21 ]
```

Najważniejszy tutaj obiekt to lista `urlpatterns`. Łączy ona adresy wpisywane w przeglądarce - czyli adres żądania (`request`) z odpowiednimi akcjami. Inaczej mówiąc routing odpowiada za to co ma się stać, gdy ktoś wejdzie na dany adres. Już teraz widać, że coś jest tutaj obsługiwane. Jeśli uruchomisz serwer i wpiszesz adres `http://localhost:8000/admin/`, to zobaczysz coś takiego:

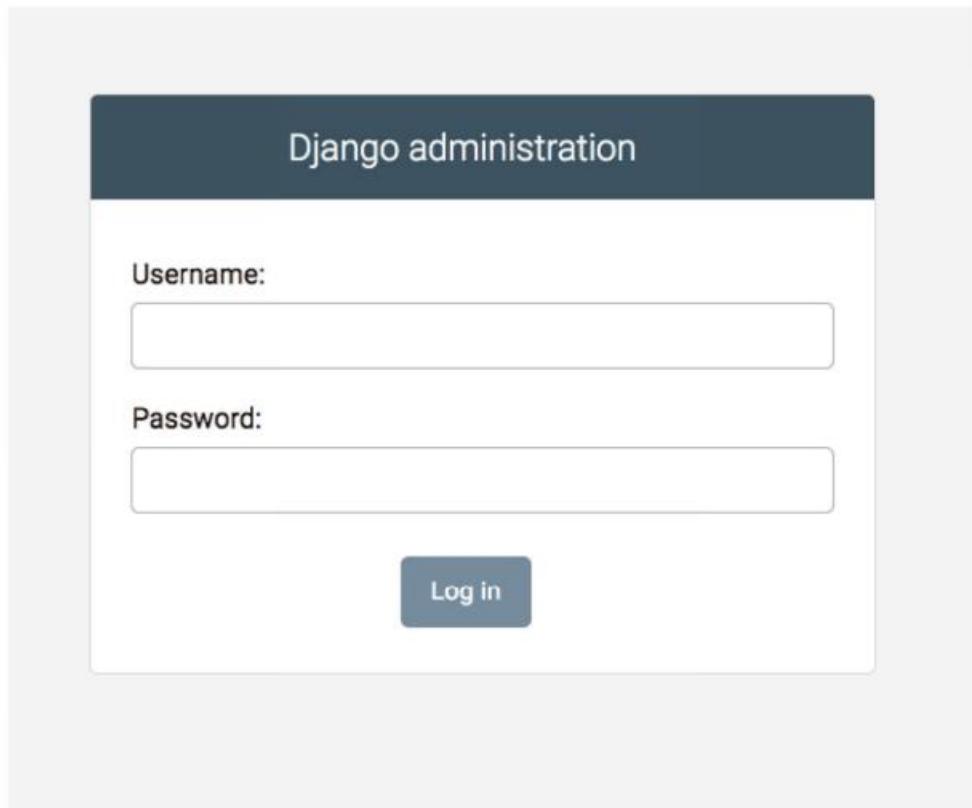


Figure 0.8: img

To formularz logowania do panelu administratora. Wróćmy do tego jeszcze. Teraz jednak skupimy się

na mechanizmie routingu. Jeśli zamiast admin w adresie wpiszemy coś innego, to dostaniemy widok błędu, np. wejście na stronę: <http://localhost:8000/math> daje wynik:

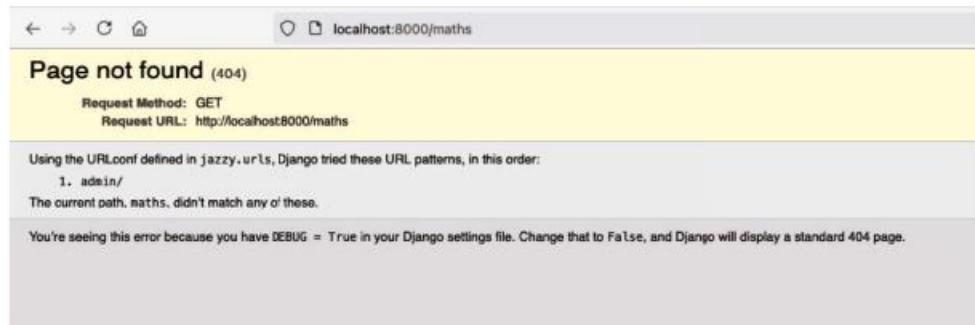


Figure 0.9: image-20210608152828038

Nasza aplikacja po prostu nie umie jeszcze zareagować na żaden inny adres. Możemy dodać do niej nową ścieżkę. Taka ścieżka (path) łączy ze sobą jakiś adres, a dokładniej wzorzec adresu, z funkcją, której zadaniem jest zwrócenie obiektu `HttpResponse`. Ten ostatni obiekt oraz `HttpRequest` to jedne z najstarszych obiektów w Django. Odpowiadają one charakterowi komunikacji przez protokół HTTP. Najogólniej mówiąc działa to tak, że jakąś aplikacja kliencka (np. nasza przeglądarka internetowa) wysyła zapytanie (request) do serwera. Serwer przygotowuje odpowiedź (response) i ją odsyła. W praktyce ten model jest bardziej skomplikowany - zapytanie bowiem wędruje przez cały szereg warstw - od warstwy fizycznej po aplikacyjną i dzieje się z nim bardzo dużo skomplikowanych rzeczy. Dla nas jednak wystarczy te uproszczenie. Nasze Django otrzymuje `request` i musi przygotować odpowiedź. Tą odpowiedzią jest zawsze obiekt `HttpResponse`. Powinien on pochodzić z czegoś, co można wywołać (a więc z funkcji), która musi przyjmować przynajmniej jeden argument - `request`. Jak mogłaby wyglądać najprostsza realizacja takiego problemu? Np. tak:

```
1 # jazzy/urls.py
2
3 from django.contrib import admin
4
5 from django.urls import path
6
7 from django.http import HttpResponse
8
9 urlpatterns = [
10     path('admin/', admin.site.urls),
11
12     path('maths/', lambda request: HttpResponse(**"Tu ębdzie matma"**))
13 ]
```

15]

Zimportowano tu po prostu `HttpResponse` i powiązano adres `maths` z funkcją przyjmującą jakiś argument (dla czytelności nazwałem go `request`) i zwracającą instancję `HttpResponse` - wypełnioną tekstem. Po wejściu na adres `http://localhost:8000/maths/` zobaczymy stronę, na której będzie napis:

```
1 Tu będzie matma
```

Jak widać, ścieżka powiązana tu jest z funkcją przyjmującą `request` i zwracającą `HttpResponse`. Takie funkcje w Django nazywamy widokami. Powyżej zastosowano funkcje anonimowe lambda. Oczywiście może to być też zwyczajnie zdefiniowana funkcja - takie za chwilę wprowadzimy. Istotne w tym miejscu jest to, że funkcja taka zawsze jako pierwszy argument przyjmuje `request`, a zwraca obiekt typu `HttpResponse`.

Teraz pomysł jest taki. Jeśli wejdziemy na stronę:

```
1 http://localhost:8000/maths/add/1/2
```

To chcemy zobaczyć wynik 3 - czyli sumę. Dzięki temu zrozumiemy trochę lepiej, jak powstają te ścieżki i jak są obsługiwane.

Dodajmy więc do urls ścieżkę:

```
1 path('maths/add/1/2', lambda request: HttpResponse(3)),
```

I rzeczywiście - po wejściu na ten adres widać 3. Ale ja zrobić z tego prawdziwe dodawanie?

Jak sprawić, by zadziałało to dla każdej liczby? Np.:

```
1 http://localhost:8000/maths/add/3/3
```

znowu da błąd `Http 404` mówiący, że nie ma takiej strony. Zamiast na sztywno wpisywać liczby, powinniśmy w urls jakoś opisać, że to są zmienne. Twórcy Django pomyśleli o tym. Zamiast wcześniejszego `path` dodajmy takie:

```
1 path('maths/add/<a>/<b>', lambda request, a, b: HttpResponse(a+b)),
```

parametry w ostrych nawiasach to będą nasze zmienne. Takie zmienne będą przekazywane do naszej funkcji i w niej użyte. Wszystko wygląda pięknie. Tylko dlaczego wynik to 33? Obie zmienne traktowane są jako napis. Teraz mamy dwa wyjścia. Możemy wskazać, że jest to liczba już w adres, np: `<int:a>` albo rzutować to na odpowiedni typ w momencie użycia. Uwaga! Pierwszy sposób zadziała tylko dla int. Poniżej przedstawię sposób kombinowany:

```
1 path(
2     'maths/add/<int:a>/<b>',
3     lambda request, a, b: HttpResponse(a+int(b)))
```

```
4 ),
```

W ramach ćwiczenia - w podobny sposób spróbuj dopisać w routingu rozwiązanie następujących requestów:

`http://localhost:8000/math/sub/1/2` powinno dać `-1`

`http://localhost:8000/math/mul/3/2` da wynik `6`

`http://localhost:8000/math/div/3/2` to z kolei `1.5`

No świetnie, ale po co nam była ta cała aplikacja? Co by było, gdyby nasz problem był bardziej skomplikowany? Gdyby prosta anoniczna funkcja nie wystarczyła? Np. co z dzieleniem przez zero? Pewnie byśmy chcieli tam dodać jakieś sprawdzenie, czy b to czasem nie jest 0. Przenieślibyśmy nasz kod do zwykłych funkcji. Zasada byłaby ta sama. Przyjmujemy `request` i co tam jeszcze trzeba, a zwracamy `HttpResponse`. Nasz plik `urls.py` mógłby wyglądać tak:

```
1 # jazzy/urls.py
2 from django.contrib import admin
3 from django.urls import path
4 from django.http import HttpResponse
5
6 def math(request):
7     return HttpResponse("Tu ębdzie matma")
8
9 def add(request, a, b):
10    return HttpResponse(a + b)
11
12 def sub(request, a, b):
13    return HttpResponse(a - b)
14
15 def mul(request, a, b):
16    return HttpResponse(a * b)
17
18 def div(request, a, b):
19    if b == 0:
20        return HttpResponse("Nie dziel przez 0")
21    return HttpResponse(a / b)
22
23 urlpatterns = [
24     path('admin/', admin.site.urls),
25     path('maths/', math),
26     path('maths/add/<int:a>/<int:b>', add),
27     path('maths/sub/<int:a>/<int:b>', sub),
28     path('maths/mul/<int:a>/<int:b>', mul),
29     path('maths/div/<int:a>/<int:b>', div),
30 ]
```

Gdyby nasza aplikacja miała się na tym zakończyć, to moglibyśmy na tym poprzestać. Ale skoro zaimerzamy ją rozwijać, to może lepiej będzie wydzielić te funkcje do osobnego modułu? A skoro to

są funkcje związane z maths, to zapewne dobrym pomysłem jest to, by znalazły się one w aplikacji maths w jakimś module. W jakim? Zapewne domyślasz się już po co jest plik views.py w aplikacji.

Takie funkcje nazywamy w Django widokami. Jest to element pewnego wzorca projektowego MVT - (Model View Template), który z kolei jest wariacją na temat innego wzorca - MVC (Model View Controller). Wzorce te zakładają podzielenie aplikacji na pewne warstwy. I grupowanie kodu odpowiedzialnego za te warstwy razem.

Musimy jeszcze dopisać tę aplikację do listy aplikacji projektu jazzy. W tym celu w pliku settings.py do krotki INSTALLED_APPS dopisujemy:

```
1 INSTALLED_APPS = (
2     ...
3     'blog',
4 )
```

Teraz mamy już wszystko przygotowane. Możemy zacząć pisać aplikację.

Dla dociekliwych: Jak działa urls.py

W urls.py możemy znaleźć np. takie wpisy:

```
1 urlpatterns = patterns('',
2     url(r'^admin/', include(admin.site.urls)),
3     url(r'^login', 'django.contrib.auth.views.login'),
4     url(r'^$', include('myapp.urls')),
5     path('blog', blog.views.index),
6 )
```

Jak widzimy są tu użyte funkcje url i path. Działają podobnie, ale używamy dziś głównie path. url znajdziemy w starszych wersjach projektów opartych o Django. Zrozummy dokładniej, jak to działa.

Funkcji path możemy podać wzorzec i nazwę funkcji. Tworzy ona obiekt klasy URLPattern pamiętający ten wzorzec i tę nazwę funkcji:

```
1 In[2]: from django.urls import path
2 In[3]: def mojafunkcja(x):
3     ...:     return 3*x
4     ...:
5 In[4]: p = path("wykonaj", mojafunkcja)
6 In[5]: p
7 Out[5]: <URLPattern 'wykonaj'>
8 p.callback()
9 Traceback (most recent call last):
10   File "/home/jazzy/venv/lib/python3.7/site-
11   packages/IPython/core/interactiveshell.py", line 3267, in run_code
```

```
12     exec(code_obj, self.user_global_ns, self.user_ns)
13 File "<ipython-input-6-cd5cbc05abb>", line 1, in <module>
14     p.callback()
15 TypeError: mojafunkcja() missing 1 required positional argument: 'x'
16 In[7]: p.callback
17 Out[7]: <function __main__.mojafunkcja(x)>
18 In[8]: p.pattern
19 Out[8]: <django.urls.resolvers.RoutePattern at 0x7fc2f4894400>
20 In[9]: p.pattern.regex()
21 Traceback (most recent call last):
22   File "/home/jazzy/venv/lib/python3.7/site-packages/IPython/core/
23     interactiveshell.py",
24   line 3267, in run_code
25     exec(code_obj, self.user_global_ns, self.user_ns)
26 File "<ipython-input-9-4dd949edde8d>", line 1, in <module>
27     p.pattern.regex()
28 TypeError: 're.Pattern' object is not callable
29 In[10]: re.compile(r'^wykonaj$', re.UNICODE)
30 In[11]: p.resolve("ala ma kota")
31 In[12]: p.resolve("wykonaj/cos")
32 In[13]: p.resolve("wykonaj")
33 Out[13]: ResolverMatch(
34     func=__main__.mojafunkcja, args=(), kwargs={},
35     url_name=None, app_names=[], namespaces=[]
36 )
37 In[14]: p = path("wykonaj/", mojafunkcja)
38 In[15]: p.resolve("wykonaj/cos")
39 In[16]: p.resolve("wykonaj/<id>")
40 In[17]: p = path("wykonaj/<id>", mojafunkcja)
41 In[18]: p.resolve("wykonaj/cos")
42 Out[18]: ResolverMatch(
43     func=__main__.mojafunkcja, args=(),
44     kwargs={'id': 'cos'}, url_name=None, app_names=[], namespaces=[])
```

Jak widzimy obiekt klasy `URLPattern` ma metodę `resolve`. Możemy tej metodzie podać napis, a ona sprawdza, czy ten napis pasuje do pamiętanego wzorca. Jeśli nie pasuje, metoda `resolve` zwraca `None`. Jeśli pasuje, zwraca obiekt klasy `ResolverMatch` zawierający między innymi nazwę funkcji.

Więcej o funkcji `path` znajdziemy w dokumentacji projektu: <https://docs.djangoproject.com/en/2.1/ref/urls/>

Połączenie z bazą

SQLite

W domyślnej konfiguracji Django używana jest baza danych SQLite (<https://www.sqlite.org/index.html>), która dobrze może sprawdzić się przy pierwszych ćwiczeniach, czy też w bardzo małych aplikacjach

- np. do użytku domowego. Na potrzeby naszych ćwiczeń taka podstawowa konfiguracja powinna wystarczyć.

Inne bazy

Django pozwala na połączenia z innymi typami baz. Współpracuje z szeregiem różnych baz relacyjnych takich jak MySQL, PostgreSQL a także bazami nierelacyjnymi, np. z mongoDB (<https://django-mongodb-engine.readthedocs.io/en/latest/>).

W zależności od wyboru bazy zmieniać się będzie konfiguracja i użyty silnik do komunikacji pomiędzy Django oraz bazą. Najczęściej stosowaną obecnie bazą danych jest PostgreSQL (<https://www.postgresql.org/>).

Nasze pierwsze projekty będziemy opierać o SQLite, gdyby jednak zaistniała potrzeba skorzystania z PostgreSQL należy upewnić się, że baza danych jest zainstalowana w systemie, np. na linux:

```
1 $ which psql
2 /usr/bin/psql
3 $ psql -V
4 psql (PostgreSQL) 9.3.25
```

Na tym etapie baza danych to program, który umożliwia nam przechowywanie, dodawanie i przeszukiwanie różnych danych. Mając bazę w naszym systemie możemy utworzyć użytkownika i bazę danych pod konkretny projekt - czyli tutaj jako bazę bardziej rozumieć już zestaw konkretnych tabel i danych w konkretnym projekcie. By tak pojęte bazy tworzyć musimy mieć użytkownika bazodanowego np.:

```
1 $ sudo su - postgres
2 $ createuser django -Wt
3 Haso: ...
4 $ createdb -O django jazzy
```

Lub po uruchomieniu programu psql:

```
1 create user django_01 with password 'django_01';
2 create database django_01 with encoding 'UTF-8';
3 alter database django_01 owner to django_01;
```

Jeśli w Django zdecydujemy się na skorzystanie z bazy innej niż SQLite, to trzeba będzie zmodyfikować plik konfiguracyjny `settings.py`. Musimy w nim odnaleźć sekcję DATABASES:

```
1     DATABASES = {
2         'default': {
3             'ENGINE': 'django.db.backends.sqlite3',
4             'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
5         }
6     }
```

```
6     }
```

i zmienić jej ustawienia:

```
1      DATABASES = {  
2          'default': {  
3              'ENGINE': 'django.db.backends.postgresql_psycopg2',  
4              'NAME': 'jazzy',  
5              'USER': 'django_01',  
6              'PASSWORD': 'django_01',  
7              'HOST': 'localhost',  
8              'PORT': '',  
9          },  
10     }
```

Zgodnie z poniższym wyjaśnieniem:

- ENGINE - rodzaj DBMS, z którego będziemy korzystać. My będziemy korzystać z PostgreSQL-a, więc wpisujemy django.db.backends.postgresql_psycopg2.
- NAME - nazwa bazy danych. Wpisujemy django_01.
- USER - wpisujemy django_01.
- PASSWORD - wpisujemy django_01.
- HOST - pozostawimy puste, a Django będzie łączyć się z lokalną maszyną.
- PORT - pozostawimy puste, a Django będzie łączyć się z portem o domyślnym numerze.

By ustawienia zadziałyły prawdopodobnie trzeba będzie jeszcze zainstalować moduł psycopg2, który pozwala na komunikację pythona z Postgres

```
1 $ pip install psycopg2
```

Przy uruchomieniu programu komendą manage.py runserver możesz zauważyc informację:

```
1 $ python manage.py runserver  
2 Performing system checks...  
3  
4 System check identified no issues (0 silenced).  
5  
6 You have 15 unapplied migration(s). Your project may not work properly  
7 until you apply the migrations for app(s): admin, auth, contenttypes,  
8 sessions.  
9  
10 Run 'python manage.py migrate' to apply them.  
11  
12 December 13, 2018 - 10:43:37  
13 Django version 2.1.4, using settings 'jazzy.settings'  
14 Starting development server at http://127.0.0.1:8000/  
15 Quit the server with CONTROL-C.
```

Oznacza to, że aplikacja co prawda działa i potrafi połączyć się z bazą danych, ale na razie nie zostały tam utworzone żadne tabelki, w których będą przechowywane dane:

Należy zgodnie z instrukcją wykonać:

```
1 $ python manage.py migrate
```

Powinniśmy otrzymać wynik podobny do tego poniżej:

```
1 Operations to perform:
2   Apply all migrations: admin, auth, contenttypes, sessions
3   Running migrations:
4     Applying contenttypes.0001_initial... OK
5     Applying auth.0001_initial... OK
6     Applying admin.0001_initial... OK
7     Applying admin.0002_logentry_remove_auto_add... OK
8     Applying admin.0003_logentry_add_action_flag_choices... OK
9     Applying contenttypes.0002_remove_content_type_name... OK
10    Applying auth.0002_alter_permission_name_max_length... OK
11    Applying auth.0003_alter_user_email_max_length... OK
12    Applying auth.0004_alter_user_username_opts... OK
13    Applying auth.0005_alter_user_last_login_null... OK
14    Applying auth.0006_require_contenttypes_0002... OK
15    Applying auth.0007_alter_validators_add_error_messages... OK
16    Applying auth.0008_alter_user_username_max_length... OK
17    Applying auth.0009_alter_user_last_name_max_length... OK
18    Applying sessions.0001_initial... OK
```

Dalej można jeszcze skorzystać z innej komendy z `manage.py` i utworzyć administratora naszej aplikacji:

```
1 $ python manage.py createsuperuser
2 Username (leave blank to use 'xxx'): <twoja nazwa użytkownika>
3 Email address: <twoj@email>
4 Password:
5 Password (again):
6 Superuser created successfully.
```

Widok

Jak mówiliśmy, widok jest funkcją. Dostaje ona jeden argument, zawierający informacje o otrzymanym żądaniu. Powinna zwrócić obiekt klasy `HttpResponse` zawierający odpowiedź, która ma być odesłana klientowi. Klasa `HttpResponse` jest zdefiniowana w module `django.http`. Jej konstruktor przyjmuje jeden argument - treść strony do wyświetlenia.

```
1 # views.py
2 from django.shortcuts import render
```

```
3 from django.http import HttpResponse
4 # Create your views here.
5 def index(request):
6     return HttpResponse("Hello World!")
```

Teraz po wejściu na stronę <http://127.0.0.1:8000/blog/> powinniśmy zobaczyć napis:

```
1 Hello World!
```

Modele i ORM

W katalogu aplikacji znajduje się automatycznie utworzony plik `models.py`. W nim powinniśmy umieszczać tworzone przez nas modele. Każdy model jest klasą (dziedziczącą po klasie `Model` z modułu `django.db.models`) zawierającą opis jednej tabelki bazodanowej. Klasa taka powinna mieć kilka atrybutów - każdy z nich jest opisem jednej kolumny w tabelce. Atrybuty powinny być obiektami klas zdefiniowanych w `django.db.models` - każda klasa oznacza jeden rodzaj kolumny.

- Klasa `BooleanField` oznacza kolumnę typu `boolean`.
- Klasa `CharField` oznacza kolumnę zawierającą krótki napis. Przy tworzeniu obiektu tej klasy trzeba podać argument `max_length` określający maksymalną długość napisu.
- klasa `DateField` oznacza kolumnę zawierającą datę.
- klasa `DateTimeField` oznacza kolumnę zawierającą timestamp.
- klasa `FloatField` oznacza kolumnę zawierającą liczbę zmiennoprzecinkową.
- klasa `IntegerField` oznacza kolumnę zawierającą liczbę całkowitą.
- klasa `TextField` oznacza kolumnę zawierającą długi tekst. Przykładowo, tabelka zawodnicy z kolumnami imię, nazwisko i wzrost może mieć taki model:

```
1 from django.db import models
2 # Create your models here.
3 class zawodnicy(models.Model):
4     imie = models.CharField(max_length=200)
5     nazwisko = models.CharField(max_length=200)
6     wzrost = models.IntegerField()
```

Kolumny będące kluczem głównym tabeli nie musimy wymieniać wprost, Django sam o niego zadba. Pełną listę klas znajdziemy pod adresem: <http://docs.djangoproject.com/en/dev/ref/models/fields/#model-field-types>

<http://tinyurl.com/bvrfuw>

W naszym projekcie utworzymy model `Wpis`. Może on wyglądać następująco:

```
1 class Wpis(models.Model):
2     tytul_wpisu = models.CharField(max_length=200)
```

```
3     tresc_wpisu = models.TextField()
```

Stworzenie tego modelu spowodowało, że:

- Django będzie umiał samemu stworzyć w bazie danych wszystkie potrzebne aplikacji tabelki.
- Django da obiektowy dostęp do stworzonych tabelek. Będziemy mogli korzystać z niego zarówno w widoku, jak i w trybie interaktywnym.
- Django stworzy panel administracyjny, pozwalający edytować dane z modelu.

Teraz jeszcze musimy nasze zmiany nanieść na bazę danych. Jeśli tylko nasza aplikacja jest dodana do `settings.INSTALLED_APPS` to powinny zadziałać poniższe komendy:

```
1 $ ./manage.py makemigrations
2 Migrations for 'blog':
3   blog/migrations/0001_initial.py
4     - Create model Wpis
```

Powyżej przygotowana została migracja - śmiało możesz podejrzeć jej kod.

```
1 $ ./manage.py migrate
2 Operations to perform:
3   Apply all migrations: admin, auth, blog, contenttypes, sessions
4   Running migrations:
5     Applying blog.0001_initial... OK
```

W tym przypadku utworzoną migrację wykonaliśmy. Zmiany zostały naniesione na bazę danych. Więcej o samych migracjach możesz przeczytać tutaj: <https://docs.djangoproject.com/en/2.1/topics/migrations/>

Obiektowy dostęp do bazy danych

Zobaczmy, jak korzystać z obiektowego dostępu do bazy danych. Aby to zrobić wejdziemy w tryb interaktywny Pythona. Najpierw jednak zainstalujemy lepszą wersję REPL - ipythona:

```
1 $ pip install ipython
```

następnie wydajemy polecenie wydajmy polecenie:

```
1 $ python manage.py shell
```

Polecenie to importuje ustawienia projektu, ustawia odpowiednie zmienne środowiskowe i uruchamia Pythona. Żeby móc korzystać z naszego modelu, musimy go zaimportować:

```
1 In [1]: from blog.models import Wpis
```

Koncepcja obiektowego dostępu do bazy danych jest prosta. Stworzona przez nas klasa Wpis reprezentuje całą tabelkę z wpisami do bloga. Obiekty tej klasy reprezentują poszczególne wiersze tej tabelki. Przy tworzeniu obiektu jako nazwane argumenty konstruktora podajemy zawartość pól tworzonyego wiersza. Utworzenie obiektu nie powoduje automatycznego umieszczenia go w bazie danych - w tym celu trzeba wywołać jego metodę save(). Aby więc stworzyć dwa nowe wpisy i umieścić je w bazie danych, wydajemy polecenia:

```
1 In [2]: w1 = Wpis(  
2     tytul_wpisu='raz dwa trzy',  
3     tresc_wpisu='Oto pierwszy testowy wpis.  
4     'Raz dwa trzy, raz dwa trzy')  
5 )  
6 In [3]: w2 = Wpis(  
7     tytul_wpisu='cztery',  
8     tresc_wpisu='Oto drugi testowy wpis. Cztery cztery cztery.'  
9 )  
10 In [4]: w1.save()  
11 In [5]: w2.save()
```

Polecenie save() zachowuje się inteligentnie. Jeśli zmienimy istniejący obiekt i wywołamy na nim save(), nie zostanie utworzony nowy wpis (nie zostanie wykonany INSERT), ale zmieniony będzie ten istniejący (wykonany zostanie UPDATE).

pobranie wiersza z bazy

W pobieraniu wierszy z tabelki bazodanowej pośredniczy menadżer - jest nim atrybut objects naszej klasy. Ma on na przykład metodę get() - służy ona do pobrania pojedynczego wiersza. Podajemy jej warunki, według których ma zostać wybrany wiersz, a ona zwraca obiekt z tym wierszem:

```
1     In [8]: w4 = Wpis.objects.get(tytul_wpisu='raz dwa trzy')  
2     In [9]: w4  
3     Out[9]: <Wpis: Wpis object (1)>
```

Jeśli chcemy pobrać obiekt o danej wartości klucza głównego, nie musimy wiedzieć, jak nazywa się kolumna, która go zawiera możemy użyć nazwy pk:

```
1     w4 = Wpis.objects.get(pk=1)
```

Pobieranie grup wierszy z bazy

Menadżer ma też metody służące do wybierania grup wierszy. Metoda all() wybiera wszystkie wiersze:

```
1 In [10]: wpisy = Wpis.objects.all()
2 In [11]: wpisy
3 Out[11]: <QuerySet [<Wpis: Wpis object (1)>, <Wpis: Wpis object (2)>]>
```

Metoda ta nie zwraca listy obiektów klasy `Wpis`, bo byłoby to niewydajne. Zwraca on obiekt klasy `QuerySet`, który przechowuje informacje o zapytaniu, które trzeba będzie wydać, żeby pobrać dane z bazy.

Treść zapytania SQL która stoi za tym zapytaniem możemy obejrzeć:

```
1 In [14]: print(wpisy.query)
2 SELECT "blog_wpis"."id", "blog_wpis"."tytul_wpisu",
3 "blog_wpis"."tresc_wpisu" FROM "blog_wpis"
```

W praktyce w większości zwykłych przypadków możemy jednak wyobrażać sobie, że w zwróconym obiekcie mamy listę wierszy. Jeśli spróbujemy po niej iterować, otrzymamy po prostu poszczególne obiekty:

```
1 In [15]: for w in wpisy:
2     ...:     print(w.tytul_wpisu)
3     ...:
4
5 raz dwa trzy
6 Cztery
```

Można też pobrać wiersze spełniające określone warunki:

```
1 In [16]: wpisy = Wpis.objects.filter(tresc_wpisu__contains='dwa')
2 In [17]: wpisy
3 Out[17]: <QuerySet [<Wpis: Wpis object (1)>]>
```

Jak widać, warunek ma postać:

```
1 nazwa kolumny + __ + rodzaj warunku = 'ścisłe warunku'
```

Warunek, który użyliśmy powyżej, oznacza: wybierz wiersze, w których kolumna `tresc_wpisu` zawiera napis 'dwa'.

Inne rodzaje warunków to: * `__exact` - szukamy wierszy o danej treści, * `__iexact` - jak `__exact`, ale bez zwracania uwagi na wielkość liter, * `__contains` - szukamy wierszy zawierających w sobie daną treść, * `__gt` - większy niż, * `__lt` - mniejszy niż, * `__in` - wartość należy do listy, * `__range` - wartość należy do zakresu podanego jako krotka, * `__year` - rok w dacie jest równy danej wartości.

Jeśli nie podamy rodzaju warunku, użyty jest warunek `__exact`. Analogiczną do `filter()` jest metoda `exclude()` - powoduje ona wybranie wierszy, które nie spełniają warunku. Warunki można ze sobą łączyć:

```
1 In [18]: wpisy = Wpis.objects.filter(
2     tresc_wpisu__contains='dwa').filter(
3     tresc_wpisu__contains='afryka')
4
5 In [19]: wpisy
6 Out[19]: <QuerySet []>
```

Warunki łączone ze sobą w ten sposób zostają połączone przez AND - powyższe polecenie wybierze wpisy zawierające w treści słowa dwa i afryka. Bardziej złożone zapytania można konstruować przy użyciu obiektów klasy Q. Obiekt klasy Q zawiera warunki pytania, więc w najprostszy sposób można użyć go tak:

```
1 In [20]: from django.db.models import Q
2 In [21]: warunek = Q(tytul_wpisu__contains='dwa')
3 In [22]: Wpis.objects.filter(warunek)
4 Out[22]: <QuerySet [<Wpis: Wpis object (1)>]>
```

Klasa Q ma metodę `__or__`, dzięki czemu jej obiekty można łączyć operatorem logicznym |:

```
1 In [23]: warunek1 = Q(tytul_wpisu__contains='dwa')
2 In [24]: warunek2 = Q(tresc_wpisu__contains='afryka')
3 In [25]: Wpis.objects.filter(warunek1 | warunek2)
4 Out[25]: <QuerySet [<Wpis: Wpis object (1)>]>
```

Jeśli jako argumenty metodzie `filter()` podamy kilka obiektów klasy Q, zostaną one połączone przez AND:

```
1 In [26]: Wpis.objects.filter(warunek1, warunek2)
2 Out[26]: <QuerySet []>
```

Ćwiczenie

- Przerób widok `index` tak, żeby korzystając z poznanych technik pobierał z bazy danych spis tytułów wpisów i wypisywał je jako listę.

Pierwsze testy

Warto od razu napisać kilka testów, które sprawdzą, czy nasza aplikacja działa poprawnie.

W pliku `blog/tests.py` utworzymy dwa pierwsze testy:

```
1 from django.test import TestCase
2 from blog.models import Wpis
3 # Create your tests here.
4
```

```
5 class TestsBlog(TestCase):
6     def test_main_page_of_app_exists(self):
7         response = self.client.get("/blog/")
8         self.assertEqual(response.status_code, 200)
9
10
11
12 class TestWpistModel(TestCase):
13     def test_create_instance(self):
14         w1 = Wpis(
15             tytul_wpisu='raz dwa trzy',
16             tresc_wpisu='Oto pierwszy testowy wpis. Raz dwa trzy, raz
17             dwa trzy'
18         )
19         w1.save()
20         w = Wpis.objects.get(tytul_wpisu='raz dwa trzy')
21         self.assertEqual(w, w1)
22         w1.delete()
```

Po pierwsze testujemy czy strona /blog istnieje. Druga klasa testów to testy modelu. Na razie sprawdzamy czy możemy utworzyć

Szablony

Django posiada wriasny mechanizm szablonów. Szablony te działają podobnie do znanej z C funkcji printf czy pythonowego operatora %. Szablon jest wzorem strony ze wstawionymi znacznikami oznaczającymi tu wstaw cenę chleba, tu wstaw dzisiejszą datę itp. Następnie przygotowujemy coś jakby słownik zawierający dane do podstawienia (cena chleba, dzisiejsza data), łączymy szablon z tymi danymi i dostajemy gotowy napis. Zmienimy teraz naszą funkcję index z modułu views.py aplikacji blog

```
1 from django.http import HttpResponse
2 from django.template import Template, Context
3 # Create your views here.
4 def index(request):
5     t = Template('chleb kosztuje {{ chleb }}')
6     c = Context({'chleb': '1,90'})
7     wynik = t.render(c)
8     return HttpResponse(wynik)
```

Widzimy tu, że obiekt typu Template renderujemy wypełniając go wartościami z obiekty typu Context. Ten ostatni przypomina nim słownik, ale oferuje inne możliwości. Można na nim np. odkładać dla danego klucza wartości - niczym na stosie:

```
1 >>> c = Context()
2 >>> c['foo'] = 'first level'
```

```
3 >>> c.push()
4 {}
5 >>> c['foo'] = 'second level'
6 >>> c['foo']
7 'second level'
8 >>> c.pop()
9 {'foo': 'second level'}
10 >>> c['foo']
11 'first level'
12 >>> c['foo'] = 'overwritten'
13 >>> c['foo']
14 'overwritten'
15 >>> c.pop()
16 Traceback (most recent call last):
17 ...
18 ContextPopException
```

więcej przykładów można znaleźć w dokumentacji: <https://docs.djangoproject.com/en/dev/ref/templates/api/#playing-with-context-objects>

Żeby szablony były lepiej oddzielone od widoków, zwykle z szablonów korzysta się inaczej - szablon zapisujemy w podkatalogu `templates/nazwa_aplikacji` w katalogu aplikacji. Następnie w widoku robimy `loader.get_template('aplikacja/szablon')` - to zwraca nam szablon z załadowaną treścią. Potem w zwykły sposób renderujemy ten szablon podając mu kontekst. Przy takim użyciu jednak context powinien być zwykłym słownikiem, w przeciwnym wypadku możemy otrzymać błąd:



Figure 0.10: image-20210607234616238

Zmieńmy więc funkcję:

```
1 # blog/views.py
2 from django.http import HttpResponse
3 from django.template import loader
4 # Create your views here.
5 def index(request):
```

```
6     t = loader.get_template('blog/index.html')
7     c = {'chleb': '1,90'}
8     wynik = t.render(c)
9     return HttpResponse(wynik)
```

a następnie wejdźmy na stronę <http://127.0.0.1:8000/blog/> Wynik powinien być następujący:

```
1 chleb kosztuje 1,90
```

Pythonowy mechanizm szablonów wygodnie jest testować w trybie interaktywnym (po wydaniu polecenia `python manage.py shell`), tak:

```
1     >>> from django.template import Template, Context
2     >>> Template("test: {{ x }}").render(Context({"x": 7}))
3     >>> 'test: 7'
```

W szablonie nie można umieszczać wstawek w Pythonie - ani pythonowych poleceń, ani pythonowych wyrażeń. Djangowy mechanizm szablonów ma swój własny język o własnej składni, inspirowanej składnią Pythona, ale często inną od składni Pythona. Składnia ta jest dziwna, niekonsekwentna i potrafi zaskakiwać - więc trzeba uważać.

wstawianie zmiennych (i niektórych innych wyrażeń)

Szablon jest zwykłym dokumentem tekstowym (zwykle, oczywiście, dokumentem HTML). Jeśli zechcemy w jakimś jego miejscu wstawić wartość ze słownika przekazanego metodzie `render()`, piszemy:

```
1 {{ klucz }}
```

Spacje przy nawiasach klamrowych nie są obowiązkowe - można też napisać:

```
1 {{klucz}}
```

Ale twórcy dokumentacji do Django w przykładach nie pomijają tych spacji, więc można uznać, że zalecanym stylem jest ich niepowomianie.

Takie wartości przekazywane szablonowi w słowniku nazywane są zmiennymi (w dokumentacji `variables`), ale oczywiście są to zmienne odrębne od zmiennych programu, który renderuje szablon.

Wartości przekazywane przez słownik mogą być nie tylko napisami czy liczbami, ale dowolnymi obiektami. W szablonie możemy odwoływać się do atrybutów i metod obiektów oraz do wartości spod kluczy słowników. Odwołujemy się do nich w taki sam sposób:

```
1 {{ obiekt.atrybut }}
2 {{ obiekt.metoda }}
3 {{ tsownik.klucz }}
```

W podwójnych klamrach nie można umieszczać wyrażeń. **Niepoprawne** są na przykład takie wstawki:

```
1 {{ x + 1 }}  
2 {{ 3 * 7 }}
```

W podwójnych klamrach można używać literałów liczb, wartości boolowskich i literatu `None` - i żadnych innych:

```
1 {{ 3 }}  
2 {{ 5 }}  
3 {{ True }}  
4 {{ False }}  
5 {{ None }}  
6 {{ [3, 5, 7] }}  
7 {{ "dominacja" }}
```

Samo w sobie nie jest to bardzo przydatne, ale ma znaczenie w tagu `if` (który jest opisany poniżej).

filtry

Wyrażenia umieszczone w podwójnych klamrach można przepuszczać przez filtry:

```
1 {{ zmienna|filtr }}  
2 {{ zmienna|filtr|drugifiltr }}
```

Na przykład taki zapis:

```
1 {{ napis|lower }}
```

powoduje wypisanie zmiennej `napis` małymi literami. Dookoła operatora `|` nie może być spacji. Taki zapis byłby więc niepoprawny:

```
1 {{ napis | lower }}
```

Niektórym filtrom można przekazywać argument, w taki sposób:

```
1 {{ zmienna|filtr:argument }}
```

Argumentami przekazywanymi filtrom mogą być wszystkie wyrażenia, które byłyby poprawne w znaczniku `{{ wyrażenie }}` oraz - dodatkowo - literał napisu zapisany w pojedynczym lub podwójnym cudzysłowie. Żaden filtr nie przyjmuje więcej iż jednego argumentu, ale niektóre filtry (na przykład `filtr yesno`) przyjmują pojedynczy argument będący napisem zawierającym kilka części rozdzielonych przecinkami. Niektóre przydatne filtry to: `* napis|capfirst` - wypisuje napis małymi literami, tylko pierwszą literę zmienia na wielką, `* napis|default:"wartość domyślna"` - wypisuje napis, a jeśli go nie ma (jest w nim `None`), wypisuje tekst `wartość domyślona`, `* napis|truncatewords:50` - przycina napis

do 50 słów, * napis|truncatewords_html:50 - przycina napis do 50 słów dbając o to, by po tej operacji nie pozostały żadne niezamknięte tagi.

tagi

Oprócz zmiennych w szablonie można używać tagów. Przykładowo, jeśli chcemy jakiś fragment szablonu powtórzyć kilka razy dla kolejnych wartości z listy, możemy użyć takiej konstrukcji:

```
1  {% for wartosc in lista %}  
2    dowolny kod html  
3  {{ wartosc }}  
4    dowolny kod html  
5  {% endfor %}
```

Jak widać, taga używamy tak, że na początku pewnego fragmentu szablonu piszemy:

```
1  {% tag dodatkowe rzeczy %}  
2    fragment szablonu  
3  {% endtag %}
```

W znaczniku rozpoczętym po nazwie taga dla niektórych tagów podaje się dodatkowe rzeczy. Te dodatkowe rzeczy są napisem zapisywanyem bez cudzysłowu , który zostaje przekazany i w całości zinterpretowany przez taga - więc nie da się podać ogólnej zasady, co jak się w nim zapisuje. W znaczniku kończącym tag po słowie end można podać dodatkowy napis. Niektóre tagi sprawdzają ten napis, inne nie. Przykładowo, tag block (o którym więcej powiemy w dalszych rozdziałach) zaczyna się znacznikiem:

```
1  {% block nazwabloku %}
```

a kończy znacznikiem:

```
1  {% endblock %}
```

lub znacznikiem:

```
1  {% endblock nazwabloku %}
```

Ten drugi zapis - z podaniem nazwy bloku przy zamykaniu bloku - jest wygodny: dzięki niemu dobrze widać, który blok zamykamy, a poza tym pomaga uniknąć krzyżowania znaczników - zamknięcia bloku zewnętrznego przed wewnętrznym. Ale choć zamykając tag for też można w znaczniku zamykającym podać nazwę zmiennej:

```
1  {% for osoba in osoby %}  
2    (...)  
3  {% endfor osoba %}
```

to tag for nie sprawdza, czy te nazwy zmiennej się zgadzają. Więc możemy też napisać:

```
1  {% for osoba in osoby %}  
2      (...)  
3  {% endfor %}
```

i Django nie ostrzeże nas, że napisaliśmy bzdurę - co jest niebezpieczne, bo daje fałszywe poczucie bezpieczeństwa. Spacje przy nawiasie można pomijać:

```
1  {%tag dodatkowe rzeczy%}  
2      fragment szablonu  
3  {%endtag%}
```

Ale twórcy dokumentacji do Django w przykładach nie pomijają tych spacji, więc można uznać, że zalecanym stylem jest ich nieompomijanie. Są również takie tagi, które nie obejmują sobą fragmentu szablonu (można powiedzieć też, że obejmują sobą pusty fragment szablonu) - przy takich tagach nie używamy znacznika `{% endtag %}`. Przykładem takiego taga może być tag `debug`, którego używamy tak:

```
1  {% debug %}
```

Jeśli chcemy, żeby pewien fragment szablonu został wyrenderowany lub nie w zależności od pewnego warunku, używamy taga `if`:

```
1  {% if x == 7 %}  
2      raz  
3  {% elif x == 5 %}  
4      dwa  
5  {% else %}  
6      trzy  
7  {% endif %}
```

Blok `elif` może być powtórzony dowolną liczbę razy. Bloki `elif` i `else` można pominąć.

Warunkiem przy `ifie` może być przede wszystkim wyrażenie takie, jakiego możemy użyć w zapisie `{} zmienna []`. Warunek jest uznawany za spełniony, kiedy to wyrażenie ewaluuje się do prawdy, liczby innej niż zero, niepustej listy itp. - podobnie jak w pythonowym `ifie`. W warunkach można też używać konstrukcji, których w zwykłych wyrażeniach (pisanych w podwójnych klamrach) używać nie można - operatorów boolowskich, operatorów porównania, operatorów `in` i `not in` oraz filtrów. W warunkach nie można używać grupowania nawiasami. Oto kilka przykładów poprawnych i niepoprawnych (niepoprawne są przekreślone):

```
1  {% if x == 3 or x == 5 %} raz {% endif %}  
2  {% if x == "dominacja" %} raz {% endif %}  
3  {% if (x == 3) or (x == 5) %} raz {% endif %}  
4  {% if x == 3 or x|add:10 > 11 %} raz {% endif %}  
5  {% if x in lista %} raz {% endif %}  
6  {% if x in [3, 5, 7] %} raz {% endif %}
```

komentarze

Komentarze niewykraczające poza jedną linijkę zapisujemy tak: to nie jest komentarz {# to jest komentarz #} to znowu nie jest komentarz Komentarze mające wiele linii zapisujemy używając taga comment:

```
1 to nie jest komentarz
2 {% comment %}
3 to jest komentarz
4 to jest komentarz
5 to jest komentarz
6 {% endcomment %}
7 to znowu nie jest komentarz
```

Pliki statyczne

<https://docs.djangoproject.com/en/2.1/howto/static-files/> Strony internetowe oprócz treści samych stron potrzebują też dodatkowych dokumentów takich jak np. arkusze stylów, czy pliki javascript. Takie pliki nazywamy plikami statycznymi. Django dostarcza aplikację, która ułatwia zarządzanie nimi. Znajdziemy ją w django.contrib.staticfiles. 1. Aplikacja ta powinna się znaleźć w INSTALLED_APPS w settings. 2. W settings należy też dodać linię: STATIC_URL = '/static/'

3. W szablonie należy wtedy używać statyków np. w ten sposób:

```
1 {% load static %}
2 
```

4. Statyki umieszczamy w folderze static, np.:

my_app/static/my_app/example.jpg

Gdyby zaistniała potrzeba by w projekcie mieć pliki statyczne niezależne od konkretnej aplikacji, to można w głównym katalogu projektu utworzyć np. katalog static i do settings dodać

```
1 STATICFILES_DIRS = [
2     os.path.join(BASE_DIR, "static"),
3 ]
```

w liście tej można wskazać inne lokalizacje, w których mogą znajdować się pliki statyczne

Często - szczególnie w testach istnieje potrzeba wypełnienia bazy danych jakimiś początkowymi danymi. W Django można to zrobić np. w czasie migracji, albo przy użyciu fixtures. Fixtures to kolekcje danych, które Django będzie umiało zimportować do bazy. Kolekcje te są przechowywane jako XML, JSON lub YAML. Django umie serializować dane do tych formatów. Więcej o nich można przeczytać np. tu: <https://docs.djangoproject.com/en/2.1/topics/serialization/#serialization-formats>

Fixture

Tworzenie fixture

Dane z modelu możemy zapisać do fixture przy pomocy jednej z instrukcji manage.py

```
1 $ ./manage.py dumpdata
```

Można tam oczywiście przekazywać dodatkowe parametry, których opis znajdziemy w dokumentacji: <https://docs.djangoproject.com/en/2.1/ref/django-admin/#django-admin-dumpdata>

W naszym przykładzie wykonać możemy następujące kroki:

```
1 $ mkdir -p blog/fixtures  
2 $ python manage.py dumpdata --indent=4 blog > blog/fixtures/blog.json
```

w pliku blog.json znajdziemy wtedy zawartość podobną do tej:

```
1 [  
2 {  
3     "model": "blog.wpis",  
4     "pk": 1,  
5     "fields": {  
6         "tytul_wpisu": "raz dwa trzy",  
7         "tresc_wpisu": "Oto pierwszy testowy wpis. Raz dwa trzy, raz  
8             dwa trzy"  
9     }  
10    },  
11    {  
12        "model": "blog.wpis",  
13        "pk": 2,  
14        "fields": {  
15            "tytul_wpisu": "Cztery",  
16            "tresc_wpisu": "Oto drugi testowy wpis. Cztery cztery cztery."  
17        }  
18    }  
]
```

Wczytywanie fixture

Przygotowaną wcześniej fixture można załadować do bazy w następujący sposób:

```
1 $ python manage.py loaddata blog/fixtures/blog.json  
2 Installed 2 object(s) from 1 fixture(s)
```

Używanie fixture w testach

W czasie wykonywania testów Django korzysta z kopii bazy danych. W związku z tym dobrze by było mieć możliwość wypełnienia takie bazy danymi tak by testy mogły na nich bazować. Moglibyśmy np. chcieć sprawdzić czy na stronie prezentowana jest odpowiednia treść. Dodajmy więc do klasy TestBlog w tests.py nowy test oraz fixtures

```
1 class TestsBlog(TestCase):
2     fixtures = ['blog']
3     def test_main_page_of_app_exists(self):
4         response = self.client.get("/blog/")
5         self.assertEqual(response.status_code, 200)
6     def test_czy_strona_glowna_ma_poprawna_tresc(self):
7         response = self.client.get('/blog/')
8         self.assertContains(response, "<h1>raz trzy</h1>")
9         self.assertContains(response, "<div class='tresc_wpisu'>Oto")
10        self.assertContains(response, "<li>raz dwa trzy</li>")
11        self.assertContains(response, '<li>Cztery</li>')
```

I sprawdźmy testy:

```
1 $ python manage.py test blog
2 Creating test database for alias 'default'...
3 System check identified no issues (0 silenced).
4
5 ...
6 -----
7
8 Ran 3 tests in 0.024s
9
10 OK
11 Destroying test database for alias 'default'...
```

Ćwiczenie

1. Napisz szablon dla widoku index.
2. Przerób widok tak, żeby korzystał z szablonu.
3. Rozbuduj szablon tak, żeby oprócz spisu tematów pokazywał pełną treść jednego z wpisów. Dodać CSS, żeby strona miała ładny wygląd.
4. Utwórz drugi widok - ze szczegółami pierwszego wpisu
5. Spróbuj napisać test sprawdzający, czy widok zwraca odpowiednią treść

Ad. 1.

```
1 <ul>
2     {% for wpis in lista_wpisow %}
```

```
3     <li>{{ wpis.tytul_wpisu }}</li>
4     {% endfor %}
5 </ul>
```

Ad. 2.

```
1 # blog/views.py
2 from django.http import HttpResponse
3 from django.template import loader
4
5 # Create your views here.
6
7 from blog.models import Wpis
8 def index(request):
9     lista_wpisow = Wpis.objects.all()
10    t = loader.get_template('blog/index.html')
11    c = {'lista_wpisow': lista_wpisow}
12    wynik = t.render(c)
13    return HttpResponse(wynik)
```

Widok pokazujący wybrany wpis

W ćwiczeniach wykonałeś widok szczegółowy pierwszego wpisu. Warto by było jednak uogólnić tę sytuację i pokazać widok dowolnego wpisu. Dodatkowo można by wzbogacić widok listy o linki do takich widoków szczegółów. Widok pokazujący jeden wybrany wpis, wywoływany mógłby być URL-em:

```
1 http://localhost:8000/blog/wpis/
```

Musimy więc dopisać do pliku urls.py wzorzec, który spowoduje, że wejście pod ten adres spowoduje wywołanie widoku detail i przekaże mu - w parametrze id_wpisu - numer wpisu zjęty z URL-a. Jak pamiętamy, wzorce i ścieżki w urls.py mogą być wyrażeniami regularnymi. Jeśli chcemy, żeby część przypasowanego wzorca została przekazana widokowi jako argument, musimy tę część wzorca otoczyć nawiasami i podać nazwę, pod jaką zostanie przekazana. Na przykład taki wzorzec

```
1 tralala(?P<abcd>\d+)hopsasa
```

będzie łapał URL-e postaci

```
1 tralala<text>hopsasa
```

i znalezioną text przekazywał jako argument abcd

powyższego wzorca użylibyśmy w funkcji url w module urls.py. W nowszych wersjach Django używa się jednak raczej funkcji path. Tam wygląda to jeszcze prościej:

```
1 In[26]: p = path("tralala<abcd>hopsasa", print)
2 In[27]: p.resolve("tralala10hopsasa")
3 Out[27]: ResolverMatch(func=builtins.print, args=(), kwargs={'abcd': '10'}, url_name=None, app_names=[], namespaces=[])

```

Jesli chcemy, by ten fragment wzorca był liczbą, to możemy od razu wskazać, że ta zmienna część będzie typu `int`:

```
1 In[28]: p = path("tralala<int:abcd>hopsasa", print)
2 In[29]: p.resolve("tralala10hopsasa")
3 Out[29]: ResolverMatch(func=builtins.print, args=(), kwargs={'abcd': 10},
4 url_name=None, app_names=[], namespaces=[])

```

Zanim zaczniemy implementować nową funkcję dodajmy test, który upewni nas, że rozwiązanie wykonaliśmy poprawnie. Test mógłby wyglądać tak:

```
1 def test_czy_mozna_obejrzec_dowolny_wpis(self):
2     response = self.client.get('/blog/wpis/2')
3     self.assertContains(response, "<h1>Cztery</h1>")
4     self.assertContains(response, "<div class='trec_wpisu'>Oto drugi
testowy wpis. Cztery cztery cztery.</div>")
5     self.assertContains(response, '<li><a href="/blog/wpis/1">raz dwa
trzy</a></li>')
6     self.assertContains(response, '<li><a href="/blog/wpis/2">Cztery</
a></li>')

```

testujemy

```
1 $ python manage.py test blog
2 Creating test database for alias 'default'...
3 System check identified no issues (0 silenced).
4 .F..
5 =====
6 FAIL: test_czy_mozna_obejrzec_dowolny_wpis (blog.tests.TestsBlog)
7 -----
8 Traceback (most recent call last):
9 ...
10 AssertionError: 404 != 200 : Couldn't retrieve content: Response code
   was 404 (expected 200)
11 -----
12 -----
13 Ran 4 tests in 0.026s
14
15 FAILED (failures=1)
16 Destroying test database for alias 'default'...
```

No tak jest 404 a nie 200 - czyli musimy dodać path w urls

```
1 path('blog/wpisy/<int:id>', details)
```

Oczywiście trzeba też funkcję details zaimportować z views.py a więc i ją tam utworzyć:

```
1 def details(request, id):
2     lista_wpisow = Wpis.objects.all()
3     wybrany_wpis = lista_wpisow.get(pk=id)
4     t = loader.get_template('blog/detail.html')
5     c = {
6         'lista_wpisow': lista_wpisow,
7         'wybrany_wpis': wybrany_wpis
8     }
9     wynik = t.render(c)
10    return HttpResponseRedirect(wynik)
```

Jak myślisz, czy widok details może w tym przypadku korzystać z tego samego szablonu co index?

Panel Administratora

Jedną z ciekawszych zalet Django jest to, że potrafi on na podstawie zdefiniowanych modelu utworzyć Panel Administracyjny, w którym możemy dla naszych modeli wykonywać akcje CRUD (Create, Read, Update, Delete). Panel taki można stosunkowo szybko i łatwo dostosowywać do potrzeb projektu. Dzięki temu możemy zaoszczędzić sporo czasu potrzebnego na zbudowanie takiego panelu od początku. Te rozwiązanie jak każde ma też swoje wady, dopóki jednak trzymamy się w miarę standardowych rozwiązań wszystko idzie sprawnie. Aby Panel Administratora wiedział o modelu należy ten model zarejestrować w Panelu Administratora. Zanim jednak to zrobimy przypomnij sobie, że na początku tworzyliśmy konto administratora. To jest dobry moment by go użyć. Zauważ też, że w url.py od początku importowany jest moduł admin z django.contrib i dodana jest ścieżka: path('admin/', admin.site.urls), Zauważ też, że aplikacja django.contrib.admin dodana jest do INSTALLED_APPS w pliku settings.py. To jedna z kilku aplikacji, które są zainstalowane już na samym początku.

```
1 INSTALLED_APPS = [
2     'django.contrib.admin',
3     'django.contrib.auth',
4     'django.contrib.contenttypes',
5     'django.contrib.sessions',
6     'django.contrib.messages',
7     'django.contrib.staticfiles',
8     'blog',
9 ]
```

Na samym początku naszej przygody z Django - przy pierwszej migracji dostaliśmy informację o szeregu migracji, które zostały wykonane. To właśnie migracje z tych aplikacji. Wróćmy teraz do samego

panelu admina. Wcześniej utworzyliśmy konto administrator. Gdyby ten krok z jakichś względów został pominięty wystarczy, że wykonasz w terminalu taką komendę:

```
1 $ python manage.py createsuperuser
```

Logowanie do Panelu Administratora.

Zalogujmy się teraz do tego Panelu. By to zrobić trzeba wpisać w przeglądarce: <http://localhost:8000/admin>

Po wpisaniu nazwy użytkownika i hasła powinniśmy dostać widok podobny do tego:

Widoczna sekcja pochodzi z aplikacji 'django.contrib.auth', to dzięki niej mogliśmy utworzyć konto użytkownika. I jak widać możemy też tymi użytkownikami zarządzać. Tworzyć, dodawać, zmieniać. Dodatkowo jest tu też mechanizm nadawania uprawnień - czyli model Groups.

Rejestracja modelu w Panelu Administratora

Byśmy mogli zarządzać w ten sposób naszym modelem, należy go zarejestrować w Panelu administratora. Robi się to w pliku admin.py aplikacji.

```
1 # blog/admin.py
2 from django.contrib import admin
3 # Register your models here.
4 from blog.models import Wpis
5 admin.site.register(Wpis)
```

jeśli po tej zmianie wejdziemy na stronę, to zobaczymy lekko zmieniony widok:

Określenie sformułowania dla liczby mnogiej modelu

Nasz model został opisany jako Wpis - a nie Wpisy. Możemy to dość łatwo zmienić dodając w modelu specjalną klasę - Meta a w niej atrybut verbose_name_plural - zgodnie z poniższy:

```
1 # blog/models.py
2 from django.db import models
3 # Create your models here.
4 class Wpis(models.Model):
5     tytul_wpisu = models.CharField(max_length=200)
6     tresc_wpisu = models.TextField()
7     class Meta:
8         verbose_name_plural = "Wpisy"
```

Te dodatkowe s wzięto się stąd, że Django próbowało z naszego modelu zrobić liczbę mnogą, a tę w języku angielskim najczęściej robi się właśnie poprzez dodanie s

Zmiana wersji językowej Panelu Admina

Zauważmy też, że PA jest w języku angielskim. Gdybyśmy chcieli zmienić go tak by domyślnie był używany język polskim możemy w pliku settings.py zmienić linię

```
LANGUAGE_CODE = 'en-us'
```

na taką:

```
LANGUAGE_CODE = 'pl-pl'
```

jednoczesne edycje

Panel administracyjny nie zabezpiecza przed jednoczesnymi sprzecznymi edycjami. By poradzić sobie z tym problemem można jednak użyć różnych zewnętrznych modułów. Przykładowy opis takiego zadania znajdziemy pod adresem: <https://medium.com/@hakibenita/how-to-manage-concurrency-in-django-models-b240fed4ee2>

pliki statyczne

Uwaga: kiedy w settings.py ustawi się DEBUG = False, w panelu administracyjnym przestaje działać CSS. To normalne i związane z udostępnianiem plików statycznych - dokładniej jest to opisane w rozdziale o plikach statycznych. testy..

Po wykonaniu poprzednich zmian dobrze było by przygotować test, który sprawdzi np., że tylko zalogowani użytkownicy mogą wejść do panelu administracyjnego. Tak naprawdę do PA mogą wejść tylko Ci użytkownicy, którzy dodatkowo mają zaznaczoną flagę is_staff (w polskiej wersji: w zespole)

Na potrzeby tych testów musimy przygotować fixtures. Zwróć uwagę, że w przygotowanej fixture nie jest przechowywane Twoje hasło, ale jego hash

```
1 $ ./manage.py dumpdata --indent=4 auth > blog/fixtures/auth.json
```

Po tym przygotowaniu nasze testy mogą wyglądać następująco:

```
1 class TestsBlog(TestCase):
2     fixtures = ['blog', 'auth']
3     def test_main_page_of_app_exists(self):
4         response = self.client.get("/blog/")
5         self.assertEqual(response.status_code, 200)
6     def test_czy_strona_glowna_ma_poprawna_tresc(self):
7         response = self.client.get('/blog/')
8         self.assertContains(response, "<h1>raz dwa trzy</h1>")
9         self.assertContains(
10             response,
```

```
11         "<div class='tresc_wpisu'>Oto pierwszy testowy wpis. Raz  
12             dwa trzy, raz dwa trzy</div>"  
13     )  
14     self.assertContains(response, '<li><a href="/blog/wpisy/1">  
15         raz dwa trzy</a></li>')  
16     self.assertContains(response, '<li><a href="/blog/wpisy/2">  
17         Cztery</a></li>')  
18     def test_czy_mozna_obejrzec_dowolny_wpis(self):  
19         response = self.client.get('/blog/wpisy/2')  
20         self.assertContains(response, "<h1>Cztery</h1>")  
21         self.assertContains(  
22             response,  
23                 "<div class='tresc_wpisu'>Oto drugi testowy wpis. Cztery  
24                 cztery cztery.</div>")  
25         self.assertContains(response, '<li><a href="/blog/wpisy/1">  
26             raz dwa trzy</a></li>')  
27         self.assertContains(response, '<li><a href="/blog/wpisy/2">  
28             Cztery</a></li>')  
29     def test_niezalogowany_uzytkownik_nie_ma_dostepu_do_PA(self):  
30         response = self.client.get('/admin/')31         self.assertEqual(response.status_code, 302)  
32     def test_zalogowany_uzytkownik_ma_dostep(self):  
33         self.client.login(username='admin', password='123!@#qweQWE')  
34         response = self.client.get('/admin/')35         self.assertEqual(response.status_code, 200)
```

W teście:

```
1     def test_niezalogowany_uzytkownik_nie_ma_dostepu_do_PA(self):  
2         response = self.client.get('/admin/')3         self.assertEqual(response.status_code, 302)
```

Sprawdzamy, czy użytkownik został przekierowany do strony logowania. Ostatni test pokazuje, że zalogowany użytkownik z uprawnieniami admina może się zalogować.

Ćwiczenie Spróbuj dopisać test dla użytkownika który nie będzie adminem ani nie będzie w zespole

Dostosowywanie wyglądu panelu administracyjnego

PA daje spore możliwości konfiguracji. Możliwe jest też stosowanie zewnętrznych modułów, które zmieniają np. wygląd całego PA. Do bardziej znanych rozwiązań tego typu zalicza się np. Django Grappelli (<https://django-grappelli.readthedocs.io/en/latest/>) czy Django Suit (<https://djangosuit.readthedocs.io/en/develop/>). Te pakiety reorganizują wygląd panelu. Stylując go na nowo

i dodają różne nowe funkcjonalności. Póki co my pozostaniemy przy standardowym panelu i nauczymy się jak możemy dopasowywać go do naszych potrzeb. Więcej szczegółów o tych możliwościach można znaleźć w dokumentacji Django:

<http://docs.djangoproject.com/en/dev/ref/contrib/admin/#ref-contrib-admin>

Świetnym materiałem jest też ogólnodostępna książka:

<http://books.agiliq.com/projects/django-admin-cookbook/en/latest/>

Aby modyfikować wygląd i zachowanie naszych modeli w PA musimy utworzyć w pliku admin.py specjalną klasę, której zmiany będą odpowiadały za zmiany w PA Nasz admin.py po zmianach może wyglądać następująco:

```
1 # blog/admin.py
2 from django.contrib import admin
3 # Register your models here.
4 from blog.models import Wpis
5 class WpisAdmin(admin.ModelAdmin):
6     pass
7 admin.site.register(Wpis, WpisAdmin)
```

Ustawiając różne atrybuty klasy WpisAdmin możemy zmieniać zachowanie panelu administracyjnego.

przykrywanie szablonów panelu administracyjnego

Kiedy oglądamy w adminie listę obiektów modelu MójModel pochodzących z aplikacji mojaaplikacja, Django najpierw próbuje załadować szablon admin/mojaaplikacja/mojmodel/change_list.html (nazwa modelu, zauważmy, jest tu zapisana małymi literami). Jeśli go nie znajdzie, spróbuje załadować szablon admin/change_list.html. Ten drugi szablon istnieje zawsze - napisali go twórcy Django, zwykle znajdziemy go w katalogu:

```
1 <python>/site-packages/django/contrib/admin/templates/admin.
```

site-packages zależy od tego gdzie jest python którego używamy w projekcie.

W ten sposób możemy przesyłać wygląd szablonów panelu admina dla poszczególnych aplikacji. Jeśli chcemy zmienić np. wygląd listy dla wszystkich modeli ze wszystkich w aplikacji - czyli przesłonić ten szablon admin/change_list.html to powinniśmy wykonać kilka kroków: Do wygenerowanej na początku listy TEMPLATES

```
1 TEMPLATES = [
2     {
3         'BACKEND': 'django.template.backends.django.DjangoTemplates',
4         'DIRS': [],
5     },
6 ]
```

```
5      'APP_DIRS': True,
6      'OPTIONS': {
7          'context_processors': [
8              'django.template.context_processors.debug',
9              'django.template.context_processors.request',
10             'django.contrib.auth.context_processors.auth',
11             'django.contrib.messages.context_processors.messages',
12         ],
13     },
14   },
15 ]
```

Powinniśmy dodać w sekcji OPTIONS listę loaderów, oraz trzeba wskazać katalog w którym będą pliki w liście DIRS. Ważna jest tu też kolejność loaderów. Musimy też zakomentować lub usunąć sekcję 'APP_DIRS'. Ważne jest tu też utworzenie katalogu templates - w naszym przypadku na tym samym poziomie co plik manage.py, czyli w głównym folderze programu

```
1 TEMPLATES = [
2     {
3         'BACKEND': 'django.template.backends.django.DjangoTemplates',
4         'DIRS': [os.path.join(BASE_DIR, 'Templates')],  

5         # 'APP_DIRS': True,  

6         'OPTIONS': {
7             'loaders': [
8                 'django.template.loaders.filesystem.Loader',
9                 'django.template.loaders.app_directories.Loader'
10            ],
11            'context_processors': [
12                'django.template.context_processors.debug',
13                'django.template.context_processors.request',
14                'django.contrib.auth.context_processors.auth',
15                'django.contrib.messages.context_processors.messages',
16            ],
17        },
18    },
19 ]
```

Warto zauważyć w sposób w jaki podana została ścieżka w DIRS.

Teraz w nowym katalogu templates możemy utworzyć inny katalog - np. admin, a w nim np. plik change_list.html

Jeśli spróbujemy teraz zobaczyć listę naszych wpisów:

<http://localhost:8000/admin/blog/wpis/>

to okaże się, że nasza strona jest pusta - lub zawiera to co wpisałeś w nowym pliku. Można do niego skopiować zawartość odpowiedniego pliku, np.: django/contrib/admin/templates/admin/change_list.html

I odpowiednio go modyfikować. Możemy też przykryć szablon w jeden sposób dla wszystkich a w jakichś innym dla konkretnej aplikacji. Albo dla każdej aplikacji inny sposób. Wystarczy wtedy plik o odpowiedniej nazwie umieścić w odpowiednim miejscu w strukturze. Przykład: jeśli chcemy zmienić widok listy dla wszystkich modeli a dla aplikacji Blog też ale w inny sposób to odpowiedni fragment struktury templates powinna wyglądać w ten sposób:

```
1   ┌── templates
2     ├── admin
3     │   └── blog
4     │       ├── change_list.html
5     │       └── change_list.html
```

Innymi szablonami, które można w ten sposób przesyłać są np.: • app_index.html • actions.html
• change_form.html • change_form_object_tools.html • change_list.html • change_list_object_tools.html
• change_list_results.html • date_hierarchy.html • delete_confirmation.html • object_history.html
• pagination.html • popup_response.html • prepopulated_fields_js.html • search_form.html • submit_line.html

Więcej o tym przeczytać można w <https://docs.djangoproject.com/en/dev/ref/contrib/admin/#overriding-admin-templates>,

Tworzenie własnego szablonu dziedziczącego po gotowym dla wybranego modelu

Możemy też tego szablonu nie kopiować z katalogu templates aplikacji admin, a napisać go samemu dziedzicząc po szablonie admin/change_list.html. W tym celu w tym naszym szablonie admin/mojaaplikacja/mojmodel/change_list.html piszemy coś takiego:

```
1  {% extends "admin/change_list.html" %} 
2  {% block content %}
3    Tu żbyłem - Tkaczuk Józef.
4    {{ block.super }}
5  {% endblock %}
```

Kiedy teraz w adminie obejrzymy listę obiektów tego modelu tej aplikacji, zobaczymy nasz dopis. Możemy teraz modyfikować różne sekcje porównując to z blokami w pliku:

django/contrib/admin/templates/admin/change_list.html

Szablony w jednym folderze

Korzystając z powyższych ustawień możemy też teraz trzymać wszystkie szablony w tym głównym katalogu templates. Czasem taka forma zarządzania szablonami jest po prostu wygodniejsza. Wystarczy

skopiować zawartość katalogu templates z poszczególnych aplikacji do tego głównego templates.

tworzenie własnego szablonu dziedziczącego po gotowym dla wszystkich modeli Jeśli chcemy stworzyć własny szablon listy obiektów działający dla wszystkich modeli i nie chcemy tworzyć go przez skopiowanie gotowego szablonu z admina, ale bardziej elegancko - przez dziedziczenie po nim - to nie jest to trywialne. Szablon ten musi się bowiem nazywać admin/changelist.html i musi dziedziczyć po szablonie admin/changelist.html - ale nie powinien dziedziczyć po sobie samym. Ten problem rozwiążujemy tak, że w naszym katalogu z szablonami tworzymy link do oryginalnego szablonu admin/changelist.html i nazywamy go admin/changelist-oryginalny.html, a nasz szablon.html admin/changelist.html dziedziczy po admin/changelist-oryginalny.html. więcej

Więcej porad dotyczących dostosowywania wyglądu panelu administracyjnego znajdziemy pod adresem <http://lincolnloop.com/static/slides/2010-djangocon/customizing-the-admin.html> Choć sam artykuł trochę się zestarzał, to jednak pomysły nadal są aktualne. Czasem może być potrzebna jednak nieco inną drogą do jego osiągnięcia.

Drobne poprawki

Jeśli poklikamy trochę po naszym blogu, zauważmy jeszcze kilka drobiazłów wartych poprawienia:
1. w panelu administracyjnym w spisie wpisów każdy wpis opisany jest jako wpis object - wygodniej byłoby, gdybyśmy widzieli tu tytuł wpisu, 2. znaki nowej linii w treści bloga nie są zamieniane na znaczniki , 3. aplikacji blog nie byłoby łatwo przenieść do innego projektu - jej schemat URL-i jest zapisany w pliku należącym do projektu, 4. wpisy w blogu nie pamiętają daty wprowadzenia - przez to nie da się pokazać najnowszego wpisu, posortować wpisów na liście, 5. w szablonie w linkach są na sztywno wpisane URL-e - jeśli kiedyś zmienimy powiązanie między URL-ami a widokami, będziemy musieli wtropić i pozmieniać wszystkie takie miejsca, 6. kiedy próbujemy obejrzeć wpis, którego nie ma, dostajemy brzydkie błęd - ładniej byłoby, gdyby pojawiała się z strona z komunikatem 404, Naprawimy je po kolei.

Rzutowanie modelu na napis

Jak widzieliśmy, w panelu administracyjnym w spisie wpisów każdy wpis opisany jest jako wpis object. Jeśli chcemy, żeby używany był inny opis, musimy określić, jaki ma być wynik rzutowania obiektu klasy Wpis na napis. W ten sposób Wpis reprezentowany będzie także w terminalu. Oznacza to, że powinniśmy dodać do modelu metodę `__str__`:

```
1 class Wpis(models.Model):
2     ...
3     def __str__(self):
4         return self.tytul_wpisu
```

2. Formatowanie treści wpisów

W tej chwili treści wpisów nie da się formatować - nie da się nawet dzielić ich na akapity. Jeśli spróbujemy dopisywać znaczniki
, nie wyświetla się one, ponieważ domyślnie szablony Django eskejpują wstawiane zmienne. Możemy wyłączyć to eskejpowanie, ale zwykle nie chcemy tego, ze względu na bezpieczeństwo, robić. Możemy w takiej sytuacji użyć jakiejś zewnętrznej biblioteki, która pozwoli nam potraktowanie pola jako edytora markdown, albo wysiwyg. Przykłady takich rozwiązań można znaleźć na listach.

<https://djangopackages.org/grids/g/markdown/> <https://djangopackages.org/grids/g/wysiwyg/>

Zrobimy to trochę później, ponieważ do tego zadania może przydać się wiedza o formularzach.

3. Oddzielmy blog od projektu jazzy

Teraz aplikacja blog jest mocno związana z projektem jazzy - jej schemat URL-i jest zapisany w pliku należącym do projektu. W spisie urlpatterns w pliku urls.py po każdym wzorcu podajemy nazwę funkcji, która ma zostać wywołana. Zamiast tej nazwy możemy podać polecenie include(nazwa_modulu). Spowoduje to, że URL pasujący do wzorca spowoduje użycie kolejnych wzorców zdefiniowanych w określonym module. Wzorce te będą przypasowywane do tej części URL-a, która nie była przypasowana do pierwszego wzorca. Jeśli więc pierwszy wzorzec (zdefiniowany w urls.py projektu) będzie wyglądał tak: 'projekt/aplikacja/', include('aplikacja.urls') a drugi (zdefiniowany w projekt/aplikacja/urls.py) tak: 'siekiera', aplikacja.views.siekiera wejście pod URL <http://serwer/projekt/aplikacja/siekiera> spowoduje wywołanie widoku siekiera. Musimy więc: * w katalogu jazzy/blog stworzyć plik urls.py, * w nim zapisać wzorce dla aplikacji blog, * w pliku urls.py projektu jazzy umieścić wzorzec, który w reakcji na każdy URL prowadzący do /jazzy/blog dalsze sprawdzanie przekaże do urls.py w katalogu aplikacji blog.

```
1 # urls.py
2 from django.contrib import admin
3 from django.urls import path, include
4 urlpatterns = [
5     path('admin/', admin.site.urls),
6     path('blog/', include("blog.urls")),
7 ]
8 # blog/urls.py
9 from django.urls import path
10 from . import views
11 urlpatterns = [
12     path('wpisy/', views.index),
13     path('wpisy/<int:id>', views.details),
14 ]
```

4. Dodanie daty utworzenia

Dodanie daty utworzenia, czy zmiany to bardzo częsta potrzeba. W najprostszym sposobie można po prostu dodać do modelów nowe pole - tym razem typu DateTimeField i zadbanie o to by w czasie używania metody save zapisywana tam była aktualna data. W pierwszym kroku po prostu dodajmy do modelu dwa pola - created i modified:

```
1 # blog/models.py
2 from django.db import models
3 # Create your models here.
4 class Wpis(models.Model):
5     tytul_wpisu = models.CharField(max_length=200)
6     tresc_wpisu = models.TextField()
7     created = models.DateTimeField()
8     updated = models.DateTimeField()
9     def __str__(self):
10        return self.tytul_wpisu
11     class Meta:
12         verbose_name_plural = "Wpisy"
```

Po takiej zmianie trzeba przygotować i wykonać migracje. Oba pola są w naszym modelu polami wymagalnymi. W związku z tym, że istnieją już w naszej bazie wpisy system migracji zapyta nas w jaki sposób wypełnić dane dla istniejących rekordów. Cała operacja przedstawiona jest poniżej. Po jej wykonaniu otwórz migrację i zwróć uwagę, na subtelną różnicę między użyciem timezone.now i timezone.now()

```
1 $ ./manage.py makemigrations
2
3 You are trying to add a non-nullable field 'created'
4 to wpis without a default; we can't do that
5 (the database needs something to populate existing rows).
6 Please select a fix:
7 1) Provide a one-off default now (will be set on all existing
8    rows with a null value for this column)
9 2) Quit, and let me add a default in models.py
10 Select an option: 1
11 Please enter the default value now, as valid Python
12 The datetime and django.utils.timezone modules are available, so
    you can do e.g. timezone.now
13 Type 'exit' to exit this prompt
14 >>> timezone.now()
15 You are trying to add a non-nullable field 'updated' to wpis
16 without a default; we can't do that (the database needs something
    to populate existing rows).
17 Please select a fix:
18 1) Provide a one-off default now (will be set on all existing
19    rows with a null value for this column)
20 2) Quit, and let me add a default in models.py
21 Select an option: 1
22 Please enter the default value now, as valid Python
23 The datetime and django.utils.timezone modules are available, so
```

```
you can do e.g. timezone.now
24 Type 'exit' to exit this prompt
25 >>> timezone.now
26 Migrations for 'blog':
27     blog/migrations/0002_auto_20181214_2059.py
28         - Change Meta options on wpis
29             - Add field created to wpis
30             - Add field updated to wpis
```

Spójrzmy na przygotowaną migrację:

```
1 # Generated by Django 2.1.4 on 2018-12-14 20:59
2 import datetime
3 from django.db import migrations, models
4 from django.utils.timezone import utc
5 import django.utils.timezone
6 class Migration(migrations.Migration):
7     dependencies = [
8         ('blog', '0001_initial'),
9     ]
10    operations = [
11        migrations.AlterModelOptions(
12            name='wpis',
13            options={'verbose_name_plural': 'Wpisy'},
14        ),
15        migrations.AddField(
16            model_name='wpis',
17            name='created',
18            field=models.DateTimeField(
19                default=datetime.datetime(2018, 12, 14, 20, 58, 58,
20                                         961037, tzinfo=utc)
21            ),
22            preserve_default=False,
23        ),
24        migrations.AddField(
25            model_name='wpis',
26            name='updated',
27            field=models.DateTimeField(default=django.utils.timezone.
28                                         now),
29            preserve_default=False,
30        ),
31    ]
```

teraz należy wykonać migracje:

```
1 $ ./manage.py migrate
2 Operations to perform:
3   Apply all migrations: admin, auth, blog, contenttypes, sessions
4 Running migrations:
5   Applying blog.0002_auto_20181214_2059... OK
```

Obecnie przy tworzeniu / edycji wpisu trzeba będzie podawać datę:

Nie do końca o to nam chodziło. Chcemy, by te daty ustawiały się same. Przy czym data utworzenia powinna ustawić się tylko raz, zaś data modyfikacji powinna zmieniać się po każdym zapisaniu zmian. Po zmianie testu jednak nie wszystkie nasze testy przechodzą. Zmienić się też powinny nasze fixture. Zróbmy z tym porządek.

```
1 $ ./manage.py dumpdata --indent=4 blog > blog/fixtures/blog.json
```

Warto też dopisać nowy przypadek. Nasze po zrobieniu porządku mogłyby wyglądać teraz tak:

```
1 from django.contrib.auth import get_user_model
2 from django.test import TestCase
3 from blog.models import Wpis
4 # Create your tests here.
5 class TestsBlog(TestCase):
6     fixtures = ['blog', 'auth']
7     def test_main_page_of_app_exists(self):
8         response = self.client.get("/blog/wpisy/")
9         self.assertEqual(response.status_code, 200)
10    def test_czy_strona_glowna_ma_poprawna_tresc(self):
11        response = self.client.get('/blog/wpisy/')
12        self.assertContains(response, "<h1>raz dwa trzy</h1>")
13        self.assertContains(response,
14                            "<div class='tresc_wpisu'>Oto pierwszy
15                                testowy wpis. Raz dwa trzy, raz dwa trzy
16                                </div>")
17        self.assertContains(response, '<li><a href="/blog/wpisy/1">raz
18            dwa trzy</a></li>')
19        self.assertContains(response, '<li><a href="/blog/wpisy/2">
20            Cztery</a></li>')
21    def test_czy_mozna_obejrzec_dowolny_wpis(self):
22        response = self.client.get('/blog/wpisy/2/')
23        self.assertContains(response, "<h1>Cztery</h1>")
24        self.assertContains(response, "<div class='tresc_wpisu'>Oto
25            drugi testowy wpis. Cztery cztery cztery.</div>")
26        self.assertContains(response, '<li><a href="/blog/wpisy/1">raz
27            dwa trzy</a></li>')
28        self.assertContains(response, '<li><a href="/blog/wpisy/2">
29            Cztery</a></li>')
30    def test_niezalogowany_uzytkownik_nie_ma_dostepu_do_PA(self):
31        response = self.client.get('/admin/')
32        self.assertEqual(response.status_code, 302)
33    def test_zalogowany_uzytkownik_ma_dostep(self):
34        user = get_user_model().objects.get(pk=1)
35        self.client.force_login(user)
36        response = self.client.get('/admin/', follow=True)
37        self.assertEqual(response.status_code, 200)
38        self.assertContains(response, '<a href="/admin/blog/wpis/">
39            Wpisy</a>')
```

```
32
33
34 class TestWpistModel(TestCase):
35     def test_create_instance(self):
36         w1 = Wpis(
37             tytul_wpisu='raz dwa trzy',
38             tresc_wpisu='Oto pierwszy testowy wpis. Raz dwa trzy, raz
39             dwa trzy'
40         )
41         w1.save()
42         w = Wpis.objects.get(tytul_wpisu='raz dwa trzy')
43         self.assertEqual(w, w1)
44         w1.delete()
45     def test_created_and_modified_date(self):
46         w1 = Wpis(
47             tytul_wpisu="Trzeci",
48             tresc_wpisu="ścTre trzeciego"
49         )
50         w1.save()
51         created = w1.created
52         updated = w1.updated
53         w1.tresc_wpisu = "Poprawiona"
54         w1.save()
55         self.assertEqual(created, w1.created)
56         self.assertNotEqual(updated, w1.updated)
```

i uruchomienie testów:

```
1 $ ./manage.py test
2 Creating test database for alias 'default'...
3 System check identified no issues (0 silenced).
4 .....
5 -----
6 Ran 7 tests in 0.103s
7
8 OK
9 Destroying test database for alias 'default'...
```

W zasadzie w większości modeli, które możemy mieć w naszym projekcie przydałby się taki timestamp.. W związku z tym można by przygotować klasę po której dziedziczyćby nasze modele, wszędzie tam, gdzie byłoby to potrzebne.

1. Utwórzmy klasę bazową:

```
1
2 # blog/models.py
3 from django.db import models
4 # Create your models here.
5 class TimeStampedModel(models.Model):
6     """
```

```
7     An abstract base class model that provides self-updating
8     `created` and `modified` fields.
9     """
10    created = models.DateTimeField(auto_now_add=True)
11    updated = models.DateTimeField(auto_now=True)
12    class Meta:
13        abstract = True
14    class Wpis(TimeStampedModel):
15        tytul_wpisu = models.CharField(max_length=200)
16        tresc_wpisu = models.TextField()
17        def __str__(self):
18            return self.tytul_wpisu
19        class Meta:
20            verbose_name_plural = "Wpisy"
```

2. W związku z tym, że klasa ta może być używana w wielu modelach wyłączmy ją z models.py naszej aplikacji. Utwórzmy w głównym katalogu projektu package lib a w niej plik model_utils.py

```
1 # blog/models.py
2 from django.db import models
3 from lib.model_utils import TimeStampedModel
4 class Wpis(TimeStampedModel):
5     tytul_wpisu = models.CharField(max_length=200)
6     tresc_wpisu = models.TextField()
7     def __str__(self):
8         return self.tytul_wpisu
9     class Meta:
10        verbose_name_plural = "Wpisy"
11 # lib/model_utils.py
12 from django.db import models
13 # Create your models here.
14 class TimeStampedModel(models.Model):
15     """
16     An abstract base class model that provides self-updating
17     `created` and `modified` fields.
18     """
19     created = models.DateTimeField(auto_now_add=True)
20     updated = models.DateTimeField(auto_now=True)
21     class Meta:
22         abstract = True
```

Pola tylko do odczytu

Zauważmy, że pola dat zniknęły z panelu administratora. Dobrze by było móc je jednak zobaczyć. Wystarczy, że w pliku admin.py dodamy listę: `readonly_fields = ['created', 'updated']`

Wyświetlenie kilku kolumn w widoku listy

Widok listy jest póki co dość mało użyteczny. Dodajmy tam kolumny created i updated - tak by łatwo było określić, które wpisy są najnowsze. To kolejna edycja pliku admin i dodanie kolejnej listy:

```
1 # blog/admin.py
2 from django.contrib import admin
3 # Register your models here.
4 from blog.models import Wpis
5 class WpisAdmin(admin.ModelAdmin):
6     readonly_fields = ['created', 'updated']
7     list_display = ['tytul_wpisu', 'created', 'updated']
8
9 admin.site.register(Wpis, WpisAdmin)
```

uwzględnianie dat wpisów przy ich wyświetlaniu

Skoro teraz każdy wpis ma zapisaną datę powstania, możemy to uwzględnić w widokach index i detail. Możemy: * wypisując listę tytułów wpisów posortować ją po datach, * w widoku index wypisywać pełną treść nie widoku pierwszego z brzegu, a najnowszego. Do tego przydadzą się metody order_by() i latest() klasy QuerySet: <http://docs.djangoproject.com/en/dev/ref/models/querysets/#order-by-fields> <http://docs.djangoproject.com/en/dev/ref/models/querysets/#latest-field-name-none>
Ćwiczenie

1. Przerób widoki tak, żeby przy każdym wpisie była widoczna jego data.
2. Przerób widoki tak, żeby: • w widoku index pokazywana była pełna treść najnowszego wpisu, • wpisy w liście wpisów były posortowane od najnowszego do najstarszego, • na liście wpisów nie było pokazywanych więcej niż 15 wpisów.

Linki wpisane na sztywno w szablon

Jak już zauważyliśmy, w szablonie bloga mamy na sztywno wpisany URL do pełnej treści wpisu. Jeśli kiedyś zechcemy zmienić powiązanie między URL-ami a widokami, będziemy musieli wytropić i pozmieniać wszystkie takie miejsca. Na szczęście w djangowym mechanizmie szablonów istnieje wygodny tag `{% url %}`. Pozwala on określić tylko do którego widoku chcemy przejść i jak mają być ustalone argumenty. Django przejrzzy wtedy pliki urls.py, znajdzie wzorzec prowadzący do danego widoku i na jego podstawie wygeneruje URL. Taga tego używa się tak: `{% url "aplikacja.views.widok" nazwa_argumentu=wartość %}` Spróbujmy teraz tak zmienić nasze szablony by uwzględniały tę funkcjonalność:

```
1 # blog/urls.py
2 from django.urls import path
3 from . import views
```

```
4     urlpatterns = [
5         path('wpisy/', views.index),
6         path('wpisy/<int:id>', views.details, name='wpisy-details'),
7     ]
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
```

```
1  {%# index.html %}
2  {% load static %}
3  <!DOCTYPE html>
4  <html>
5      <head>
6          <title>{{wybrany_wpis.tytul_wpisu}}</title>
7          <link rel="stylesheet" href="{% static 'blog/style.css' %}" />
8      </head>
9      <body>
10         <div class='main'>
11             <div class='wpis'>
12                 <h1>{{wybrany_wpis.tytul_wpisu}}</h1>
13                 <div class='trec_wpisu'>{{wybrany_wpis.trec_wpisu}}</div>
14             </div>
15         </div>
16         <div class='boczek'>
17             <div class='lista_wpisow'>
18                 <ul>
19                     {% for wpis in lista_wpisow %}
20                         <li>
21                             <a href="{% url 'wpisy details' id=wpis.pk %}">
22                                 {{ wpis.tytul_wpisu }}
23                             </a>
24                         </li>
25                     {% endfor %}
26                 </ul>
27             </div>
28         </div>
29     </body>
30 </html>
```

Próba pobrania wpisu, którego nie ma

Co się stanie, gdy ktoś spróbuje wejść np. na adres:

```
1  /blog/wpisy/4
```

Obecnie dostaniemy komunikat błędu:

Jeśli w settings ustawimy DEBUG=False,

Spróbujmy jakoś tę sytuację obsłużyć. Moglibyśmy po prostu w widoku użyć `try ...except`.

```
1 from django.http import HttpResponse, Http404
2
3 def details(request, id):
4
5     lista_wpisow = Wpis.objects.all()
6     try:
7         wybrany_wpis = Wpis.objects.get(pk=id)
8     except Wpis.DoesNotExist:
9         raise Http404
10
11    t = loader.get_template('blog/index.html')
12    c = {
13        'lista_wpisow': lista_wpisow,
14        'wybrany_wpis': wybrany_wpis
15    }
16    wynik = t.render(c)
17    return HttpResponse(wynik)
```

Po takiej zmianie otrzymamy już błąd 404 a nie 500

Możemy w templates utworzyć szablon 404.html - tak by jeszcze później obsługiwać ten błąd

Na skróty

get_object_or_404 Django posiada mechanizmy, dzięki którym to, co napisaliśmy do tej pory, da się napisać krócej, często powtarzające się idiomy zapisując jednym skrótnym poleceniem. W aplikacjach internetowych często powtarza się fragment:

- pobierz (metoda get()) wiersz z bazy danych,
- jeśli takiego wiersza nie ma, rzuć wyjątek 404.

Można to zrobić jednym poleceniem `get_object_or_404()`. Jest ono zdefiniowane w module `django.shortcuts`. Jako argumenty podajemy mu nazwę modelu i warunek - taki sam jak ten podawany metodzie `get()`. Jeśli wiersz uda się znaleźć, zwracany jest reprezentujący go obiekt. Jeśli nie uda się go znaleźć, rzucany jest wyjątek 404.

```
1
2 from django.shortcuts import render, get_object_or_404
3 def details(request, id):
4     lista_wpisow = Wpis.objects.all()
5     wybrany_wpis = get_object_or_404(Wpis, pk=id)
6     t = loader.get_template('blog/index.html')
7     c = {
8         'lista_wpisow': lista_wpisow,
9         'wybrany_wpis': wybrany_wpis
10    }
```

```
11     wynik = t.render(c)
12     return HttpResponseRedirect(wynik)
```

Ćwiczenie:

Sprawdź czy testy działają poprawnie. Popraw je jeśli jest taka potrzeba. `render_to_response`

Innym skrótem z `django.shortcuts` jest `render_to_response()`. Jest to funkcja, której podajemy dwa argumenty: nazwę pliku z szablonem i słownik ze zmiennymi, które mają być wstawione w szablon. W odpowiedzi dostajemy gotowy do zwrócenia obiekt klasy `HttpResponse`.

Ćwiczenie:

Zmień widok `index` tak by korzystał ze skrótu `render_to_response`

```
1 from django.shortcuts import get_object_or_404, render_to_response
2 # blog/views.py
3 from django.http import HttpResponseRedirect
4 from django.template import loader
5 # Create your views here.
6 from blog.models import Wpis
7
8 def index(request):
9     lista_wpisow = Wpis.objects.all()
10    wybrany_wpis = lista_wpisow[0]
11    c = {
12        'lista_wpisow': lista_wpisow,
13        'wybrany_wpis': wybrany_wpis
14    }
15    return render_to_response('blog/index.html', context=c)
16
17 def details(request, id):
18     lista_wpisow = Wpis.objects.all()
19     wybrany_wpis = get_object_or_404(Wpis, pk=id)
20     t = loader.get_template('blog/index.html')
21     c = {
22         'lista_wpisow': lista_wpisow,
23         'wybrany_wpis': wybrany_wpis
24     }
25     wynik = t.render(c)
26     return HttpResponseRedirect(wynik)
```

Stronicowanie, czyli paginacja

Jeśli naszych Wpisów będzie więcej więc dobrze było by móc podzielić je na strony po kilka Wpisów. Django udostępnia mechanizm stronicowania do obsługi takich problemów. Spójrzmy jak to działa:

```
1 >>> from django.core.paginator import Paginator
2 >>> objects = ['john', 'paul', 'george', 'ringo']
3 >>> p = Paginator(objects, 2)
4 >>> p.count
5 4
6 >>> p.num_pages
7 2
8 >>> type(p.page_range)
9 <class 'range_iterator'>
10 >>> p.page_range
11 range(1, 3)
12 >>> page1 = p.page(1)
13 >>> page1
14 <Page 1 of 2>
15 >>> page1.object_list
16 ['john', 'paul']
17 >>> page2 = p.page(2)
18 >>> page2.object_list
19 ['george', 'ringo']
20 >>> page2.has_next()
21 False
22 >>> page2.has_previous()
23 True
24 >>> page2.has_other_pages()
25 True
26 >>> page2.next_page_number()
27 Traceback (most recent call last):
28 ...
29 EmptyPage: That page contains no results
30 >>> page2.previous_page_number()
31 1
32 >>> page2.start_index() # The 1-based index of the first item on this
   page
33 3
34 >>> page2.end_index() # The 1-based index of the last item on this page
35 4
36 >>> p.page(0)
37 Traceback (most recent call last):
38 ...
39 EmptyPage: That page number is less than 1
40 >>> p.page(3)
41 Traceback (most recent call last):
42 ...
43 EmptyPage: That page contains no results
```

Wykorzystajmy ten mechanizm do listy naszych wpisów. Po pierwsze wprowadźmy zmiany do funkcji widoku:

```
1 # blog/views.py
2 from django.core.paginator import Paginator
3 from django.shortcuts import get_object_or_404, render_to_response
4 from django.http import HttpResponseRedirect
5 from django.template import loader
6 # Create your views here.
7 from blog.models import Wpis
8 def index(request):
9     lista_wpisow = Wpis.objects.all()
10    wybrany_wpis = lista_wpisow[0]
11    paginator = Paginator(lista_wpisow, 1) # żpoka 1 wpisie na stronie
12    page = request.GET.get('page')
13    wpisy = paginator.get_page(page)
14    c = {
15        'lista_wpisow': wpisy,
16        'wybrany_wpis': wybrany_wpis
17    }
18    return render_to_response('blog/index.html', context=c)
```

Zauważmy, że funkcja ta spodziewa się w metodzie GET parametru page odpowiadającego numerowi strony. Po drugie wprowadźmy zmiany w szablonie dodając sekcję odpowiedzialną za paginację w div klasie „boczek”

```
1 {# index.html #-}
2 {% load static %} 
3 <!DOCTYPE html>
4 <html>
5 <head>
6     <title>{{ wybrany_wpis.tytul_wpisu }}</title>
7     <link rel="stylesheet" href="{% static 'blog/style.css' %}" />
8 </head>
9 <body>
10 <div class='main'>
11     <div class='wpis'>
12         <h1>{{ wybrany_wpis.tytul_wpisu }}</h1>
13         <div class='tresc_wpisu'>{{ wybrany_wpis.tresc_wpisu }}</div>
14     </div>
15 </div>
16 <div class='boczek'>
17     <div class='lista_wpisow'>
18         <ul>
19             {% for wpis in lista_wpisow %}
20                 <li><a href="{% url 'wpisy-details' id=wpis.pk %}">{{ wpis.tytul_wpisu }}</a></li>
21             {% endfor %}
22         </ul>
23     </div>
```

```
24     <div class="pagination">
25         <span class="step-links">
26             {% if lista_wpisow.has_previous %}
27                 <a href="?page=1">&laquo; first</a>
28                 <a href="?page={{ lista_wpisow.previous_page_number }}">
29                     >previous</a>
30             {% endif %}
31             <span class="current">
32                 Page {{ lista_wpisow.number }} of {{ lista_wpisow.
33                     paginator.num_pages }}.
34             </span>
35             {% if lista_wpisow.has_next %}
36                 <a href="?page={{ lista_wpisow.next_page_number }}">
37                     next</a>
38                 <a href="?page={{ lista_wpisow.paginator.num_pages }}">
39                     last &raquo;</a>
40             {% endif %}
41         </span>
42     </div>
43 </body>
44 </html>
```

Więcej informacji o paginacji: <https://docs.djangoproject.com/pl/2.1/topics/pagination/>

Relacje pomiędzy obiektami

Nowy model - Tagi

Przydałoby się dodać możliwość oznaczania wpisów tagami. W tym celu musimy: * stworzyć model reprezentujący tag, * dodać do modelu Wpis kolumnę z kluczem obcym do modelu tag, * uzupełnić widoki i szablony tak, by przy wpisach podawane były ich tagi. Aby stworzyć kolumnę zawierającą klucz obcy z połączeniem wiele do wielu definiujemy kolumnę jako ManyToManyField. Konstruktorowi podajemy jeden obowiązkowy argument - nazwę modelu, do którego łączymy. Choć w bazie danych to połączenie będzie reprezentowane przez pośrednią tabelkę, my nie musimy się nią zajmować. Przy połączeniu wiele do wielu klucz obcy definiujemy tylko w jednym modelu - obojętnie którym. Zwykle jednak definiujemy go w tym modelu, który będziemy edytować w panelu administracyjnym. Przykładowo, gdybyśmy chcieli pamiętać wykłady, studentów i ich powiązanie, moglibyśmy stworzyć takie dwa modele:

```
1  class Wyklad(models.Model):
2      tytul = models.CharField(max_length=200)
3  class Student(models.Model):
4      imie = models.CharField(max_length=200)
5      nazwisko = models.CharField(max_length=200)
6      wykłady = models.ManyToManyField(wykład)
```

Ćwiczenie:

Stwórzmy sobie na szybko pomocniczy projekt, który pomoże nam zrozumieć te zagadnienie.

1. Przejdź do swojego głównego folderu z projektami - np. workspace, utwórz wirtualne środowisko, zainstaluj django i utwórz nowy projekt

```
1 $ cd ~/workspace
2 $ mkvirtualenv uczelnia
3 (uczelnia) $ pip install django
4 (uczelnia) $ django-admin startproject uczelnia
5 (uczelnia) $ cd uczelnia
6 (uczelnia) $ ./manage.py startapp studenci
7 (uczelnia) $ tree
8 .
9   manage.py
10  studenci/
11    admin.py
12    apps.py
13    __init__.py
14    migrations/
15      __init__.py
16    models.py
17    tests.py
18    views.py
19  uczelnia/
20    __init__.py
21    __pycache__/
22      __init__.cpython-37.pyc
23      settings.cpython-37.pyc
24    settings.py
25    urls.py
26    wsgi.py
```

Wykonaj migracje, dodaj aplikację do settings, przygotuj modele, przygotuj migrację i ponownie je wykonaj, a następnie uruchom

```
1 $ python manage.py shell
2
3     >>> from studenci.models import Wyklad, Student
4     >>> w = Wyklad(tytul='astronawigacja')
5     >>> w.save()
6     >>> w2 = Wyklad(tytul='nawigacja')
```

```
5     >>> w2.save()
6     >>> s = Student(imie='Piotr', nazwisko='Sobolewski')
7     >>> s.save()
8     >>> s.wyklady.add(w)
9     >>>
```

Zauważmy, że żeby połączyć wykład i studenta obaj muszą być zapisani (metodą save()) - inaczej nie mieliby jeszcze id, więc nie byłoby wiadomo, co wpisać w tabelkę pomocniczą. Zauważmy, że po wydaniu polecenia s.wyklady.add(w) nie musimy wołać save ani na wykładzie, ani na studencie - samo polecenie s.wyklady.add(w) zapisuje co trzeba w bazie danych. Obiektowe API dostępu do bazy danych ułatwia przeglądanie takich połączonych tabelek. W atrubucie wyklady modelu student mamy menadżera pozwalającego w zwykły sposób wybierać wykłady związane z danym studentem. Na przykład polecenie:

```
1 >>> s.wyklady.all()
2 <QuerySet [<<wyklad: wyklad object (1)>]>
```

zwróci wszystkie wykłady związane z danym studentem.

Przy wyszukiwaniu wiersza w bazie danych oprócz warunków dotyczących kolumn tej tabelki możemy też używać warunków na kolumnach tej drugiej, powiązanej tabelki. To znaczy, że w naszym przykładzie przy wybieraniu studentów możemy nakładać warunki nie tylko na kolumny modelu student, ale też na kolumny wykładów powiązanych z danym studentem. Możemy więc na przykład wybrać wszystkich studentów uczęszczających na wykłady z astronawigacji:

```
1 student.objects.filter(wyklady__tytuł='astronawigacja')
```

Zapis wyklady__tytuł (zauważmy - rozdzielony podwójnym podkreśleniem) oznacza kolumna tytuł w wierszach obcej tabelki powiązanych przez klucz obcy wykłady.

Ćwiczenia

1. Stwórz model Tag. Zrób, żeby był on dostępny w panelu administracyjnym. Do modelu Wpis dodaj klucz obcy wiele do wielu łączący wpisy z tagami. Stwórz nowe tabelki w bazie danych.

```
1 # blog/models.py
2 from django.db import models
3 from lib.model_utils import TimeStampedModel
4
5 class Tag(models.Model):
6     nazwa_taga = models.CharField(max_length=200)
7     def __str__(self):
8         return self.nazwa_taga
9     class Meta:
10         verbose_name_plural = 'tagi'
```

```
11 class Wpis(TimeStampedModel):
12     ...
13     tagi_wpisu = models.ManyToManyField(Tag, blank=True)
14     ...
15     ...
```

I rejestrujemy nasz model w Panelu Administratora. Tym razem zrobimy to przy użyciu dekoratora. Efekt jest taki sam:

```
1 # blog/admin.py
2 from django.contrib import admin
3 # Register your models here.
4 from blog.models import Wpis, Tag
5 ...
6 @admin.register(Tag)
7 class TagAdmin(admin.ModelAdmin):
8     pass
```

W czasie edycji Wpisów w PA będziemy teraz widzieć dodatkowe pole:

Na razie nie jest jeszcze utworzony żaden tag. Możemy go utworzyć już z tego poziomu - klikając w przycisk +. Nowo utworzony Tag powinien być jednocześnie zaznaczony. Jeśli jakieś taki istniały już, to wystarczy je tutaj zaznaczyć.

Django tworzy tabelę pośrednią w której zapisywane są połączenia między Tag i Wpis. Działa to mniej więcej tak:

```
1 >>> from blog.models import Wpis, Tag
2 >>> w = Wpis.objects.last()
3 >>> w
4 <Wpis: Wpis nr 19>
5 >>> w.tagi_wpisu
6 <django.db.models.fields.related_descriptors.
    create_forward_many_to_many_manager.
7 <locals>.ManyRelatedManager object at 0x7f6aba178d68>
8 >>> w.tagi_wpisu.all()
9 <QuerySet [<Tag: Definicja>]>
10 >>> Wpis.tagi_wpisu.through
11 <class 'blog.models.Wpis_tagi_wpisu'>
12 >>> pierwszy = Wpis.objects.first()
13 >>> pierwszy.tagi_wpisu.all()
14 >>> <QuerySet []>
15 >>> tag = Tag.objects.last()
16 >>> tag
17 <Tag: Definicja>
18 >>> polaczenie = Wpis.tagi_wpisu.through()
19 >>> polaczenie.tag = tag
20 >>> polaczenie.wpis = pierwszy
21 >>> polaczenie.save()
22 >>> pierwszy.tagi_wpisu.all()
```

```
23 <QuerySet [<Tag: Definicja>]>
```

Dodatkowo - poprzez dodanie do WpisAdmin listy filter_horizontal, możemy uzyskać nieco ciekawszą reprezentację wyboru tagów

```
1 class WpisAdmin(admin.ModelAdmin):
2     readonly_fields = ['created', 'updated']
3     list_display = ['tytul_wpisu', 'created', 'updated']
4     filter_horizontal = ("tagi_wpisu",)
```

Ćwiczenia:

1. W panelu administracyjnym stwórz kilka nowych wpisów na blogu.
2. W trybie interaktywnym spróbuj znaleźć wszystkie tagi danego wpisu oraz wszystkie wpisy związane z danym tagiem.
3. W trybie interaktywnym stwórz nowy tag i oznacz nim jeden z wpisów.
4. Przerób widoki i szablony tak, żeby przy wpisie, który wyświetlony jest z pełną treścią, wypisane były wszystkie jego tagi.
5. Niech kliknięcie na nazwę taga prowadzi do nowego widoku, który pokaże wszystkie wpisy związane z danym tagiem.

Ad. 4

```
1 # blog/views.py
2 ...
3 def index(request):
4     lista_wpisow = Wpis.objects.all().order_by('-updated')
5     wybrany_wpis = lista_wpisow.latest()
6     tagi = wybrany_wpis.tagi_wpisu.all()
7     paginator = Paginator(lista_wpisow, 15) # pokaz 15 wpisów na stronie
8     page = request.GET.get('page')
9     wpisy = paginator.get_page(page)
10    c = {
11        'lista_wpisow': wpisy,
12        'wybrany_wpis': wybrany_wpis,
13        'tagi': tagi,
14    }
15    return render_to_response('blog/index.html', context=c)
```

```
1 {# blog/index.html #}
2 <div class='main'>
3     <div class='wpis'>
4         <h1>{{ wybrany_wpis.tytul_wpisu }}</h1>
5         <div class='tresc_wpisu'>{{ wybrany_wpis.tresc_wpisu }}</div>
```

```

6      <div class='data_wpisu'>{{ wybrany_wpis.created|date:"Y.m.d H:i
7          " }}</div>
8      <div id='tagi_wpisu'>Tagi:
9          <ul>
10         {% for tag in tagi %}
11             <li>{{ tag.nazwa_taga }}</li>
12         {% endfor %}
13     </ul>
14 </div>
15 </div>
```

Ad. 5.

Powinniśmy dodać nową ścieżkę w blog/urls.py

```
1 path('wpisy/tagi/<nazwa_taga>', views.tagi, name='wpisy-po-tagu')
```

Jak widać powinna ona prowadzić do akcji widoku views.tagi. Dodajemy tę akcję:

```

1 # blog/views.py
2 ...
3 def tagi(request, nazwa_taga):
4     wybrane_wpisy = Wpis.objects.filter(tagi_wpisu__nazwa_taga=
5         nazwa_taga)
6     return render_to_response(
7         'blog/wpisy_po_tagu.html',
8         {
9             'nazwa_taga': nazwa_taga,
10            'wybrane_wpisy': wybrane_wpisy,
11        })
```

w pliku szablonu blog/index.html chcemy wprowadzić zmiany prowadzące do takiego widoku. W pętli iterującej po tagach

```
1 <li><a href="{% url 'wpisy-po-tagu' nazwa_taga=tag.nazwa_taga %}">{{ tag.nazwa_taga }}</a></li>
```

Jak widzimy w tagu url używamy name z funkcji path dla tej ścieżki.

Pozostaje jeszcze stworzyć odpowiedni szablon:

Zanim go napiszemy zainstalujemy sobie jeszcze pierwszy dodatek: django_markup

<https://github.com/bartTC/django-markup>

Postugując się instrukcją zainstaluj tę aplikację i dodaj ją do settings. A następnie dodaj nowy szablon:

```
1 {# blog/wpisy_po_tagu.html #}
```

```
2  {% load markup_tags static %} 
3  <html>
4  <head>
5      <title>{{ wybrany_wpis.tytul_wpisu }}</title>
6      <link rel="stylesheet" href="{% static 'blog/style.css' %}" />
7  </head>
8  <body>
9  <div class='main'>
10     wpisy oznaczone jako <i>{{ nazwa_taga }}</i>
11     <div class='lista_wpisow'>
12         {% for wpis in wybrane_wpisy %}
13             <div class='wpis'>
14                 <h1><a href="{% url 'wpisy-details' id=wpis.id %}">{{ wpis.tytul_wpisu }}</a></h1>
15                 <div class='trec_wpisu'>
16                     {{ wpis.trec_wpisu|apply_markup:"markdown" }}
17                 </div>
18             </div>
19         {% endfor %}
20     </div>
21 </div>
22 <div class='boczek'>
23     <div class='lista_wpisow'>
24         <ul>
25             {% for wpis in lista_wpisow %}
26                 <li><a href="{% url 'wpisy-details' id=wpis.id %}">{{ wpis.tytul_wpisu }}</a></li>
27             {% endfor %}
28         </ul>
29     </div>
30 </div>
31 </body>
32 </html>
```

Po wejściu na adres:

<http://localhost:8000/blog/wpisy/tagi/Definicja>

Powinniśmy zobaczyć coś podobnego do tego:

Dziedziczenie szablonów

Zauważmy, że teraz oba szablony - pokazujący treść wybranego wpisu i pokazujący wpisy związane z danym tagiem - są bardzo podobne, dużą część kodu mają wspólną. W Django w takiej sytuacji możemy stworzyć jeden szablon zawierający wspólną część kodu, a następnie dodać dwa dziedziczące po nim szablony nadpisujące wybrane bloki. W szablonie można zaznaczyć dowolny fragment i nadać mu nazwę:

```
1 (...)  
2  
3 <div class='opis_produktu'>  
4 {% block produkt %}  
5 tu  ebdzie opis produktu  
6 {% endblock produkt %}  
7 </div>  
8 (...)
```

Nast epnie mo na stworzy  drugi szablon, dziedziczący po nim i nadpisuj cy wybrane sekcje rodzica:

```
1 {% extends "sklep/prezentacja_produktu.html" %}  
2 {% block produkt %}  
3 Wybrany produkt: {{ produkt.nazwa }}  
4 Cena: {{ produkt.cena }}  
5 {% endblock produkt %}
```

Przy dyrektywie `endblock` mo na (jak w powy szym przyk adzie), ale nie trzeba podawa  nazwy ko czonego bloku.

Je li szablon rodzicielski  aduje dodatkow  bibliotek  (poleciem `{% load biblioteka %}`), nie spowoduje to,  e za aduj  j  szablony dziedzic ce po nim - w ka dym dziecku z osobna r wnie  trzeba doda  polecenie `load`.

 wiczenia:

1. Przer b szablonu tak, by korzysta  z dziedziczenia.

Przorganizuj my szablony w nast puj cy spos b:

```
1 templates/|  
2   about.html|  
3   admin| |  
4     blog|| |  
5       change_list.html| |  
6       change_list.html| |  
7     base.html| |  
8   blog| |  
9     index.html| |  
10    wpisy_po_tagu.html
```

`blog/index.html` i `blog/wpisy_po_tagu.html` b  d  dziedziczy  po `base.html`

```
1 {% # base.html #}  
2 {% load static %}
```

```
3 <html>
4 <head>
5     <title>{% block title %}Strona główna{% endblock %}</title>
6     <link rel="stylesheet" href="{% static 'blog/style.css' %}" />
7 </head>
8 <body>
9 <div class='main'>
10    {% block main %}
11    {% endblock %}
12 </div>
13 <div class='boczek'>
14     <div class='lista_wpisow'>
15         <ul>
16             {% for wpis in lista_wpisow %}
17                 <li><a href="{% url 'wpisy-details' id=wpis.pk %}">{{ wpis.tytul_wpisu }}</a></li>
18             {% endfor %}
19         </ul>
20     </div>
21     {% if lista_wpisow %}
22         <div class="pagination">
23             <span class="step-links">
24                 {% if lista_wpisow.has_previous %}
25                     <a href="?page=1">&lquo; first</a>
26                     <a href="?page={{ lista_wpisow.previous_page_number }}">previous</a>
27                 {% endif %}
28                 <span class="current">
29                     Page {{ lista_wpisow.number }} of {{ lista_wpisow.paginator.num_pages }}.
30                 </span>
31                 {% if lista_wpisow.has_next %}
32                     <a href="?page={{ lista_wpisow.next_page_number }}">next</a>
33                     <a href="?page={{ lista_wpisow.paginator.num_pages }}">last &raqo;</a>
34                 {% endif %}
35             </span>
36         </div>
37     {% endif %}
38 </div>
39 </body>
40 </html>
41
42 # index.html #
43 {% extends "base.html" %}
44 {% load static %}
45 {% block title %}{{ wybrany_wpis.tytul_wpisu }}{% endblock %}
46 {% block main %}
47     <div class='wpis'>
48         <h1>{{ wybrany_wpis.tytul_wpisu }}</h1>
```

```
49      <div class='trec_wpisu'>{{ wybrany_wpis.trec_wpisu }}</div>
50      <div class='data_wpisu'>{{ wybrany_wpis.created|date:"Y.m.d H:i
51          " }}</div>
52      <div id='tagi_wpisu'>Tagi:
53          <ul>
54              {% for tag in tagi %}
55                  <li><a href="{% url 'wpisy-po-tagu' nazwa_taga=tag.
56                      nazwa_taga %}">{{ tag.nazwa_taga }}</a></li>
57              {% endfor %}
58          </ul>
59      </div>
60  {% endblock %}
61 {# blog/wpisy_po_tagu.html #}
62 {% extends "base.html" %}
63 {% load markup_tags %}
64 {% block title %}{{ wybrany_wpis.tytul_wpisu }}{% endblock %}
65 {% block main %}
66     wpisy oznaczone jako <i>{{ nazwa_taga }}</i>
67     <div class='lista_wpisow'>
68         {% for wpis in wybrane_wpisy %}
69             <div class='wpis'>
70                 <h1><a href="{% url 'wpisy-details' id=wpis.id %}">{{
71                     wpis.tytul_wpisu }}</a></h1>
72                 <div class='trec_wpisu'>
73                     {{ wpis.trec_wpisu|apply_markup:"markdown" }}
74                 </div>
75             {% endfor %}
76         </div>
77     {% endblock %}
```

DRY

Zauważmy, że w metodach `index` i `details` robimy właściwie to samo - pobieramy jakiś wybrany wpis, stronicowaną listę wpisów i tagi. Moglibyśmy te czynności zrobić w jednej funkcji i wywoływać ją w naszych widokach, np.:

```
1 def pobierz_dane(request, model, id=None, ord_by='-updated',
2     ile_na_stronie=15):
3     lista_wpisow = model.objects.all().order_by(ord_by)
4     if id:
5         wybrany_wpis = get_object_or_404(lista_wpisow, pk=id)
6     else:
7         wybrany_wpis = lista_wpisow.latest()
8         paginator = Paginator(lista_wpisow, ile_na_stronie)
9         page = request.GET.get('page')
```

```
9     return wybrany_wpis, paginator.get_page(page), wybrany_wpis.
10    tagi_wpisu.all()
11
12 def index(request):
13     wybrany_wpis, wpisy, tagi = pobierz_dane(request, Wpis)
14     c = {
15         'lista_wpisow': wpisy,
16         'wybrany_wpis': wybrany_wpis,
17         'tagi': tagi,
18     }
19     return render_to_response('blog/index.html', context=c)
20
21 def details(request, id):
22     wybrany_wpis, wpisy, tagi = pobierz_dane(request, Wpis, id)
23     t = loader.get_template('blog/index.html')
24     c = {
25         'lista_wpisow': wpisy,
26         'wybrany_wpis': wybrany_wpis,
27         'tagi': tagi,
28     }
29     wynik = t.render(c)
30     return HttpResponseRedirect(wynik)
```

Choć trzeba przyznać, że w tym przypadku zasada DRY zaczyna lekko kłocić się z zasadą czytelności. Metoda pobierz dane wydaje się być tak na granicy tej czytelności.. ale możemy uznać, że kompromisowo ja tu pozostawimy na razie w takiej formie

korzystanie z ustawień zapisanych w settings.py

Moglibyśmy też chcieć jakoś odgórnie ustawiać ilość wpisów, które powinny pojawiać się na stronie. Np moglibyśmy dodać takie ustawienie w settings.

```
PAGINATION_PER_PAGE = 15
```

By potem skorzystać z tego kodu należałoby po prostu zimportować moduł jazzy.settings. Miałoby to jednak dwie wady: w ten sposób w kodzie aplikacji umieścilibyśmy nazwę projektu (a aplikacja powinna być niezależna od projektu, tak żeby łatwo było używać istniejących aplikacji w nowych projektach), a do tego w zimportowanych ustawieniach nie mielibyśmy ustawień domyślnych, których w settings.py nie zmieniliśmy. Dlatego robi się inaczej: z modułu django.conf importujemy zmienną settings - jest to obiekt, który w swoich atrybutach zawiera wszystkie zmienne z pliku settings.py bieżącego projektu złożone z ustawieniami domyślnymi:

```
1 from django.conf import settings
2 print(settings.PAGINATION_PER_PAGE)
```

Widoki oparte na klasach

Widok to wywoływalny obiekt, który przyjmuje request i zwraca response. To nie musi być tylko funkcja. Django dostarcza przykłady klas, które mogą być użyte jako widoki. Użycie klas zamiast funkcji pozwala na strukturyzację kodu i jego re-używalność poprzez wykorzystywanie mechanizmów dziedziczenia i mixinów. Istnie też kilka widoków generycznych dla prostych, powtarzalnych zadań. Pełna dokumentacja widoków opartych na klasach znajduje się w dokumentacji: <https://docs.djangoproject.com/en/dev/ref/class-based-views/> Wszystkie widoki dziedziczą po klasie View, która obsługuje takie zadania jak połączenie z URLami czy obsługę metod HTTP. RedirectView służy do prostego przekierowywania HTTP, zaś TemplateView rozszerza bazowe klasy w taki sposób, by renderować także szablon.

Proste użycie - w URLconf:

Najprostszym użyciem widoków generycznych jest utworzenie ich bezpośrednio w pliku URLconf

```
1 from django.urls import path
2 from django.views.generic import TemplateView
3 urlpatterns = [
4     path('about/', TemplateView.as_view(template_name="about.html")),
5 ]
```

Ćwiczenie

Używając widoku klasowego TemplateView dodaj widok wyświetlający stronę about.html. Przygotuj odpowiedni szablon.

Tworzenie widoku klasowego

Widok oparty na klasach generalnie umożliwia reagowanie na różne metody żądań HTTP. Robi to za pomocą różnych metod instancji klasy. W przypadku funkcji efekt taki uzyskalibyśmy rozgałęzując kod wewnątrz pojedynczej funkcji widoku:

Podejście funkcyjne:

```
1 from django.http import HttpResponse
2 def my_view(request):
3     if request.method == 'GET':
4         # <view logic>
5         return HttpResponse('result')
```

podejście klasowe:

```
1 from django.http import HttpResponse
2 from django.views import View
3 class MyView(View):
4     def get(self, request):
5         # <view logic>
6         return HttpResponse('result')
```

Ponieważ resolver URL w Django oczekuje, że będzie mógł przestać request i powiązane z nim parametry do wywoływalnej funkcji a nie klasy, widoki bazujące na klasach mają specjalną metodę `as_view()`, która zwraca funkcję, która w razie potrzeby może być przez ten resolver wywołana. Funkcja ta tworzy instancję klasy i wywołuje metodę `dispatch`, która patrzy na `request` i określa jakiego typu jest to pytanie i przekazuje `request` do odpowiedniej metody - o ile taka jest zaimplementowana. Jeśli nie znajdzie takiej metody to rzuca wyjątek: `HttpResponseNotAllowed`

Np:

```
1 # urls.py
2 from django.urls import path
3 from myapp.views import MyView
4 urlpatterns = [
5     path('about/', MyView.as_view()),
6 ]
```

Metody te zwracają jakaś formę `HttpResponse`, działa to tak samo jak w widokach opartych o funkcje.

Widoki oparte na klasach przydatne są szczególnie wtedy, gdy mamy sporo podobnych widoków i chcielibyśmy wykorzystać mechanizmy dziedziczenia itp. W klasach przydatne bywają atrybuty. Co prawda w minimalnym pojęciu nie jest potrzebny żaden atrybut klasowy, by zbudować klasowy widok. Czasem jednak atrybuty mogą być przydatne. Wtedy mamy zasadniczo dwie możliwości ich konfigurowania czy ustawiania. Po pierwsze można je po prostu przesyłać w klasie potomnej. Np: rodzić:

```
1 from django.http import HttpResponse
2 from django.views import View
3 class GreetingView(View):
4     greeting = "Good Day"
5     def get(self, request):
6         return HttpResponse(self.greeting)
```

klasa potomna:

```
1 class MorningGreetingView(GreetingView):
2     greeting = "Morning to ya"
```

Inną opcją jest ustawianie atrybutów klasowych jako kluczowych argumentów metody `as_view()`

wywoływanej w URLconf:

```
1 urlpatterns = [
2     path('about/', GreetingView.as_view(greeting="G'day")),
3 ]
```

Mixiny

Inną korzyścią płynącą z zastosowania widoków klasowych jest możliwość stosowania mixinów. Po polsku nazwalibyśmy to domieszkowaniem. Mixin to taka forma/sposób na wielokrotne dziedziczenie, które pozwala na połączenie zachowań wielu klas rodzicielskich. Mixiny mają być dziedziczone a nie implementowane bezpośrednio. W widokach klasowych powinniśmy podawać Mixiny z lewej strony, klasy bardziej ogólnych widoków powinny być bardziej z prawej strony:

Tworzymy przykładowy mixin:

```
1 class NewMixin(object):
2     def do_whatever(self, *args, **kwargs):
3         #do whatever
```

i domieszkujemy nim nasz widok:

```
1 from django.views.generic import TemplateView
2 class FreshViewClass(NewMixin, TemplateView):
3     def do_something_awesome(self):
4         'continue here'
```

metody z mixinu mogą być wywoływane bezpośrednio w FreshViewClass

Więcej: <https://www.codementor.io/jamesezechukwu/working-with-class-based-views-in-django-5zkjnrwv> <https://docs.djangoproject.com/pl/2.1/topics/class-based-views/intro/>

widoki generyczne

Żeby korzystać z gotowych widoków generycznych, trzeba wiedzieć, jakie klasy mamy do dyspozycji i jakie parametry możemy im ustawać. Te informacje można znaleźć w oficjalnej dokumentacji do Django, w rozdziale: <https://docs.djangoproject.com/en/2.1/topics/class-based-views/generic-display/> Poniżej opiszę kilka wybranych widoków generycznych. Widok django.views.generic.base.View to bardzo prosty widok generyczny - tak prosty, że nie ma sensu go używać, jeśli nie przykryjemy w nim żadnych metod. Więc korzystać z niego jest sens tylko jeśli tworzymy własną klasę dziedziczącą po nim. Widok django.views.generic.base.TemplateView to prosty widok, który wyświetla zadany widok. Kiedy znajdziemy opis tej klasy w dokumentacji Django, nie znajdziemy bezpośrednio przy nim informacji o tym, jakie parametry możemy przekazać

metodzie `as_view`. Ale w przykładzie użycia w dokumentacji jest taka przykładowa klasa dziedzicząca po `TemplateView`:

```
1 from django.views.generic.base import TemplateView
2 from articles.models import Article
3
4 class HomePageView(TemplateView):
5     template_name = "home.html"
6     def get_context_data(self, **kwargs):
7         context = super(HomePageView, self).get_context_data(**kwargs)
8         context['latest_articles'] = Article.objects.all()[:5]
9         return context
```

Jak widać, twórcy klasy określają nazwę szablonu przez przykrycie atrybutu `template_name`. Kiedy korzystamy z widoku generycznego bez tworzenia własnej klasy dziedziczącej po widoku generycznym, przez wołanie `as_view`, każdy atrybut klasy, z której korzystamy, możemy przykryć przekazując tak samo nazywający się parametr metodzie `as_view`. A więc jeśli chcemy, żeby pewien url powodował wyświetlenie szablonu bez przekazania do niego żadnych ciekawych zmiennych, w `urls.py` możemy dopisać:

```
1 url(r'^test3/$', TemplateView.as_view(template_name="blog/wpis.html")),
```

Informację o atrybucie `template_name` możemy też wyszukać w dokumentacji w inny sposób. Większość klas odpowiadających za widoki generyczne dziedziczy po kilku klasach. W dokumentacji możemy wyczytać, że `TemplateView` dziedziczy po klasach `django.views.generic.base.View` i `django.views.generic.base.TemplateResponseMixin` oraz `django.views.generic.base.ContextMixin`. Patrząc na te klasy widzimy, że są tam między innymi takie atrybuty jak `extra_content` i `template_name`, `template_engine`, czy metoda `get_context_data`.

Widok `django.views.generic.list.ListView` służy do wyświetlania listy jakichś obiektów - zwykle pobranych z bazy danych na podstawie modelu. Podobnie jak w przypadku `TemplateView`, z dokumentacji dowiadujemy się, że żeby ustawić, obiekty jakiego modelu chcemy pobrać z bazy danych, musimy ustawić atrybut `model`. Zamiast atrybutu `model` można też ustawić atrybut `queryset` i przekazać w nim obiekt klasy `QuerySet`.

Jeśli nie ustawimy temu widokowi nazwy szablonu, spróbuje on (ale to już nielatwo wyczytać w dokumentacji) użyć szablonu o nazwie `aplikacja/model_list.html` (na przykład szablonu `blog/wpis_list.html`). Do tego szablonu przekazane zostaną dwie zmienne zawierające tę samą listę obiektów: zmienna `object_list` (to można w dokumentacji znaleźć w sekcji `context` w opisie klasy `MultipleObjectMixin`) oraz zmienna o nazwie `<model>_list` (na przykład `wpis_list`) (to można znaleźć w dokumentacji w opisie klasy `MultipleObjectMixin`, przy opisie metody `get_context_object_name`). Jeśli np. w

```
1 # urls.py
```

```
2 ...
3 from django.views.generic import TemplateView, ListView
4 ...
5 urlpatterns = [
6     ...
7     path('wpislist/', ListView.as_view(model=Wpis))
8 ]
```

i utworzymy szablon:

```
1 {% blog/wpis_list.html %}
2 {% extends 'base.html' %}
3 {% block main %}
4     <ul>
5         {% for wpis in wpis_list %}
6             <li>{{ wpis.tytul_wpisu }} </li>
7         {% endfor wpis %}
8     </ul>
9 {% endblock %}
```

To otrzymamy po wejściu na
<http://localhost:8000/wpislist/>

listę wszystkich wpisów.

Gdybyśmy mieli np. model:

```
1 class Osoba(models.Model):
2     imie = models.CharField(max_length=200)
3     nazwisko = models.CharField(max_length=200)
4     wzrost = models.IntegerField()
```

i chcielibyśmy pod jakimś adresem wyświetlić tylko osoby wyższe niż 199, to moglibyśmy do urlpatterns dodać:

```
1 path('wyzsiniz199/', ListView.as_view(queryset=Osoba.objects.filter(
    wzrost__gt=199)))
```

- Widok `django.views.generic.detail.DetailView` służy do wyświetlenia pojedynczego obiektu danego modelu.
- Widok `django.views.generic.dates.YearArchiveView` służy do wyświetlenia wszystkich obiektów danego modelu, które leżą (według wybranego atrybutu zawierającego datę) w danym roku.

Może on też wyświetlić wszystkie miesiące tego roku, w których leży jakiś obiekt tego modelu. Widok ten powinien dostawać w argumencie `year` interesujący nas rok.

Atrybutem (lub, jak zwykle, argumentem przekazywanym metodzie `as_view`) model ustaviamy,

jakiego modelu obiekty chcemy oglądać. Atrybutem date_field ustawiamy, w jakim atrybucie modelu przechowywana jest data, według której mają być wybierane obiekty z bazy danych. Atrybutem make_object_list decydujemy, czy szablonowi mają być przekazane obiekty leżące w danym roku (jeśli nie, zostaną przekazane tylko miesiące danego roku). Domyslnie ten widok wyświetla szablon <model>_archive_year.html (na przykład wpis_archive_year.html). Szablon dostaje między innymi zmienną object_list zawierającą listę obiektów modelu leżących w danym roku. Szablon dostaje też zmienne year, previous_year i next_year, w których są obiekty klasy datetime.date zawierające aktualnie oglądany rok, poprzedni rok w którym są jakieś obiekty i następny rok, w którym są jakieś obiekty. Szablon dostaje też zmienną date_list, w której jest lista obiektów klasy datetime.date z wszystkimi miesiącami aktualnie oglądanego roku, w których jest jakiś obiekt oglądanego modelu. Jeśli przy wyświetlaniu tej listy miesiące pojawiają się niewłaściwe miesiące, oznacza to problemy ze strefą czasową - najprostszym rozwiązaniem jest wtedy ustawienie w settings.py projektu USE_TZ = False. Przykładowo, żeby wyświetlać listę osób urodzonych w danym roku, w urls.py możemy napisać:

```
1 path(
2     'test7/(?P<year>\d\d\d\d)$',
3     YearArchiveView.as_view(model=Osoba, date_field="data_urodzenia",
4                               make_object_list=True)),
```

Widok django.views.generic.dates.MonthArchiveView służy do wyświetlenia wszystkich obiektów danego modelu, które leżą (według wybranego atrybutu zawierającego datę) w danym miesiącu danego roku. Widok ten powinien dostawać w argumencie year interesujący nas rok, a w argumencie month interesujący nas miesiąc. Atrybutem month_format ustawiamy, w jakim formacie będziemy przekazywać (w argumencie month) miesiąc. Jeśli chcemy przekazywać go jako liczbę, w atrybucie month_format powinien znaleźć się napis %m. Atrybuty model i date_field działają jak w klasie YearArchiveView. W przeciwieństwie do klasy YearArchiveView nie trzeba (i nawet nie da się) ustawiać atrybutu make_object_list.

Ćwiczenie: Korzystając z generycznego widoku generycznego MonthArchiveView YearArchiveView dodaj do bloga archiwum podzielone na lata i miesiące. Dodaj do urlpatterns:

```
1 path('archiwum/<int:year>', YearArchiveView.as_view(model=Wpis,
2                                                       date_field="created", make_object_list=True), name='rok'),
2 path('archiwum/<int:year>/<int:month>', MonthArchiveView.as_view(model=
Wpis, date_field="created", month_format="%m"), name='miesiac'),
```

```
]
```

Stwórzmy szablon:

```
1 {# blog/szablon.html #}
```

```
2  {% load markup_tags %} 
3  <html>
4  <head>
5      <meta charset="utf-8">
6      <link rel="stylesheet" href="//netdna.bootstrapcdn.com/twitter-
    bootstrap/2.3.2/css/bootstrap-combined.min.css">
7  </head>
8  <body>
9  <div class="container">
10     <div class="row">
11         <div class="span12">
12             <div class="page-header">
13                 <h1>blog
14                     <small>{% block podtytul %}zrobiony w Django{%
    endblock %}</small>
15                 </h1>
16             </div>
17         </div>
18     </div>
19     <div class="row">
20         <div class="span4">
21             <ul class="nav nav-list well">
22                 {% block menu %}
23                 {% endblock %}
24             </ul>
25         </div>
26         <div class="span8">
27             {% block cialo %}
28             {% endblock %}
29         </div>
30     </div>
31 </div>
32 </body>
33 </html>
```

oraz szablony dziedziczące:

```
1  {# blog/wpis_archive_year.html #}
2  {% extends "blog/szablon.html" %}
3  {% load markup_tags %}
4  {% block podtytul %}wpisy z roku {{ year.year }}{% endblock %}
5  {% block menu %}
6      <li class="nav-header">lata</li>
7      {% if previous_year %}
8          <li><a href="{% url "rok" year=previous_year.year %}">{{ previous_year.year }}</a></li>{% endif %}
9      <li class="active"><a href="{% url "rok" year=year.year %}">{{ year.year }}</a></li>
10     {% if next_year %}
11         <li><a href="{% url "rok" year=next_year.year %}">{{ next_year.year }}</a></li>{% endif %}
```

```
12     <li class="nav-header">rok {{ year.year }}</li>
13     {% for miesiac in date_list %}
14         <li><a href="{% url "miesiac" year=year.year month=miesiac.
15             month %}">{{ miesiac|date:"F" }}</a></li>
16     {% endfor %}
17 {% endblock menu %}
18 {% block cialo %}
19     {% for wpis in object_list %}
20         <h2>{{ wpis.tytul_wpisu }}
21             <small>{{ wpis.created|date:"Y.m.d H:i" }}</small>
22         </h2>
23         <div>
24             {% for tag in wpis.tagi.all %}
25                 <a href="{% url "wpisy_po_tagu" nazwa=tag.nazwa %}">{{ tag.nazwa }}</a>{% endfor %}
26             </div>
27             <div class="tresc">
28                 {{ wpis.tresc|apply_markup:"markdown" }} (<a href="{% url "wpisy-details" id=wpis.pk %}">więcej</a>)
29             </div>
30     {% endfor %}
31 {% endblock cialo %}
```

oraz

```
1  {%# blog/wpis_active_month.html #}
2  {% extends "blog/szablon.html" %}
3  {% load markup_tags %}
4  {% block podtytul %}wpisy z: {{ month|date:"F Y"|lower }}{% endblock %}
5  {% block menu %}
6      <li class="nav-header">świetnie w miesiącu</li>
7      {% if previous_month %}
8          <li>
9              <a href="{% url "miesiac" year=previous_month.year month=
10                  previous_month.month %}">{{ previous_month|date:"F Y"
11                      }}</a>
12      </li>{% endif %}
13      <li class="active"><a href="{% url "miesiac" year=month.year month=
14                  month.month %}">{{ month|date:"F Y" }}</a></li>
15      {% if next_month %}
16          <li><a href="{% url "miesiac" year=next_month.year month=
17                  next_month.month %}">{{ next_month|date:"F Y" }}</a>
18      </li>{% endif %}
19  {% endblock menu %}
20  {% block cialo %}
21      {% for wpis in object_list %}
22          <h2>{{ wpis.tytul_wpisu }}
23              <small>{{ wpis.created|date:"Y.m.d H:i" }}</small>
24          </h2>
25          <div>
26              {% for tag in wpis.tagi.all %}
```

```
23         <a href="{% url "wpisy_po_tagu" nazwa=tag.nazwa %}">{{ tag.nazwa }}</a>{% endfor %}
24     </div>
25     <div class="tresc">
26         {{ wpis.tresc|apply_markup:"markdown" }} (<a href="{% url "wpisy-details" id=wpis.pk %}">więcej</a>)
27     </div>
28 {% endfor %}
29 {% endblock cialo %}
```

Dodawanie obrazków

Nasz blog stałby się atrakcyjniejszy, gdyby do wpisów można było dodawać obrazki. Wśród rodzajów pól, które może mieć model, są rodzaje ImageField i FileField. W polach tych można przechowywać ścieżki dostępu do plików znajdujących się w systemie plików (ze względu na wydajność nie chcemy przechowywać treści plików w bazie danych). Kiedy model ma takie pole, panel administracyjny pozwala uploadować pliki na serwer. Aby dodać do modelu możliwość uploadowania obrazków, należy:

- * W settings.py w MEDIA_ROOT ustawić pełną ścieżkę do katalogu (na serwerze), w którym będziemy chcieli przechowywać pliki.
- * W settings.py w MEDIA_URL ustawić URL prowadzący do katalogu wskazanego w MEDIA_ROOT.
- * Dodać modelowi pole ImageField; jego konstruktorowi należy przekazać argument upload_to zawierający nazwę podkatalogu (w MEDIA_ROOT), do którego będą wrzucane uploadowane obrazki. W zmiennej upload_to możemy używać znaków formatujących taki jak %Y (rok), %m (miesiąc), %d (dzień).
- * Kiedy chcemy dowiedzieć się, gdzie został umieszczony obrazek związany z danym wierszem modelu, możemy sprawdzić atrybut obiekt.pole.url. Atrybut ten jest tworzony na podstawie ustawionej przez nas zmiennej MEDIA_URL. Aby pokazywać dodane obrazki w widokach, możemy dodać do szablonu znacznik . Możemy też pozwolić użytkownikom, żeby sami zajęli się umieszczaniem obrazków we wpisach - markdown pozwala w dowolnym miejscu tekstu umieścić obrazek. Robi się to przy pomocą składni:

```
1 ! [GitHub Logo] (/images/logo.png)
```

Jeśli w treści wpisu wpisu użyję notacji markdown:

```
1 ## To łwesoy wpis o radosnej łmapie
2
3 ### łMapy ąskacz śniedociglet
4
5 Mapy ąrobił
6
7     Mapie figle
8
9 ####
10
11 Ta łmapa łzrobia sobie selfie.
```

```

12
13 ! [GitHub Logo] (/media/images/monkey.jpg)

```

I w szablonie skorzystam z markup_tags

```

1  {# index.html #}
2  {% extends "base.html" %}
3  {% load static markup_tags%}
4  {% block title %}{{ wybrany_wpis.tytul_wpisu }}{% endblock %}
5  {% block main %}
6      <div class='wpis'>
7          <h1>{{ wybrany_wpis.tytul_wpisu }}</h1>
8          <div class='tresc_wpisu'>{{ wybrany_wpis.tresc_wpisu|apply_markup:"markdown" }}</div>
9          <div class='data_wpisu'>{{ wybrany_wpis.created|date:"Y.m.d H:i" }}</div>
10         <div id='tagi_wpisu'>Tagi:
11             <ul>
12                 {% for tag in tagi %}
13                     <li><a href="{% url 'wpisy-po-tagu' nazwa_taga=tag.nazwa_taga %}">{{ tag.nazwa_taga }}</a>
14                 {% endfor %}
15             </ul>
16         </div>
17     </div>
18 {% endblock %}

```

w settings ustawię:

```

1 MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
2 MEDIA_URL = "/media/"

```

a w głównym urlpattern dodam:

```

1 from django.conf import settings
2 from django.conf.urls.static import static
3 ...
4 urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)

```

i do katalogu

```

1 ś
2 <żcieka do jazzy>/media/images/

```

wrzucę obrazek malpa.jpg

To powiniensem dostać w wyniku wyświetlony obrazek w sformatowanej treści:



Figure 0.11: image-20210608012114375

Źródło obrazka: <https://fotoblogia.pl/10948,fotograf-splajtowal-przez-malpe-teraz-mysli-o-wyprowadzaniu-psow-by-zarobic-na-zycie>

udostępnianie statycznych plików

Oprócz zasobów generowanych dynamicznie aplikacje django potrzebują udostępniać dwa rodzaje statycznych plików. Czasem potrzebujemy udostępniać pliki stworzone przez twórcę aplikacji, takie jak pliki css, programy javaskryptowe i obrazki używane w szablonach - te pliki w django nazywamy static files. Potrzebujemy też czasem udostępniać pliki wrzucane na serwer przez użytkowników, na przykład zdjęcia dodawane do wpisów na blogu czy do opisów produktów w sklepie internetowym - te pliki nazywamy w django media files. Django nie służy do udostępniania statycznych plików. Na produkcji Django zawsze działa w połączeniu z jakimś zwykłym

serwerem WWW (na przykład Apaczem) i na produkcji statyczne pliki serwuje się głównym serwera WWW (na przykład Apaczem), bez udziału Django. Ale przy deweloperce, kiedy korzysta się z wbudowanego w Django serwera WWW, wygodnie jest, żeby sam Django udostępniał statyczne pliki. Po to jest w django.contrib aplikacja staticfiles - to jest aplikacja, która umie udostępniać statyczne pliki. Jeśli chcemy, żeby częścią naszej aplikacji były statyczne pliki, umieszczamy je w katalogu aplikacji (a nie w katalogu projektu) w podkatalogu static/nazwa_aplikacji. Upewniamy się, że w settings.py projektu zmienna DEBUG jest ustawiona na true, na liście włączonych aplikacji jest aplikacja django.contrib.staticfiles, a w zmiennej STATIC_URL jest napis /static/. Kiedy zaserwujemy nasz projekt serwerem WWW wbudowanym w Django, pod adresem http://ip:port/static/nazwa_aplikacji/nazwa_pliku udostępniony będzie statyczny plik. Jako prosty konkretny przykład stworzymy projekt z dwiema aplikacjami, w których będą statyczne pliki. Zaczynamy od stworzenia projektu, a w nim dwu aplikacji: udostępnianie statycznych plików Oprócz zasobów generowanych dynamicznie aplikacje djangowe potrzebują udostępniać dwa rodzaje statycznych plików. Czasem potrzebujemy udostępniać pliki stworzone przez twórcę aplikacji, takie jak pliki css, programy javaskryptowe i obrazki używane w szablonach - te pliki w django nazywamy static files. Potrzebujemy też czasem udostępniać pliki wrzucane na serwer przez użytkowników, na przykład zdjęcia dodawane do wpisów na blogu czy do opisów produktów w sklepie internetowym - te pliki nazywamy w django media files. Django nie służy do udostępniania statycznych plików. Na produkcji Django zawsze działa w połączeniu z jakimś zwykłym serwerem WWW (na przykład Apaczem) i na produkcji statyczne pliki serwuje się głównym serwerem WWW (na przykład Apaczem), bez udziału Django. Ale przy deweloperce, kiedy korzysta się z wbudowanego w Django serwera WWW, wygodnie jest, żeby sam Django udostępniał statyczne pliki. Po to jest w django.contrib aplikacja staticfiles - to jest aplikacja, która umie udostępniać statyczne pliki. Jeśli chcemy, żeby częścią naszej aplikacji były statyczne pliki, umieszczamy je w katalogu aplikacji (a nie w katalogu projektu) w podkatalogu static/nazwa_aplikacji. Upewniamy się, że w settings.py projektu zmienna DEBUG jest ustawiona na true, na liście włączonych aplikacji jest aplikacja django.contrib.staticfiles, a w zmiennej STATIC_URL jest napis /static/. Kiedy zaserwujemy nasz projekt serwerem WWW wbudowanym w Django, pod adresem http://ip:port/static/nazwa_aplikacji/nazwa_pliku udostępniony będzie statyczny plik. Jako prosty konkretny przykład stworzymy projekt z dwiema aplikacjami, w których będą statyczne pliki. Zaczynamy od stworzenia projektu, a w nim dwu aplikacji:

udostępnianie statycznych plików

Oprócz zasobów generowanych dynamicznie aplikacje djangowe potrzebują udostępniać dwa rodzaje statycznych plików. Czasem potrzebujemy udostępniać pliki stworzone przez twórcę aplikacji, takie jak pliki css, programy javaskryptowe i obrazki używane w szablonach - te pliki w django nazywamy static files. Potrzebujemy też czasem udostępniać pliki wrzucane na serwer przez użytkowników, na przykład zdjęcia dodawane do wpisów na blogu czy do opisów produktów

w sklepie internetowym - te pliki nazywamy w django media files. Django nie służy do udostępniania statycznych plików. Na produkcji Django zawsze działa w połączeniu z jakimś zwykłym serwerem WWW (na przykład Apaczem) i na produkcji statyczne pliki serwuje się głównym serwerem WWW (na przykład Apaczem), bez udziału Django. Ale przy deweloperce, kiedy korzysta się z wbudowanego w Django serwera WWW, wygodnie jest, żeby sam Django udostępniał statyczne pliki. Po to jest w django.contrib aplikacja staticfiles - to jest aplikacja, która umie udostępniać statyczne pliki. Jeśli chcemy, żeby częścią naszej aplikacji były statyczne pliki, umieszczamy je w katalogu aplikacji (a nie w katalogu projektu) w podkatalogu static/nazwa_aplikacji. Upewniamy się, że w settings.py projektu zmienna DEBUG jest ustawiona na true, na liście włączonych aplikacji jest aplikacja django.contrib.staticfiles, a w zmiennej STATIC_URL jest napis /static/. Kiedy zaserwujemy nasz projekt serwerem WWW wbudowanym w Django, pod adresem http://ip:port/static/nazwa_aplikacji/nazwa_pliku udostępniony będzie statyczny plik. Jako prosty konkretny przykład stworzymy projekt z dwiema aplikacjami, w których będą statyczne pliki. Zaczynamy od stworzenia projektu, a w nim dwu aplikacji:

```
1 $ mkvirtualenv statycznepliki
2 $ pip install django
3 $ django-admin startproject statycznepliki
4 $ cd statycznepliki
5 $ python manage.py startapp bolek
6 $ python manage.py startapp lolek
7 $ mkdir -p bolek/static/bolek
8 $ echo 'kuku<b>ryku</b>!' > bolek/static/bolek/test.html
9 $ mkdir -p lolek/static/lolek
10 $ echo 'bum <i>cyk</i>' > lolek/static/lolek/test.html
```

Zagładamy do settings.py i dopisujemy do listy aplikacji aplikacje bolek i lolek, oraz upewniamy się, że są tam te wpisy:

```
1 DEBUG = True
2 (...)
3 INSTALLED_APPS = (
4     'django.contrib.admin',
5     ...
6     'django.contrib.staticfiles',
7     'bolek',
8     'lolek',
9 )
10 (...)
11 STATIC_URL = '/static/'
```

Uruchamiamy serwer WWW wbudowany w Django:

```
1 $ python manage.py runserver
```

Wchodzimy na odpowiedni adres:

<http://localhost:8000/static/bolek/test.html>

I powinniśmy zobaczyć, że plik jest serwowany:

```
1 kukuryku!
```

Aplikacja staticfiles będzie serwować w ten sposób pliki tylko jeśli będą spełnione dwa warunki: w settings.py w zmiennej DEBUG będzie True (co oznacza, że nie jesteśmy na produkcji), a projekt jest serwowany przez serwer wbudowany w Django. Możemy łatwo sprawdzić, że kiedy aplikację zasugerujemy przy użyciu np. Gunicorn, pliki statyczne nie będą serwowane: \$gunicorn statycznepliki.wsgi. Możemy też łatwo sprawdzić, że pliki statyczne nie będą serwowane nawet przez serwer wbudowany w Django, jeśli zmienną DEBUG ustawimy na False (tylko wtedy musimy jeszcze ustawić zmienną ALLOWED_HOSTS):

```
1 DEBUG = False
2 (... )
3 ALLOWED_HOSTS = ['127.0.0.1']
```

Gdybyśmy bardzo chcieli, moglibyśmy też zmusić aplikację staticfiles, żeby serwowała pliki statyczne również na produkcji - ale nie jest to dobry pomysł. Lepiej przekopiować katalogi static wszystkich aplikacji używanych w projekcie w jeden katalog i skonfigurować serwer WWW tak, żeby serwował ten katalog bezpośrednio, bez przekazywania żądania do Django. W tym celu w settings.py wpisujemy, w którym katalogu chcemy trzymać wszystkie pliki statyczne naszego projektu:

```
1 STATIC_ROOT = '/home/django/statycznepliki/static/'
```

Następnie wydajemy polecenie:

```
1 $ ./manage.py collectstatic
2
3 121 static files copied to '/home/rkorzen/workspace/python-django-alx/
projekty/statycznepliki/static'.
```

To polecenie przekopiuje zawartości katalogów static wszystkich aplikacji do katalogu wskazanego przez zmienną STATIC_ROOT. Następnie musimy skonfigurować używany przez nas serwer WWW tak, żeby serwował ten katalog bezpośrednio. Jak to zrobić sprawdź w rozdziale dotyczącym wdrożeń. Często potrzebujemy w szablonach odwoływać się do plików statycznych, na przykład do plików CSS lub programów javaskryptowych. W naszym przykładzie wśród plików statycznych umieścmy plik test.js:

```
1 $ echo "alert('prot i filip lat juz wiele zyja jako przyjaciele'))" >
bolek/static/bolek/test.js
2 $ python manage.py collectstatic
```

Teraz zrobimy szablon ładujący ten skrypt. Moglibyśmy zaszyć w nim URL, spod którego serwujemy

pliki statyczne, ale nie jest to dobry plik - szablon jest częścią aplikacji, a URL katalogu ze statycznymi plikami konfigurujemy w projekcie. Dlatego w szablonie zrobimy tak:

```
1 $ mkdir -p bolek/templates/bolek
2 $ cat > bolek/templates/bolek/test.html <<KONIEC
3 ${% load staticfiles %}
4 <script src="{% static "bolek/test.js" %}"></script>
```

W statycznepliki/urls.py dopisujemy:

```
1 from django.views.generic import TemplateView
2 urlpatterns = [
3     path('admin/', admin.site.urls),
4     path('', TemplateView.as_view(template_name='bolek/test.html')),
5 ]
```

Następnie wchodzimy pod właściwy adres i widzimy naszą stronę:

A teraz zobaczymy - nadal na naszym prostym przykładzie - jak udostępniać pliki uploadowane przez użytkowników (media files). Najpierw instalujemy pakiet PIL:

```
1 $ pip install pillow
2 W aplikacji bolek tworzymy taki model:
3 $ cat >> bolek/models.py <<KONIEC
4 class Produkt(models.Model):
5     nazwa = models.CharField(max_length=200)
6     fotka = models.ImageField(upload_to='ilustracje_do_produktow/%Y/%m
    /%d')
```

Przygotowujemy i wykonujemy migracje:

```
1 $ ./manage.py makemigrations
2 Migrations for 'bolek':
3   bolek/migrations/0001_initial.py
4     - Create model Produkt
5 (statycznepliki) rkorzen@rkorzen-lat:~/workspace/python-django-alx/
    projekty/statycznepliki$ ./manage.py migrate
6 Operations to perform:
7   Apply all migrations: admin, auth, bolek, contenttypes, sessions
8 Running migrations:
9   Applying contenttypes.0001_initial... OK
10  Applying auth.0001_initial... OK
11  Applying admin.0001_initial... OK
12  Applying admin.0002_logentry_remove_auto_add... OK
13  Applying admin.0003_logentry_add_action_flag_choices... OK
14  Applying contenttypes.0002_remove_content_type_name... OK
15  Applying auth.0002_alter_permission_name_max_length... OK
16  Applying auth.0003_alter_user_email_max_length... OK
17  Applying auth.0004_alter_user_username_opts... OK
18  Applying auth.0005_alter_user_last_login_null... OK
19  Applying auth.0006_require_contenttypes_0002... OK
```

```
20 Applying auth.0007_alter_validators_add_error_messages... OK
21 Applying auth.0008_alter_user_username_max_length... OK
22 Applying auth.0009_alter_user_last_name_max_length... OK
23 Applying bolek.0001_initial... OK
24 Applying sessions.0001_initial... OK
25
26 Dodajemy model do panelu admina:
27 $ cat >> bolek/admin.py << KONIEC
28 from bolek.models import Produkt
29 admin.site.register(Produkt)
```

Do settings.py dopisujemy potrzebne ustawienia:

```
1 MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
2 MEDIA_URL = '/media/'
```

Uruchamiamy serwer:

```
1 $ python manage.py runserver
```

Wchodzimy przeglądarką pod odpowiedni adres, w panelu administracyjnym tworzymy kilka produktów i wrzucamy do nich zdjęcia: W katalogu media powinny pojawić się wgrane media.

Jeśli teraz pobierzemy z bazy danych obiekt klasy Produkt, możemy sprawdzić, jaki jest jego URL:

```
1 from bolek.models import Produkt
2 Produkt.objects.first().fotka.url
3 '/media/ilustracje_do_produktow/2018/12/16/mapapl-sam.jpg'
```

Żeby media files były serwowane przez serwer wbudowany w Django, w urls.py projektu dopisujemy:

```
1 # statycznepliki/urls.py
2 from django.conf import settings
3 from django.conf.urls.static import static
4 from django.contrib import admin
5 from django.urls import path
6 from django.views.generic import TemplateView
7 urlpatterns = [
8     path('admin/', admin.site.urls),
9     path('', TemplateView.as_view(template_name='bolek/test.html')),
10 ]
11 urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Ten sposób działa tylko wtedy, kiedy w settings.py w zmiennej DEBUG będzie True

Ćwiczenia:

1. Dodaj do bloga możliwość dodawania obrazków. Niech z każdym wpisem będzie związany jeden nieobowiązkowy obrazek, niech szablony pokazujące treść wpisu pokazują też treść jego obrazka.
2. Dodaj do bloga możliwość dodawania obrazków - ale w inny sposób. Niech użytkownik ma możliwość dodawania dowolnej ilości obrazków, bez wiążania ich z wpisami. Powiadamiamy użytkownika, pod jakim URL-em dostępny jest załadowany przez niego obrazek. Kiedy użytkownik chce umieścić obrazek przy wpisie, niech używa markdown.
3. Zrób stronę zawierającą chmurę tagów.

Ad. 1.

Dodaję w modelu nowe pole:

```
obrazek = models.ImageField(upload_to='obrazki_do_wpisow/%Y/%m/%d', blank=True)
```

i oczywiście robię migrację:

```
1 jazzy$./manage.py makemigrations
2 Migrations for 'blog':
3   blog/migrations/0005_auto_20181216_1334.py
4     - Create model Osoba
5     - Add field obrazek to wpis
6 jazzy$./manage.py migrate
7 Operations to perform:
8   Apply all migrations: admin, auth, blog, contenttypes, sessions
9 Running migrations:
10 Applying blog.0005_auto_20181216_1334... OK
11 jazzy$
```

Urle powinny być już przygotowane w jednym z poprzednich kroków. Gdyby tak nie było:

```
1 from django.conf import settings
2 from django.conf.urls.static import static
3 urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Dodajmy jeszcze zmianę w szablonach dopisując tam:

```
1 <div class='obrazek'>
2   {% if wybrany_wpis.obrazek %}
6             {{ tag.nazwa_taga }}
7         </a>
8     {% endfor %}
9 {% endblock main %}
```

Komentarze

Teraz dodamy do bloga możliwość komentowania wpisów. Zmiany w modelach i szablonach będą proste - nie będą w niczym różne od tego, co już robiliśmy. Nową rzeczą będzie natomiast wyświetlanie i walidowanie formularzy oraz przyjmowanie danych z formularzy. Najpierw zrobimy to w prosty sposób. Do szablonu wyświetlającego wpis dodamy zwykły formularz html. Będzie on wysyłał metodą post swoje dane pod adres /jazzy/blog/komentuj/. Adres ten będzie prowadził do zwykłego widoku. Widok ten znajdzie dane przesłane przez formularz w atrybutie POST atrybutu request - będzie mógł więc stworzyć nowy obiekt klasy Komentarz i zapisać go w bazie. Na koniec powinien on przekierować przeglądarkę pod stronę wyświetlającą komentowany wpis. W tym celu możemy stworzyć obiekt klasy django.http.HttpResponseRedirect (jako argument przekazując jego konstruktorowi nowy URL) i zwrócić go z widoku.

Zabezpieczenie przed CSRF

<https://docs.djangoproject.com/en/2.1/ref/csrf/>

Django ma wbudowane zabezpieczenie przed CSRF. Działa ono w następujący sposób: Kiedy wysyłamy do przeglądarki formularz, mamy dwa obowiązki: musimy umieścić w odpowiedzi HTTP ciasteczkę o nazwie csrfmiddlewaretoken zawierającą w sobie dowolny losowy napis. Ten sam napis musimy umieścić w ukrytym polu formularza nazwanym csrfmiddlewaretoken. A kiedy Django dostaje z przeglądarki żądanie przysłane metodą POST, sprawdza, czy zawiera ono ciasteczkę csrfmiddlewaretoken i parametr POST csrfmiddlewaretoken i czy są one równe. Mechanizmy, które pozwalają łatwo umieszczać w formularzu ten parametr i dodawać do odpowiedzi HTTP to ciasteczko, i które sprawdzają przychodzące żądania POST, czy mają w sobie właściwy parametr i ciasteczko, wykorzystują dwa ogólniejsze mechanizmy - mechanizm middleware i mechanizm procesorów kontekstu.

Middleware

<https://docs.djangoproject.com/en/2.1/topics/http/middleware/>

Każda aplikacja jest podzielona na widoki. Ale są problemy, które są prostopadłe do tego podziału - problemy, które wymagają, żeby coś było zrobione niezależnie od tego, który widok jest wyświetlany. Takim problemem jest właśnie sprawdzanie ciasteczka csrfmiddlewaretoken i parametru POST csrfmiddlewaretoken. Takim problemem jest też dostarczanie każdemu szablonowi tokena csrf, żeby mógł on być wyświetlony w odpowiednim miejscu szablonu. Do wykonywania działań przy obsłudze każdego żądania służą middleware'y. Kiedy chcemy, żeby przy obsłudze każdego żądania było robione coś, tworzymy klasę (nazywaną właśnie middleware), umieszczamy w niej metody o specjalnych nazwach, po czym w settings.py projektu dopisujemy je do listy middleware'ów. Teraz przy obsłudze każdego żądania staną się między innymi trzy rzeczy. Przed uruchomieniem widoku uruchamiane są po kolei wszystkie middleware'y (w kolejności, w jakiej zostały wymienione na liście w settings.py), każdy z nich dostaje obiekt z żądaniem i może go zmienić. Middleware może też zwrócić na tym etapie własną odpowiedź - wtedy przekazywanie żądania kolejnym middleware zostanie przerwane, a przeglądarkę zostanie wysłana ta odpowiedź (tylko przedtem ta odpowiedź zostanie przepuszczona przez wszystkie middleware'y, ale o tym za chwilę). Dzięki temu, że na etapie przetwarzania dowolny middleware może zwrócić odpowiedź, dowolny middleware może spodowować odrzucenie żądania bez przekazywania go widokowi (na przykład jeśli zostanie wykryty atak CSRF). Następnie, kiedy żądanie przejdzie przez wszystkie middleware'y, na chwilę przed uruchomieniem widoku znów zostają uruchomione wszystkie middleware'y i każdy dostaje obiekt z żądaniem (request), widok który ma zostać uruchomiony, oraz argumenty, które mają zostać mu przekazane. Na tym etapie middleware może znów zmodyfikować żądanie. Middleware może też na tym etapie zwrócić obiekt HttpRequest, co spowoduje, że widok nie zostanie uruchomiony, a odpowiedź zwrocona przez middleware zostanie wysłana przeglądarkę (tylko przedtem ta odpowiedź zostanie przepuszczona przez wszystkie middleware'y, ale o tym za chwilę).

A potem, kiedy widok zostanie już uruchomiony, i jest już gotowa odpowiedź (obiekt `HttpRequest`), którą trzeba wysłać przeglądarce, wszystkie middleware'y zostają uruchomione jeszcze raz, tym razem w kolejności odwrotnej do tej, w jakiej są na liście w `settings.py`. Każdy z nich dostaje obiekt z żądaniem (`HttpRequest`) i obiekt z odpowiedzią (`HttpResponse`) i może zmodyfikować odpowiedź. Przykładowo, jeśli chcemy, żeby każde żądanie było odrzucane z komunikatem tu nic nie ma, możemy zrobić (zwyczajowo w pliku aplikacja/middleware.py) taki middleware:

```
1 from django.http import HttpResponse
2 class MojMiddleware(object):
3     def __init__(self, get_response):
4         self.get_response = get_response
5     def __call__(self, request):
6         return self.get_response(request)
7     def process_view(self, request, *args, **kwargs):
8         return HttpResponse("tu nic nie ma")
```

I zarejestrować go w `settings.py` pisząc:

```
1 MIDDLEWARE = [
2     'blog.middleware.MojMiddleware',
3     ...
```

Procesory kontekstu

Chcemy czasem, żeby middleware mógł przekazać jakieś dane szablonowi. Mechanizm, który służy do tego, jest następujący. Middleware może umieścić dowolne rzeczy w obiekcie z żądaniem (w obiekcie klasy `HttpRequest`), zwykle w jego atrybutu `META` (który to atrybut zawiera słownik ze wszystkimi nagłówkami HTTP). Następnie twórca widoku musi nie tworzyć kontekstu jako obiektu `Context`, ale jako obiekt dziedziczącej po `Context` klasy `RequestContext`. Konstruktor tej klasy w pierwszym argumentie potrzebuje dostać obiekt z żądaniem, a następnie uruchamia po kolei procesory kontekstu wymienione w liście `TEMPLATES`, w sekcji `context_processors` zdefiniowanej w `settings.py` projektu. Każdy taki procesor kontekstu jest callablem (na przykład funkcją). Konstruktor klasy `RequestContext` uruchamia te callable, przekazuje im obiekt z żądaniem, a one mają zwracać słowniki - które konstruktor `RequestContext` doda do kontekstu. Zauważmy, że żeby korzystać z tego mechanizmu, musimy w naszym widoku zamiast korzystać ze zwykłej klasy `Context`, korzystać z klasy `RequestContext`. Jeśli w naszym widoku nigdzie nie tworzymy obiektu klasy `Context`, to pewnie dla tego, że korzystamy z widoków generycznych, funkcji `render_to_response` lub podobnego mechanizmu. W takim przypadku nie musimy robić niczego szczególnego, bo te mechanizmy korzystają z klasy `RequestContext`. Dla przykładu wyobraźmy sobie, że zechcemy każdemu szablonowi dostarczyć w kontekście zmienną `kot` zawierającą napis `bella`. W tym celu w naszej aplikacji (zwyczajowo w pliku `context_processors.py`) piszemy taki procesor kontekstu:

```
1 def moj_context_processor(request):
2     return {"kot": "bella"}
```

Następnie musimy zarejestrować nasz procesor kontekstu w settings.py, dopisując go do listy context_processors w TEMPLATES.

```
1 TEMPLATES = [
2     {
3         'BACKEND': 'django.template.backends.django.DjangoTemplates',
4         'DIRS': [os.path.join(BASE_DIR, 'templates')],
5         # 'APP_DIRS': True,
6         'OPTIONS': {
7             'loaders': ...],
8             'context_processors': [
9                 ...
10                'blog.context_processors.moj_context_processor'
11            ],
12        },
13    },
14 ]
```

Teraz żeby sprawdzić, czy nasz procesor kontekstu rzeczywiście działa, tworzymy jakiś szablon - np. blog/test_context_procesora.html: mój kot nazywa się {{kot}} i podłączamy go pod jakiś URL pisząc w pliku urls.py:

```
1 # blog/urls.py
2 path('testcp/', TemplateView.as_view(template_name='blog/
test_context_procesora.html'))
```

Po wejściu na ten adres powinniśmy zobaczyć szablon z zawartością pochodząą z procesora kontekstu

Formularze

Zwykły formularz - prosty przykład

Zobaczmy na prostym przykładzie, jak zrobić prosty formularz pozwalający dodawać rzeczy do bazy danych. Zaczynamy od stworzenia projektu, a w nim aplikacji:

```
1 $ mkvirtualenv formularze
2 $ django-admin startproject formularze
3 $ cd formularze/
4 $ ./manage.py startapp auto
```

Dodajemy apkę do aplikacji w settings, tworzymy model

```
1 from django.db import models
2 # Create your models here.
3 class Auto(models.Model):
4     marka = models.CharField(max_length=50)
5     numer = models.CharField(max_length=10)
6 Tworzymy i wykonujemy migracje.
7 Wykorzystujemy generyczny widok w URLs
8 urlpatterns = [
9     ...
10    path('', TemplateView.as_view(template_name="auta/formularz.html"))
11 ]
```

Uruchamiamy aplikację:

```
1 $ python manage.py runserver
```

Po wejściu na <http://localhost:8000> powinniśmy zobaczyć formularz. Po wypełnieniu go dostaniemy informację o tym, że nie mamy odpowiedniej akcji Poprawiamy to:

```
1 # urls.py
2 from django.views.generic import TemplateView
3 from auta.views import dodaj
4 urlpatterns = [
5     path('admin/', admin.site.urls),
6     path('', TemplateView.as_view(template_name="auta/formularz.html"))
7     ,
8     path('dodaj', dodaj)
9 ]
10 # auta/views.py
11 from django.http import HttpResponseRedirect
12 from .models import Auto
13 # Create your views here.
14 def dodaj(request):
15     marka = request.POST['marka']
16     numer = request.POST['numer']
17     Auto.objects.create(marka=marka, numer=numer)
18     return HttpResponseRedirect('dodane')
```

Tym razem przy próbie wysłania dostaniemy następujący błąd:

poprawiamy go modyfikując szablon

```
1 <form method="POST" action="dodaj">
2     {% csrf_token %}
3     <input type="text" name="marka"><br>
4     <input type="text" name="numer"><br>
5     <button type="submit">dodaj</button>
6 </form>
```

Teraz formularz powinien się poprawnie wysłać. Możemy to sprawdzić w konsoli bądź utworzyć użytkownika, zarejestrować model w adminie i przeglądać go z poziomu PA!

Ćwiczenie.

1. Dodaj blogowi możliwość komentowania wpisów.

```
1 # blog/urls.py
2 from django.urls import path
3 from django.views.generic import TemplateView
4 from . import views
5 urlpatterns = [
6     ...
7     path('komentuj/<int:id_wpisu>', views.komentuj,
8          name='komentuj-
9          wpis'),
10 ]
```

```
1 # blog/models.py
2 ...
3 class Komentarz(models.Model):
4     nick = models.CharField(max_length=200)
5     email = models.CharField(max_length=200, blank=True, null=True)
6     tytul = models.CharField(max_length=200)
7     tresc = models.TextField()
8     data = models.DateTimeField(auto_now_add=True)
9     wpis = models.ForeignKey(Wpis, on_delete=models.CASCADE)
10    def __str__(self):
11        return self.tytul
12    class Meta:
13        verbose_name_plural = "komentarze"
14        ordering = ['data']
15
16 # blog/admin.py
17 ...
18 from blog.models import Wpis, Tag, Komentarz
19
20 @admin.register(Komentarz)
21 class AdminKomentarz(admin.ModelAdmin):
22     pass
```

```
1 {# index.html #}
2 ...
3 <div class='formularz_komentarze'>
4     <form action='{% url 'wpisy-details' id=wybrany_wpis.pk %}' method='
5         post'>
6         {% csrf_token %}
7         nick: <input type='text' name='nick_autora_komentarza' /><br/>
8         email: <input type='text' name='email_autora_komentarza' /><br/>
9         tytul: <input type='text' name='tytul_komentarza' /><br/>
```

```
9   komentarz: <br/>
10  <textarea name='tresc_komentarza' cols='40' rows='12'></textarea>
11  <input type='submit' value='ok' />
12  </form>
13 </div>
14
15  {% if wybrany_wpis.komentarz_set.all %}
16  <div class='lista_komentarzy'>
17  {% for komentarz in wybrany_wpis.komentarz_set.all %}
18    <div class='komentarz'>
19      <div class='nick_autora_komentarza'>{{ komentarz.nick }}</div>
20      <div class='email_autora_komentarza'>{{ komentarz.email }}</div>
21      <div class='tytul_komentarza'>{{ komentarz.tytul }}</div>
22      <div class='data_komentarza'>
23        {{ komentarz.data|date:"Y.m.d H:i" }}
24      </div>
25      <div class='tresc_komentarza'>
26        {{ komentarz.tresc|apply_markup:"markdown" }}
27      </div>
28    </div>
29  {% endfor %}
30 </div>
31 {% endif %}
```

Zmienić też będącemu musieli nieco funkcje widoków. Bezpieczniej będzie korzystać w nich z funkcji render (Ma to związek z tym problemem: <https://stackoverflow.com/questions/13048228/django-csrf-token-was-used-in-a-template-but-the-context-did-not-provide>)

```
1 def index(request):
2     wybrany_wpis, wpisy, tagi = pobierz_dane(request, Wpis)
3     c = {
4         'lista_wpisow': wpisy,
5         'wybrany_wpis': wybrany_wpis,
6         'tagi': tagi,
7     }
8     return render(request, 'blog/index.html', context=c)
9
10
11 def details(request, id):
12     """Przykład widoku korzystajcego ze zwykłego formularza html"""
13     if request.method == 'POST':
14         nick = request.POST['nick']
15         email = request.POST['email']
16         tytul = request.POST['tytul']
17         tresc = request.POST['tresc']
18         k = Komentarz()
19         k.nick = nick
20         k.email = email
21         k.tytul = tytul
22         k.tresc = tresc
```

```
23         k.wpis_id = id
24         k.save()
25     else:
26         wybrany_wpis, wpisy, tagi = pobierz_dane(request, Wpis, id)
27         c = {
28             'lista_wpisow': wpisy,
29             'wybrany_wpis': wybrany_wpis,
30             'tagi': tagi,
31         }
32     return render(request, 'blog/index1.html', c)
```

jak korzystać z zabezpieczenia przed CSRF

Żeby skorzystać z zabezpieczenia przed CSRF, musimy najpierw spowodować, żeby w formularzu pojawiło się ukryte pole wyświetlające token umieszczony w kontekście przez procesor kontekstu, który z kolei dostaje ten token od middleware'u. Musimy więc sprawdzić czy settings.py włączony jest odpowiedni middleware. Domyslnie powinien znajdować się na liśc:

```
1 MIDDLEWARE = [
2     ...
3     'django.middleware.csrf.CsrfViewMiddleware',
4     ...
5 ]
```

Po włączeniu middleware'u CsrfViewMiddleware w żądaniu powinien pojawić się token. Możemy to sprawdzić w dowolnym widoku pisząc:

```
1 # blog/views.py
2 def test_csrf_cookie(request):
3     return HttpResponse("csrf cookie: " + request.META["CSRF_COOKIE"])
```

i dodać odpowiednią ścieżkę do urls

```
1 path('testcsrfcookie/', views.test_csrf_cookie),
```

Teraz musimy włączyć procesor kontekstu, który przepisze ten token z żądania do kontekstu - jest to procesor kontekstu django.core.context_processors.csrf. W tym celu nie musimy nic robić - ten procesor kontekstu jest włączony zawsze, nawet jeśli nie jest wymieniony na liście context_processors. Ten procesor kontekstu wpisuje token do zmiennej csrf_token. Że tak jest, możemy się przekonać pisząc w dowolnym szablonie: token: {{ csrf_token }} Zmienna csrf_token zawiera tylko wartość parametru, który ma być wstawiony w formularz. Żeby było wygodnie, w djangowym języku szablonów istnieje tag {{ csrf_token }}, który wstawia w formularzu to ukryte pole, w całości, z całą potrzebną otoczką. Używa się go tak, że w szablonie piszemy:

```
1 <form action='...' method='post'>
```

```
2      {% csrf_token %}  
3      (...) reszta formularza  
4  </form>
```

To wszystko, co zrobiliśmy do tej pory, spowodowało, że do formularza zostaje dodane ukryte pole. Trzeba jeszcze, żeby razem z formularzem było wysyłane ciasteczko. W tym celu nie trzeba robić nic ponad to, co już zrobiliśmy - za każdym razem, kiedy wstawiamy w szablon token csrf, do odpowiedzi HTTP wysyłanej do przeglądarki dołączane jest ciasteczko, o czym można się przekonać wykonując w przeglądarce (na przykład w konsoli Firebuga)/javaskryptowe polecenie alert(document.cookie).

Teraz trzeba jeszcze sprawdzić w żądaniach POST, czy zawierają one poprawne ciasteczko i parametr. W tym celu nie musimy robić nic ponad to, co już zrobiliśmy - middleware django.middleware.csrf.CsrfViewMiddleware, który już włączyliśmy, sprawdza w przychodzących żądaniach POST, czy zawierają one poprawne ciasteczko i parametr.

django.forms

Żeby oszczędzić sobie czasu i pisania, warto skorzystać z biblioteki django.forms. Biblioteka ta ułatwia wyświetlanie i walidowanie formularzy. Formularze definiujemy podobnie jak modele - dziedzicząc po klasie Form i definiując atrybuty opisujące poszczególne pola formularza. Każdy taki atrybut jest obiektem klasy, która określa jego rodzaj. Na przykład aby zdefiniować formularz z polami do przechowywania napisu, adresu emailowego i liczby tworzymy taką klasę:

```
1 from django import forms  
2 class MojFormularz(forms.Form):  
3     imie = forms.CharField()  
4     email = forms.EmailField()  
5     wiek = forms.IntegerField()
```

Przy rysowaniu formularza każde pole zostanie przedstawione jako pewien widget. Domyslnie każde pole CharField jest rysowane jako widget TextInput (czyli pole <input type='text'>), pole BooleanField jako widget CheckboxInput (pole <input type="checkbox">) itp. Można to zmienić - konstruktorowi każdego pola można podać argument widget:

```
1 wiek = forms.IntegerField(widget=forms.HiddenInput)
```

Niektóre przydatne rodzaje pól to:

- BooleanField,
- CharField,
- DateTimeField,
- EmailField,
- IntegerField,

- ChoiceField.

Niektóre przydatne widgety to:

- Textarea,
- HiddenInput,
- PasswordInput,
- TextInput,
- CheckboxInput,
- Select.

Pole ChoiceField (związane z widgetem Select) powoduje stworzenie pola <select><option ...></select>. Tworząc go jego konstruktorowi musimy przekazać listę możliwości do wyboru. Argument ten powinien być listą dwuelementowych krotek zawierających nazwy i wartości opcji, spośród których będzie można wybierać.

Podobne do ChoiceField jest pole ModelChoiceField - pozwala ono wybrać wiersz z tabelki (modelu). Tworząc pole tego typu jego konstruktorowi przekazujemy argument queryset zawierający obiekt QuerySet związany z modelem, z którego chcemy wybierać: wybór = forms.ModelChoiceField(queryset=MojModel.objects.all()). Jeśli w widoku stworzymy instancję naszej klasy z formularzem i przekażemy ją szablonowi, może on wyrysować szablon po prostu wypisując go: {{ formularz }}. To dlatego, że metoda __str__ klasy Form wyołuje jej metodę as_table. Metoda ta wypisuje poszczególne pola formularza umieszczając je w sekcjach <tr> i <td>. Wypisaniem znaczników <table> i <form> musimy zająć się sami:

```
1 <form method="post" action="">
2 <table>{{ formularz }}</table>
3 <input type="submit" />
4 </form>
```

Stworzona przez nas klasa dziedzicząca po Form potrafi też walidować dane. Tworząc nowy obiekt stworzonej przez nas klasy możemy jako argument przekazać konstruktorowi słownik zawierający dane dla formularza (wzięte na przykład z request.POST). Tak stworzony formularz jest związany z danymi. Można sprawdzić, czy dane są poprawne:

```
1 formularz.is_valid()
```

Po zwalidowaniu dane związane z formularzem zostają znormalizowane. Możemy je znaleźć w atrybutie cleaned_data:

```
1 formularz.cleaned_data
```

Kiedy spróbujemy wyrysować formularz związany z danymi, będzie on wypełniony tymi danymi. Jeśli dane te nie walidują się, że wypełnione pola zostaną odpowiednio oznaczone. <https://docs.djangoproject.com/en/2.1/>

Ćwiczenie

1. Napisz prosty widok, który tworzy formularz i go pokazuje. Użyj różnych rodzajów pól i różnych widgetów.

Klasa ModelForm

Zauważmy, że stworzony przez nas formularz jest blisko związany z modelem Komentarz. Pola formularza odpowiadają polom modelu, a wysłanie formularza powoduje stworzenie i zapisanie nowego obiektu klasy Komentarz. W takiej sytuacji możemy użyć klasy ModelForm, która pozwala stworzyć formularz na podstawie modelu:

```
1 class MójFormularz(forms.ModelForm):
2     class Meta:
3         model = MójModel
```

Z tak zdefiniowanego formularza możemy korzystać w zwykły sposób: rysować go, walidować, przeglądać oczyszczone dane. Każde pole tak stworzonego formularza powstaje na podstawie jednego pola modelu. Od rodzaju pola modelu zależy rodzaj pola formularza - pole CharField modelu powoduje stworzenie pola CharField w formularzu, pole TextField modelu powoduje stworzenie w formularzu pola CharField z widgetem Textarea itp. Jeśli potrzebujemy, możemy nakazać, aby określone pole formularza było określonego rodzaju:

```
1 class MójFormularz(forms.ModelForm):
2     wiek = forms.FloatField()
3     class Meta:
4         model = MójModel
```

Formularz zawiera wszystkie pola modelu. Jeśli chcemy, by zawierał tylko niektóre pola, możemy to ustawić:

```
1 class MójFormularz(forms.ModelForm):
2     wiek = forms.FloatField()
3     class Meta:
4         model = MójModel
5         fields = ('imie', 'wiek')
```

Jeśli chcemy, żeby formularz nie zawierał niektórych pól modelu, również możemy to zrobić:

```
1 class MójFormularz(forms.ModelForm):
2     wiek = forms.FloatField()
3     class Meta:
4         model = MójModel
5         exclude = ('nazwisko',)
```

Na formularzu zawierającym dane możemy wywołać metodę save(). Stworzy ona instancję odpowiedniego modelu, po czym zapisze ją do bazy. Jeśli chcemy przed zapisaniem obiektu dokonać w nim jakichś zmian, możemy metodzie save() przekazać argument commit=False. Tak wywołana metoda save() nie zapisze stworzonego obiektu, ale go zwróci:

```
1 o = formularz.save(commit=False)
2 o.haslo_dostepu = 'tralala'
3 o.save()
```

Serwery aplikacji i http

Serwer developerski

Django - jak już wiemy dostarcza nam w postaci serwera deweloperskiego narzędzie, dzięki któremu możemy sprawdzić jak nasza aplikacja działa w przeglądarce internetowej. Polecenie

```
1 python manage.py runserver
```

uruchomi taki serwer lokalnie - na adresie 127.0.0.1:8000. Usługa nie będzie dostępna z zewnętrznych komputerów. Można to sprawdzić. Po pierwsze należy sprawdzić adres naszego urządzenia, np:

```
1 $ ifconfig
2 wlp3s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
3         inet 192.168.8.105 netmask 255.255.255.0 broadcast
4             192.168.8.255
5             inet6 fe80::b2fc:6239:c96e:57f9 prefixlen 64 scopeid <link>
6             ether 94:53:30:26:9e:07 txqueuelen 1000 (Ethernet)
7             RX packets 70862 bytes 65848967 (65.8 MB)
8             RX errors 0 dropped 0 overruns 0 frame 0
9             TX packets 46379 bytes 10515812 (10.5 MB)
9             TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Z drugiego komputera w naszej sieci powinniśmy móc puścić „pinga” do naszej maszyny

```
1 $ ping 192.168.8.105
2 PING 192.168.8.105 (192.168.8.105) 56(84) bytes of data.
3 64 bytes from 192.168.8.105: icmp_seq=1 ttl=64 time=0.073 ms
4 64 bytes from 192.168.8.105: icmp_seq=2 ttl=64 time=0.065 ms
5 64 bytes from 192.168.8.105: icmp_seq=3 ttl=64 time=0.068 ms
```

Nasza maszyna odpowiada

Jeśli spróbujemy teraz wejść w przeglądarkę na adres naszej maszyny na porcie na którym serwujemy aplikację: <http://192.168.8.105:8000>

to dostaniemy błąd wskazujący na to, że nie udało się nawiązać połączenia:

Na dobrą sprawę wpisanie tego adresu nawet na naszej lokalnej maszynie powinno zakończyć się tak samo. Możemy uruchomić program nieco inaczej:

```
1 python manage.py runserver 0.0.0.0:8000
```

W tym momencie nasza aplikacja powinna być już dostępna także spoza naszego komputera. Podobny wynik otrzymamy wpisując:

```
1 python manage.py runserver 192.168.8.105:8000
```

Gdzie oczywiście 192.168.8.105 to jest adres IP naszego komputera. 0.0.0.0 - to skrót, który umożliwia przypięcie apki do wszystkich adresów, na których może być widoczna nasza maszyna.

Zastosowania tego serwera deweloperskiego do celów produkcyjnych nie zaleca się nawet w przypadku prostych, wewnętrznych aplikacji. Zgodnie z dokumentacją:

„DO NOT USE THIS SERVER IN A PRODUCTION SETTING. It has not gone through security audits or performance tests. (And that's how it's gonna stay. We're in the business of making Web frameworks, not Web servers, so improving this server to be able to handle a production environment is outside the scope of Django.)”

<https://docs.djangoproject.com/en/2.2/ref/django-admin/#runserver>

Gunicorn

Serwer wbudowany w Django jest wygodny w użyciu, ale jak już wspomniano wyżej nie nadaje się do zastosowań produkcyjnych. Popularne serwery WWW takie jak nginx, czy Apache wymagają zdobycia dodatkowej wiedzy. Ich konfiguracja i użycie może okazać się dość wymagającym zajęciem. Kusi nas wtedy prostota serwera deweloperskiego. Dobrym rozwiązaniem może być skorzystanie wtedy z takiego rozwiązania jak Gunicorn. Przy generowaniu projekty przez aplikację django-admin tworzony jest też plik `wsgi.py`. Moduł ten zawiera funkcję `application`, która jest odpowiedzialna za komunikację Django ze światem zewnętrznym. Funkcja ta wywołana z odpowiednimi argumentami, w których umieszczone są informacje o żądaniu HTTP, zwróci odpowiedź na to żądanie. Dzięki temu w dość łatwy sposób można sprawić by serwer WWW przekazał żądanie do Django. W przypadku Gunicorna to on sam może przejąć na siebie rolę takiego serwera WWW. Czasem w literaturze można spotkać jeszcze podział na serwery aplikacji i serwery WWW. Ten podział jest raczej historyczny. Obecnie większość serwerów łączy w sobie cechu obu tych narzędzi. Aby skorzystać z Gunicorna, należy go najpierw zainstalować. Z uwagi na to, że jest to narzędzie napisane w Pythonie instalujemy go jak większość modułów pythonowych

```
1 pip install gunicorn
```

Po zainstalowaniu należy go jeszcze uruchomić. Założymy, że nasz projekt nazywa się example. Przechodzimy do głównego katalogu tego projektu (tam gdzie znajduje się plik manage.py) i wywołujemy komendę: gunicorn example.wsgi:application Możemy też np. uruchomić serwer z większą ilością workerów i na jakimś konkretnym adresie IP oraz porcie

```
1 gunicorn -w 2 -b 192.168.0.14:8000 example.wsgi:application
```

Zadziała tu też poznany wcześniej skrót 0.0.0.0. Warto zwrócić uwagę, że ścieżka do pliku settings wskazana jest przy użyciu „.” a nie systemowego „”. Po wskazaniu nazwy modułu użyty jest znak „.” a po nim nazwa funkcji odpowiedzialnej za komunikację poprzez wsgi. Po więcej możliwości warto zjrzeć do dokumentacji gunicorna <https://github.com/benoitc/gunicorn>

Apache + mod_wsgi

Apache to jeden z najpopularniejszych serwerów HTTP. Jest to dojrzały i sprawdzony serwer dostępny na wielu platformach, systemach operacyjnych. Przez wiele lat jednym z najpopularniejszych stacków technologicznych było użycie połączenia Apache + PHP + MySql. Serwery ten dobrze jednak sprawdza się także w połączeniu z językiem Python. Choć w tym przypadku konfiguracja środowiska jest znacznie bardziej skomplikowana. Początkowo korzystanie ze skryptów Pythona - jak i wielu innych języków realizowane było przy pomocy CGI (np.: <https://www.linux.com/blog/configuring-apache2-run-python-scripts>). Z czasem powstawały specjalne dodatki do Apache tzw. mody, które upraszczały i standaryzowały sposób w jaki Apache mógł serwować aplikacje pythonowe (np. mod_python). Obecnie najpopularniejszym standardem do komunikacji z aplikacjami pythonowymi jest wsgi - jest to standard mówiący, jak ma działać omawiana wcześniej funkcja application (jakie ma przyjmować argumenty, co będą zawierać te argumenty, co ma ona zwracać, jak ma się zachowywać). Django nie jest jedynym pythonowym frameworkiem zgodnym ze standardem WSGI - ten protokół obsługuje też CherryPy, Flask, TurboGears, Pylons i wiele innych frameworków. WSGI nie jest jedynym protokołem pozwalającym serwerowi WWW uruchamiać programy pythonowe, ale jest najbardziej popularny. Aby Apache umiał rozmawiać z Django, musimy doinstalować mu dodatek obsługujący któryś z protokołów pozwalających serwerowi uruchamiać programy pythonowe. Kiedyś popularny był dodatek mod_python (nie jest on zgodny ze standardem WSGI, wywołuje aplikację pythonową w swój własny sposób), ale obecnie jest on mniej popularny. Gdybyśmy z jakiegoś powodu mimo to chcieli użyć tego dodatku, musimy zainstalować pakiet libapache2-mod-python (przynajmniej tak nazywa się ten pakiet pod Debianem). Następnie do httpd.conf dopisujemy:

```
1 <Location "/śśśjakacieka">
2     SetHandler python-program
3     PythonHandler django.core.handlers.modpython
4     SetEnv DJANGO_SETTINGS_MODULE mójprojekt.settings
5     PythonDebug On
```

```
6     PythonPath "ś['żcieka do mojego projektu'] + sys.path"
7 </Location>
```

Przykładowo, jeśli mamy djangowy projekt jazzy umieszczony w katalogu /home/adam/mojprojekty/jazzy i chcemy, żeby był on serwowany spod URL-i zaczynających się od http://localhost/pliszka, piszemy:

```
1 <Location "/pliszka">
2     SetHandler python-program
3     PythonHandler django.core.handlers.modpython
4     SetEnv DJANGO_SETTINGS_MODULE jazzy.settings
5     PythonDebug On
6     PythonPath "[ '/home/adam/mojprojekty/jazzy' ] + sys.path"
7 </Location>
```

Częściej jednak używa się modułu mod_wsgi - jak sama nazwa wskazuje, obsługuje on protokół WSGI. Musimy zacząć od zainstalowania tego dodatku. W tym celu ściągamy go z <http://code.google.com/p/modwsgi/> (na przykład z http://modwsgi.googlecode.com/files/mod_wsgi-2.0c4.tar.gz). Następnie, żeby móc go zbudować, instalujemy pakiet zawierający plik apxs (lub apxs2) - pod Debianem ten pakiet nazywa się httpd-devel (pakiet może też nazywać się apache2-threaded-dev). Teraz już możemy zbudować mod_wsgi w zwykły sposób, czyli poleceniami:

```
1 # ./configure
2 # make
3 # make install
```

Teraz do httpd.conf dopisujemy linijkę, która spowoduje załadowanie tego modułu: LoadModule wsgi_module modules/mod_wsgi.so Teraz musimy ustawić, jakie URL-e mają uruchamiać skąd wziętą funkcję application. Jak pamiętamy, każdy projekt Django ma swoją własny moduł wsgi z funkcją application. Ale na początek, przy testach, wygodniej będzie napisać własną, bardzo prostą funkcję application. Tworzymy w katalogu /home/adam/ plik hello.py o takiej treści:

```
1 def application(env, start_response):
2     start_response('200 OK', [('Content-Type','text/html')])
3     return ["Hello World"]
```

Teraz ustawiamy, żeby ta funkcja application była wywoływana przy żądaniach dotyczących URL-i z katalogu http://localhost/pliszka. W tym celu do httpd.conf dopisujemy:

```
1 WSGIScriptAlias /pliszka /home/adam/hello.py
```

Teraz kiedy zrestartujemy Apacza i wejdziemy pod adres http://localhost/pliszka, powinniśmy zobaczyć napis hello world.

Kiedy to działa, przerabiamy wpis WSGIScriptAlias tak, żeby wskazywał na plik wsgi.py wygenerowany przez Django w katalogu naszego projektu. Po tej zmianie wszedłszy pod adres

<http://localhost/pliszka> powinniśmy zobaczyć naszą aplikację zrobioną w Django.

Na razie mod_wsgi działa w trybie wembedowującym. Żeby działało w trybie demonicznym, trzeba w httpd.conf wpisać:

```
1 WSGIDaemonProcess jakasnazwa threads=5 maximum-requests=10000
2 WSGIScriptAlias /pliszka ścieka-do-projektu/wsgi.py
3 WSGIProcessGroup jakasnazwa
```

Dyrektyna WSGIDaemonProcess każe Apaczowi stworzyć nową grupę procesów o nazwie jakasnazwa. Dyrektywa WSGIProcessGroup każe serwować nasz moduł wsgi.py tą grupą procesów.

Jeśli chcemy, żeby ta grupa procesów stworzona przez dyrektywę WSGIDaemonProcess działała na użytkowniku innym niż domyślny www-data, dodajemy przy niej parametry user i group:

```
1 WSGIDaemonProcess jakasnazwa user=piotr group=piotr threads=5 maximum-
   requests=10000
2 (...)
```

Jeśli chcemy, żeby nasza aplikacja była widoczna tylko pod pewną domeną, tworzymy w zwykły apaczowy sposób wirtualkę:

```
1 <VirtualHost *:80>
2   ServerName jazzy.pwr
3   WSGIScriptAlias / /home/piotr/mpa.py
4   WSGIDaemonProcess procespiotra user=piotr group=piotr
5   WSGIProcessGroup procespiotra
6 </VirtualHost>
```

Nginx

Apache to projekt o dużych możliwościach. Jednakże jest też dość duży, ciężki i trudny w konfiguracji. Z kolei serwerom aplikacji takim jak wspomniany wcześniej Gunicorn, czy uWSGI brakuje czasem niektórych możliwości oferowanych przez standardowe serwery WWW. W przypadku bardziej wymagających aplikacji stosuje się często podejście polegające na połączeniu prostego, wydajnego serwera Http z serwerem aplikacji WSGI. Bardzo często wybierany jest wtedy jako serwer WWW - nginx. Ma on za zadanie serwowanie statycznych treści - takich jak różnego rodzaju pliki js, czy obrazki mediów oraz przekazywanie zapytań do serwera aplikacji. Poniżej znajdziemy przykładowy opis konfiguracji nginxa i uWSGI (pod Debianem) tak, żeby serwowały aplikację napisaną w Django (napisany na podstawie http://uwsgi-docs.readthedocs.org/en/latest/tutorials/Django_and_nginx.html). Warto mieć świadomość, że te technologie często się starzeją, ewoluują i przy deploymencie najlepiej jest posiłkować się aktualną dokumentacją projektów i narzędzi z których zamierzamy skorzystać. Zaczynamy od zainstalowania potrzebnych pakietów:

```
1 # apt-get update
2 # apt-get install python-virtualenv build-essential python-dev nginx
   tree
```

Następnie tworzymy konto użytkownika django - to na nim będziemy prowadzić eksperymenty:

```
1 # useradd -m django
```

Jako użytkownik django tworzymy wirtualne środowisko i aktywujemy je:

```
$ virtualenv test1 $ source test1/bin/activate
```

Teraz polecienniem pip instalujemy najnowszą wersję Django i uwsgi (ten krok trwa około pięciu minut):

```
1 (test1)django@core ~$ pip install Django uwsgi
```

Tworzymy prostą aplikację zgodną ze standardem WSGI:

```
1 (test1)django@core ~$ cat > test.py <<KONIEC
2
3 def application(env, start_response):
4     start_response('200 OK', [('Content-Type','text/html')])
5     return ["Hello World"]
6 KONIEC
```

Uruchamiamy uwsgi każąc, żeby serwował tę aplikację:

```
1 (test1)django@core ~$ uwsgi --http :8000 --wsgi-file test.py
```

Wchodzimy przeglądarką pod odpowiedni adres i widzimy, że nasza aplikacja działa:

Tworzymy projekt djangowy:

```
1 (test1)django@core ~$ django-admin.py startproject mysite
```

Uruchamiamy serwer WWW wbudowany w Django, żeby serwował ten projekt:

```
1 (test1)django@core ~$ cd mysite/
2 (test1)django@core ~/mysite$ python manage.py runserver 0.0.0.0:8000
```

Zagładamy pod odpowiedni adres, żeby zobaczyć, że projekt jest serwowany:

Zatrzymujemy djangowy serwer (wciskając ctrl+c). Uruchamiamy uWSGI i każemy mu serwować nasz projekt:

```
1 (test1)django@core ~/mysite$ uwsgi --http :8000 --module mysite.wsgi
```

Zagładamy pod odpowiedni adres, żeby zobaczyć, że projekt nadal jest serwowany:

Zatrzymujemy uwsgi (wciskając ctrl+c). Uruchamiamy nginxa:

```
1 # /etc/init.d/nginx start
```

Zagłädamy pod odpowiedni adres (nginx domyślnie słucha na porcie 80), żeby przekonać się, że nginx działa:

Konfigurujemy nginxa, żeby z URL-i /media i /static serwował statyczne pliki z odpowiednich podkatalogów djangowego projektu, a żądania na URL / przekazywał do uWSGI (w ustawieniu server_name musimy wstawić nasz adres IP).

```
1 (test1)$ ~/mysite$ wget https://raw.githubusercontent.com/nginx/nginx/
      master/conf/uwsgi_params
2
3
4 (test1)$ ~$ cat > mysite_nginx.conf <<KONIEC
5     # mysite_nginx.conf
6     # the upstream component nginx needs to connect to
7     upstream django {
8         # server unix:///path/to/your/mysite/mysite.sock; # for a file
9             socket
10        server 127.0.0.1:8001; # for a web port socket (we'll use this
11            first)
12    }
13    # configuration of the server
14    server {
15        # the port your site will be served on
16        listen      8000;
17        # the domain name it will serve for
18        server_name 192.168.0.102; # substitute your machine's IP
19            address or FQDN
20        charset    utf-8;
21        # max upload size
22        client_max_body_size 75M;   # adjust to taste
23        # Django media
24        location /media {
25            alias /home/django/mysite/media; # your Django project's
26                media files - amend as required
27        }
28        location /static {
29            alias /home/django/mysite/static; # your Django project's
30                static files - amend as required
31        }
32        # Finally, send all non-media requests to the Django server.
33        location / {
34            uwsgi_pass  django;
35            include   /home/django/mysite/uwsgi_params; # the
36                uwsgi_params file you installed
37        }
38    }>>>KONIEC
```

```
33  
34  
35  
36     root@core /home/django/mysite# ln -s `realpath mysite_nginx.conf` /  
          etc/nginx/sites-enabled/
```

Sprawdzamy, czy działa serwowanie plików statycznych. Tworzymy jeden taki plik:

```
1     (test1)django@core ~/mysite$ mkdir static  
2     (test1)django@core ~/mysite$ cat > static/test.html <<KONIEC  
3         raz <b>dwa</b>
```

Restartujemy nginx:

```
1 # /etc/init.d/nginx restart
```

Zagłädamy pod odpowiedni adres, żeby przekonać się, że plik jest serwowany:

Ustawiamy w naszym projekcie, że statyczne pliki wszystkich aplikacji mają być zbierane do tego katalogu serwowanego statycznie przez nginxa:

```
1 (test1)django@core ~/mysite$ cat >> mysite/settings.py <<KONIEC  
2 STATIC_ROOT = os.path.join(BASE_DIR, "static/")  
3  
4  
5 (test1)django@core ~/mysite$ yes yes | python manage.py collectstatic
```

Zagłädamy pod odpowiedni adres, żeby sprawdzić, czy nginx serwuje statyczne pliki aplikacji admin:

Sprawdzamy, czy nginx żądania przychodzące pod URL / przekazuje uWSGI. Uruchamiamy uWSGI:

```
(test1)django@core ~$ uwsgi --socket :8001 --wsgi-file test.py
```

Zagłädamy pod odpowiedni adres, żeby sprawdzić, czy dostaniemy odpowiedź z uWSGI:

Zmieniamy konfigurację nginxa, żeby żądania do uWSGI przekazywać lokalnym socketem, a nie po TCP. W tym celu w mysite_nginx.conf odpowiedni fragment przerabiamy tak:

```
1 server unix:///home/django/mysite/mysite.sock; # for a file socket  
2 #server 127.0.0.1:8001; # for a web port socket (we'll use this first)
```

Restartujemy nginxa:

```
1 # /etc/init.d/nginx restart  
2 Uruchamiamy uWSGI żąkac mu ćkomunikowa ęsi przez ten sam socket:  
3 (test1)django@core ~/mysite$ uwsgi --socket mysite.sock --chmod-socket  
=666 --wsgi-file ../test.py
```

Zagłädamy pod odpowiedni adres, żeby sprawdzić, czy znowu dostaniemy odpowiedź z uWSGI:

Sprawdzamy, czy to samo będzie działać, kiedy uWSGI będzie serwować nie hello world, tylko nasz projekt djangowy. Zatrzymujemy uWSGI (wciskając **ctrl+c**), po czym uruchamiamy go znowu, tak:

```
1 (test1)django@core ~/mysite$ uwsgi --socket mysite.sock --module mysite.wsgi --chmod-socket=666
```

Zaglądamy pod odpowiedni adres, żeby upewnić się, że dostaniemy odpowiedź od Django:

Żeby nie trzeba było przy każdym uruchomieniu uWSGI przekazywać mu tych wszystkich opcji, tworzymy plik konfiguracyjny:

```
1 (test1)django@core ~/mysite$ cat > mysite_uwsgi.ini <<KONIEC
2 [uwsgi]
3 chdir = /home/django/mysite
4 module = mysite.wsgi
5 home = /home/django/test1
6 master = true
7 processes = 10
8 socket = /home/django/mysite/mysite.sock
9 chmod-socket = 666
10 vacuum = true
11 KONIEC
```

Uruchamiamy uWSGI z tym plikiem konfiguracyjnym:

```
(test1)django@core ~/mysite$ uwsgi -ini mysite_uwsgi.ini
```

Zaglądamy pod odpowiedni adres, żeby upewnić się, że dostaniemy odpowiedź od Django:

Teraz uruchomimy taką samą konfigurację, ale z uWSGI zainstalowanym normalnie, z repozytorium. Instalujemy uWSGI z repozytorium:

```
1 # apt-get install uwsgi
```

Uwsgi z repozytorium nie rozumie z pudełka opcji **-module**. Dlatego instalujemy jeszcze:

```
1 # apt-get install uwsgi-plugin-python
```

Uruchamiamy uWSGI polecienniem:

```
1 django@core ~$ uwsgi --http :8080 --plugin python --wsgi-file test.py
```

Zaglądamy pod odpowiedni adres, żeby upewnić się, że uWSGI odpowiada:

Teraz uruchamiamy uWSGI z repozytorium z naszym plikiem konfiguracyjnym:

```
1 django@core ~/mysite$ uwsgi --plugin python --ini mysite_uwsgi.ini
```

Zaglądamy pod adres obsługiwany przez nginxa (który przekaże żądanie do uWSGI, który obsługa je djangiem):

Teraz w /etc/uwsgi/apps-available/ robimy symlink do mysite_uwsgi.ini:

```
1 # ln -s /home/django/mysite/mysite_uwsgi.ini /etc/uwsgi/apps-available/
```

A w /etc/uwsgi/apps-enabled/ robimy symlink do tamtego symlinku:

```
1 # ln -s /etc/uwsgi/apps-available/mysite_uwsgi.ini /etc/uwsgi/apps-enabled/
```

Przerabiamy mysite_uwsgi.ini tak, żeby soket był tworzony w /home/django/mysite/sokety/:

```
1 socket = /home/django/mysite/sokety/mysite.sock
```

Podobnie przerabiamy mysite_nginx.conf:

```
1 server unix:///home/django/mysite/sokety/mysite.sock;
```

Tworzymy katalog /home/django/mysite/sokety/ i nadajemy mu uprawnienia 0777:

```
1 django@core ~./mysite$ mkdir /home/django/mysite/sokety/
2 django@core ~./mysite$ chmod 0777 /home/django/mysite/sokety/
```

Restartujemy uWSGI i nginxa:

```
1 # /etc/init.d/nginx restart
2 # /etc/init.d/uwsgi restart
```

Zagładamy pod odpowiedni adres, żeby upewnić się, że wszystko działa:

Na koniec możemy zainstalować Django z repozytorium:

```
1 # apt-get install python-django
```

po czym z pliku mysite_uwsgi.ini usunąć linię:

```
1 home = /home/django/test1
```

Deployment

Kiedy nasz projekt jest gotowy, możemy przy pomocy narzędzia dostarczonego przez Django sprawdzić jakie problemy związane z deploymentem powinniśmy rozwiązać.

```
1 $ python manage.py check --deploy
```

Polecenie te może zwrócić wynik podobny do tego:

```
1 System check identified some issues:
```

```
2
3 WARNINGS:
4 ?: (security.W004) You have not set a value for the SECURE_HSTS_SECONDS
   setting. If your entire site is served only over SSL, you may want
   to consider setting a value and enabling HTTP Strict Transport
   Security. Be sure to read the documentation first; enabling HSTS
   carelessly can cause serious, irreversible problems.
5 ?: (security.W008) Your SECURE_SSL_REDIRECT setting is not set to True.
   Unless your site should be available over both SSL and non-SSL
   connections, you may want to either set this setting True or
   configure a load balancer or reverse-proxy server to redirect all
   connections to HTTPS.
6 ?: (security.W012) SESSION_COOKIE_SECURE is not set to True. Using a
   secure-only session cookie makes it more difficult for network
   traffic sniffers to hijack user sessions.
7 ?: (security.W016) You have 'django.middleware.csrf.CsrfViewMiddleware'
   in your MIDDLEWARE, but you have not set CSRF_COOKIE_SECURE to True
   . Using a secure-only CSRF cookie makes it more difficult for
   network traffic sniffers to steal the CSRF token.
8 ?: (security.W018) You should not have DEBUG set to True in deployment.
9 ?: (security.W022) You have not set the SECURE_REFERRER_POLICY setting.
   Without this, your site will not send a Referrer-Policy header. You
   should consider enabling this header to protect user privacy.
10
11 System check identified 6 issues (0 silenced).
```

Warto skorzystać z tych porad. Oprócz nich należy też zadbać o to, by dane wrażliwe, takie jak wszelkiego rodzaju hasła, dane dostępowe, czy klucze (SECRET_KEY) nie były wprost wpisane w naszym kodzie.

Taki kod trafia często do publicznego repozytorium i te dane mogą być wtedy z niego odczytane. To poważna luka w zabezpieczeniach. Poza tym konfiguracje na różnych maszynach mogą się po prostu różnić. Dlatego lepiej zastosować jakieś inne rozwiązanie. Proponujemy dwa rozwiązania. Można oprzeć się o zmienne środowiskowe lub jakiś plik, w którym umieścimy odpowiednie konfiguracje.

Zmienne środowiskowe ustawiamy poleceniem export, np.:

```
1 $ export SECRET_KEY=Verysecret!@#!@#key
```

Odczytanie ze zmiennej środowiskowej w pliku settings.py:

```
1 import os
2 SECRET_KEY = os.environ.get('SECRET_KEY', "key testowy")
```

Odczytanie z pliku w settings.py:

```
1 with open('/etc/secret_key.txt') as f:
2     SECRET_KEY = f.read().strip()
```

Oczywiście to nie jedynie możliwości, ale większość sprowadza się do jakiejś wariacji na ich temat. Jeśli zdecydujesz się na plik, zadbaj o to, by nie dodawać go do repozytorium.

Zadziałajmy teraz lokalnie dla zmiennej DEBUG, w terminalu:

```
1 $ export DEBUG=False
```

W settings.py:

```
1 DEBUG = os.environ.get("DEBUG", False)
```

Jeśli flaga ta ma wartość False, to muszą znajdować się jakieś wpisy w ALLOWED_HOSTS. Na razie są to:

```
1 ALLOWED_HOSTS = ['localhost', '127.0.0.1', 'testserver']
```

W podobny sposób, jak z SECRET_KEY i DEBUG, powinniśmy zadziałać z wszelkimi danymi dostępowymi do baz danych. My korzystamy z SQLite i tu dostęp nie jest wymagany w konfiguracji.

Po ustaleniu DEBUG na False może okazać się, że wszystkie nasze statyczne pliki nie działają. Często jest tak, że takie pliki serwowane są z pominięciem Django. Wymaga to odpowiedniej konfiguracji serwera HTTP. Wygodnie jest też, kiedy pliki są zgromadzone w jednym miejscu. By to zrobić, w settings.py dodajemy ustawienie:

```
1 STATIC_ROOT = os.path.join(BASE_DIR, "static")
```

Następnie wykonujemy polecenie:

```
1 $ python manage.py collectstatic
```

W tym miejscu może wystąpić błąd, polegający na braku ustawienia STATIC_URL.

```
1 django.core.exceptions.ImproperlyConfigured: You're using the
      staticfiles app without having set the required STATIC_URL setting.
```

By ten problem rozwiązać, należy dodać w settings ustawienie:

```
1 STATIC_URL = "/static/"
```

Takie polecenie warto wykonać na docelowym środowisku. Jeśli je wywołamy, powstanie folder static z kopiami wszelkich plików statycznych z naszych aplikacji.

Lokalnie możemy jeszcze sprawdzić, czy nasza aplikacja działa poprawnie z jakimś serwerem WSGI. *Web Server Gateway Interface* to standard komunikacji z aplikacjami webowymi napisanymi w Django. Przykłady takiego serwera to np. Gunicorn czy uWSGI. Są też dodatki do takich serwerów jak np. Apache 2 (mod_wsgi).

Dość szybkim i prostym rozwiązaniem może okazać się Gunicorn. Jest to serwer napisany w Pythonie. Możemy go zainstalować przy pomocy pip.

```
1 $ pip install gunicorn
```

Uruchomienie aplikacji sprawdza się do wydania polecenia `gunicorn` z podaniem dostępu do pliku `wsgi.py`. Odpalając je z katalogu zawierającego plik `manage.py`, polecenie może wyglądać tak:

```
1 $ gunicorn jazzy.wsgi
```

Aplikacja powinna działać, choć pliki statyczne nadal nie będą widoczne.

Jeśli chcemy, możemy to naprawić, np. poprzez dodatek `dj-static`

Zrobimy to w trzech krokach:

```
1 $ pip install dj-static
```

Dopisujemy to też do `requirements.txt`, a następnie upewniamy się, że w `settings.py` gdzieś na dole są takie linie:

```
1 STATIC_URL = '/static/'  
2 STATIC_ROOT = os.path.join(BASE_DIR, "static")
```

Zmieniamy też plik `jazzy/wsgi.py`.

```
1 """  
2 WSGI config for jazzy project.  
3  
4 It exposes the WSGI callable as a module-level variable named ``  
5 application``.  
6  
7 For more information on this file, see  
8 https://docs.djangoproject.com/en/3.0/howto/deployment/wsgi/  
9 """  
10 import os  
11  
12 from django.core.wsgi import get_wsgi_application  
13 from dj_static import Cling  
14  
15 os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'jazzy.settings')  
16  
17 application = Cling(get_wsgi_application())
```

Jeśli wcześniej nie zrobiono `collectstatic`, to należy wykonać to teraz.

```
1 $ python manage.py collectstatic
```

Uruchamiamy serwer:

```
1 $ gunicorn jazzy.wsgi
```

Obrazki i statyki powinny działać.

Dodatki

Najprostsza aplikacja w Django

Oto opis, jak zrobić najprostszą aplikację w Django.

Zaczynamy od stworzenia katalogu:

```
1 $ mkdir hello
2 $ cd hello
```

Próbujemy uruchomić w nim serwer WWW wbudowany w Django, ale widzimy, że to nie działa:

```
1 $ django-admin runserver 0.0.0.0:8000
2 Traceback (most recent call last):
3   File "/usr/lib/python2.7/dist-packages/django/bin/django-admin.py",
4     line 5, in <module>
5     (...)
```

5 django.core.exceptions.ImproperlyConfigured: Requested setting USE_I18N
 , but settings are not configured. You must either define the
 environment variable DJANGO_SETTINGS_MODULE or call settings.
 configure() before accessing settings.

6 Jak widać, musimy stworzyć plik z ustawieniami i wskazać go zmiennej
 środowiskowej DJANGO_SETTINGS_MODULE. Robimy to – plik z ustawieniami
 na razie zostawiamy pusty:

```
7
8 $ touch settings.py
9
10 $ export DJANGO_SETTINGS_MODULE=settings
```

Próbujemy znowu uruchomić serwer i znowu napotykamy problem:

```
1 $ django-admin runserver 0.0.0.0:8000
2 Traceback (most recent call last):
3   File "/usr/lib/python2.7/dist-packages/django/bin/django-admin.py",
4     line 5, in <module>
5     (...)
```

5 ImportError: Could not import settings 'settings' (Is it on sys.path?
 Is there an import error in the settings file?): No module named
 settings

Jak widać, Django nie wie, gdzie ma szukać modułu settings. Ustawiamy zmienną środowiskową PYTHONPATH:

```
1 $ export PYTHONPATH=`pwd`
```

Próbowiemy znowu uruchomić serwer i znowu napotykamy problem:

```
1 $ django-admin runserver 0.0.0.0:8000
2 Traceback (most recent call last):
3   File "/usr/lib/python2.7/dist-packages/django/bin/django-admin.py",
4     line 5, in <module>
5     (...)
```

5 django.core.exceptions.ImproperlyConfigured: The SECRET_KEY setting must not be empty.

Zgodnie z komunikatem, w pliku z ustawieniami ustawiamy ustawienie SECRET_KEY:

```
1 $ cat > settings.py <<KONIEC
2 SECRET_KEY='xyz'
3 KONIEC
```

Próbowiemy znowu uruchomić serwer i znowu napotykamy problem:

```
1 $ django-admin runserver 0.0.0.0:8000
2 CommandError: You must set settings.ALLOWED_HOSTS if DEBUG is False.
```

Zgodnie z komunikatem, w pliku z ustawieniami ustawiamy ustawienie DEBUG:

```
1 $ cat >> settings.py <<KONIEC
2 DEBUG=True
3 KONIEC
```

Próbowiemy znowu uruchomić serwer:

```
1 $ django-admin runserver 0.0.0.0:8000
```

Tym razem problem jest subtelniejszy. Serwer co prawda startuje, ale kiedy próbujemy wejść przeglądarką pod odpowiedni adres, widzimy komunikat o błędzie:

```
1 A server error occurred. Please contact the administrator.
```

Dokładniejszy komunikat zobaczymy na konsoli, z której uruchomiliśmy serwer:

```
1      Traceback (most recent call last):
2        File "/usr/lib/python2.7/wsgiref/handlers.py", line 85, in run
3          self.result = application(self.environ, self.start_response)
4          (...)
```

5 AttributeError: 'Settings' object has no attribute 'ROOT_URLCONF'

Jak widać, musimy stworzyć moduł z ustawieniami, jakie URL-e mają prowadzić do jakiego widoku, i wskazać go ustawieniem ROOT_URLCONF w pliku z ustawieniami. Robimy to:

```
1 $ cat > urls.py <<KONIEC
2 from django.core.urlresolvers import RegexURLPattern
3 from django.http import HttpResponseRedirect
4 def hello(r):
5     return HttpResponseRedirect('hello world!')
6 urlpatterns = [RegexURLPattern('', hello)]
7 KONIEC
8
9 $ cat >> settings.py <<KONIEC
10 ROOT_URLCONF='urls'
11 KONIEC
```

Uruchamiamy serwer:

```
1 $ django-admin runserver 0.0.0.0:8000
```

Wchodzimy przeglądarką pod odpowiedni adres i widzimy, że tym razem działa:

```
1 hello world!
```

Sygnały

Czasem istnieje konieczność wykonania jakichś czynności przed, czy po jakimś istotnym zdarzeniu - np. przed save, albo po save. Wtedy bardzo pomocny jest mechanizm sygnałów. Zagadnienie to jest tu jedynie sygnalizowane. Po więcej szczegółów odsyłam do dokumentacji <https://docs.djangoproject.com/en/2.1/topics/signals/>

Redis

Redis często bywa używany w aplikacji Django jako forma pamięci podręcznej, cache aplikacji. Możemy w nim odkładać klucze i wartości dla tych kluczów. Wartościami mogą być też np. listy. Możemy tak odkądać pewne zdarzenia - np. powiększać licznik odwiedzin jakieś strony o później innym mechanizmem pobierać co jakiś czas te dane i zapisywać w bazie połączonej z Django. Dzięki temu baza danych nie jest nieustannie bombardowana różnego rodzaju zapytaniami i insertami i cała aplikacja zyskuje w aspekcie wydajnościowym. Po więcej informacji odsyłam do projektu i tutoriali: <https://redis.io/> <https://github.com/niwinz/django-redis> <https://realpython.com/caching-in-django-with-redis/>

Celery

Celery to narzędzie do asynchronicznej obsługi kolejek różnego rodzaju zadań, które mogą być zlecanie przez aplikację webową - w tym także taką napisaną w Django. Zadania, które są długotrwałe dobrze jest wykonywać w sposób asynchroniczny. Dzięki temu aplikacja nie oczekuje na ich zakończenie. Zarządzanie takimi zadaniami wygodnie jest przekazać do Celery, które zadba o to by zadania się wykonywały <http://docs.celeryproject.org/en/latest/django/first-steps-with-django.html> <https://rk.edu.pl/pl/asynchroniczne-zadania-celery-w-projektach-django/> <http://docs.celeryproject.org/en/latest/userguide/periodic-tasks.html>

REST

REST jest zbiorem reguł i dobrych praktyk budowania serwisów webowych wykorzystujących protokół HTTP.

REST można określić jako styl architektoniczny opisywany sześcioma warunkami: 1 - Uniform Interface (jednolity interfejs) 2 - Stateless (bezstanowość) 3 - Cacheable (możliwość cache'owania danych) 4 - Client-Server (model klient-serwer) 5 - Layered System (warstwowość) 6 - Code on Demand (kod na żądanie)

REST (*Representational State Transfer*) jest wzorcem opisującym sposób, w jaki powinna być tworzona architektura aplikacji rozproszonych pracujących w modelu klient-serwer. Nie jest to więc framework, nie jest to biblioteka. REST jest zbiorem pewnych dobrych praktyk, które mają pomóc budować funkcjonalne i zrozumiałe aplikacje. Zasady te opisane są w kilku punktach.

Client-Server – architektura typu klient-serwer

Aplikacja kliencka powinna być niezależna od serwera. Oznacza to dowolność w stosowaniu technologii. Aplikacja serwerowa może być np. napisana przy pomocy Pythona, a kliencka w innym języku.

Nic nie stoi na przeszkodzie, by było wiele aplikacji klienckich, np. na różne platformy mobilne, czy systemy operacyjne. Aplikacje te powinny być rozwijane niezależnie. Zmiana w kliencie nie powinna mieć wpływu na serwer. Nie może np. wymuszać zmian w bazie danych.

Z drugiej strony zmiany na serwerze też nie powinny wpływać na aplikacje klienckie. Można o tym myśleć w ten sposób, że API jest formą kontraktu, którego trzymają się obie strony. Zmiany po stronie serwera nie powinny więc np. usuwać końcówek, czy jakichś pól z przesyłanych odpowiedzi, dopóty dopóki obowiązuje dana wersja API. Możliwe jest natomiast rozwijanie, dodawanie nowych końcówek, czy dodanie nowych danych do tych istniejących.

Stateless (bezstanowość)

Zasada ta oznacza, że każde połączenie z serwerem powinno być całkowicie niezależne od poprzednich i następnych.

W wielu tradycyjnych aplikacjach, po zalogowaniu stan tzw. zdalnej sesji utrzymywany jest po stronie serwera. Raz ustanowiona sesja utrzymywana jest przez jakiś czas, np. do wylogowania się klienta. W tym czasie może on korzystać z różnego rodzaju usług świadczonych przez serwer. Ma to swoje zalety, ale i wady. Przy takim podejściu łatwiej jest utrzymywać centralnie interfejs na serwerze, trudniej jednak skalować serwer, który musi być dodatkowo obciążony dbaniem o utrzymywanie stanu aplikacji pomimo tych wszystkich sesji klienckich itd.

W podejściu bezstanowym klient musi przesłać komplet informacji potrzebnych do jej wykonania. Mowa tu nie tylko o danych dotyczących konkretnych zasobów, ale też informacji potrzebnych do autoryzacji. Odciąża to bardzo stronę serwerową, która musi się "skupić" jedynie na obsłudze pojedynczych requestów. Ułatwia to bardzo implementację, poprawia skalowalność i niezawodność.

Autoryzacja odbywa się w REST, głównie na dwa sposoby. Można użyć podstawowego schematu autoryzacji, w którym login i hasło przekazywane są w zakodowanej formie. Transport odbywa się w oparciu o nagłówki HTTP, w formie:

```
1 Authorization: Basic <credentials>
```

Podejście takie wymaga wysyłania loginu i hasła z każdym requestem.

W produkcyjnych zastosowaniach częściej jednak stosuje się podejście oparte o tokeny (np. JWT), które w zamian są wysyłane w requestach. Taki token autoryzuje żądania – ponownie w oparciu o nagłówek HTTP:

```
1 Authorization: Bearer <token>
```

Cache

Buforowanie danych tam, gdzie to możliwe (po stronie klienta).

W celach optymalizacyjnych dane powinny być cache'owane po stronie klienta. Serwer powinien informować go, czy takie dane mogą być buforowane i jeśli tak, to przez jaki okres. Dzięki temu można znacznie ograniczyć ilość zapytań do serwera.

Uniform Interface

Jednolity interfejs to sposób na uproszczenie architektury w komunikacji klient-serwer. Istotne są tu 4 zasady mówiące o tym, że:

1. Interfejs jest oparty na zasobach. Każdy taki zasób ma unikalny URI. Dane mogą być zwracane na wiele sposobów, na przykład w postaci XML, JSON, czy YAML. Nie musi to być forma, w jakiej dane są przechowywane w bazie i najczęściej tak nie jest.
2. Zasoby są manipulowane poprzez ich reprezentację. Po otrzymaniu zasobu od serwera klient, w oparciu o te dane, może modyfikować lub usuwać zasób z serwera – o ile tylko ma do tego uprawnienia.
3. Odpowiedzi serwera są samoapisujące się. Informują klienta, jak mają być obsługiwane, np. poprzez podanie wartości MIME type.
4. HATEOAS (*Hypermedia as the Engine of Application State*) – sterowanie powinno być możliwe poprzez dodatkowe informacje umieszczane w odpowiedzi. Te informacje to takie dodatkowe linki – hypermedia prowadzące do dodatkowych zasobów. Dzięki temu możliwe jest poruszanie się po API, nawet bez znajomości jego dokumentacji. Wystarczy eksplorować wystawione końcówki, by dowiedzieć się, co takie API oferuje. Np. jeśli mamy aplikację domowej biblioteki – możemy mieć tam informacje o książkach, które się w niej znajdują oraz o ich autorach. Główny endpoint aplikacji mógłby wyświetlić linki do listy książek i listy autorów, te z kolei mogłyby prowadzić do ich szczegółów, czy jakichś powiązanych zasobów. Np. jeśli wejdziemy w szczegóły autora, to moglibyśmy tam oczekiwąć linku do listy jego książek.

```
1 {
2     "timestamp": "2017-09-10T12:20:22",
3     ...
4     "_links": [
5         { "rel": "books", "href": "/v1/books", "method": "GET" },
6         { "rel": "authors", "href": "/v1/authors", "method": "GET" }
7     ]
8 }
```

Layered System

Dzielnie architektury REST na współdziałające ze sobą warstwy. Klient, komunikując się z serwerem, nie musi wiedzieć nic o jego architekturze. Interesuje go otrzymanie zasobu, a nie to, w jaki sposób jego *request* został przetworzony.

Code on Demand

Pozwala na przesyłanie nowych fragmentów kodu poprzez API. Jest to opcjonalna reguła. W kodzie może być zawarta pewna logika do wykonania.

W aplikacjach budowanych w oparciu o REST dużą rolę odgrywają także inne metody HTTP. API projektuje się w dużej mierze także w oparciu o nie. Pozwala to na sterowanie zasobami bez konieczności tworzenia rozbudowanych końcówek API. W podejściach takich jak RPC (*Remote Procedure Call*) do tego, by wykonywać operacje CRUD (*Create/Read/Update/Delete*) moglibyśmy tworzyć kilka końcówek, a co za tym idzie także widoków, które za nie odpowiadają. Np.

1 /get_posts_list	- pobranie listy postów
2 /create_post	- utworzenie nowego postu
3 /get_post_details?id=1	- pobranie szczegółów postu
4 /delete_post?id=1	- usunięcie postu
5 /update_post?id=1&published=y	- update postu

Od użytkowników API wymaga to znajomości procedur, które chcemy wywołać. Stosowane nazwy mogą bardzo się różnić w projektach.

W podejściu REST-owym jest to ustandardyzowane. Analogią powyższego przykładu mogą być poniższe dwie końcówki. Wybrane zmiany osiągniemy, stosując odpowiednie metody HTTP.

1 GET /posts/	- pobranie listy postów
2 POST /posts/	- utworzenie nowego postu
3 GET /posts/<id>	- pobranie szczegółów postu
4 DELETE /posts/<id>	- usunięcie postu
5 PUT /posts/<id>	- update postu
6 PATCH /todos/<id> konkretnych pól)	- update wybranego postu (nadpisanie

Prawda, że tak jest dużo lepiej? Zasada jest tu dość prosta. Trzon adresu stanowi przeważnie liczba mnoga jakiegoś rzecznika, najlepiej w języku angielskim. Zatem gdybyśmy mieli książki, byłoby to /books/, a w przypadku autorów /authors/.

GET na takiej końcówce powinien zwrócić listę zasobów – książek, autorów czy zadań. Dodanie do adresu identyfikatora pozwoli nam operować na konkretnym zasobie. Jak widać, każde zadanie będzie miało tu swój unikalny URL. Bo każde powinno mieć jakieś unikalne id.

W zależności od metod na tych samych końcówkach wykonujemy inne czynności. Dodatkowo sterując nagłówkami możemy np. określić jaki typ danych powinien być zwrócony, możemy w nich też przekazywać dane potrzebne do autoryzacji. REST pozwala na budowanie czytelnych, przejrzystych, przewidywalnych i często też samo opisujących się API. Nic dziwnego, że standard ten stał się tak popularny.

Django REST Framework

Jednym ze sposobów zastosowania podejścia RESTowego w projekcie opartym o Django jest DRF. Narzędzie to pozwala na wygodne budowanie webowych API opartych o REST.

<https://www.djangoproject.com/> <https://www.djangoproject.com/tutoria.../quickstart/>

Odpowiedzią na wyzwania związane z REST jest projekt **Django Rest Framework** (dalej DRF). Umożliwia on wygodną integrację z kodem Django różnego rodzaju serializerów i budowanie odpowiednich odpowiedzi API.

Kluczową rolę w tym procesie pełnią serializerzy. Możemy o nich pomyśleć trochę jak o formularzach, pełnią one podobną funkcję i podobnie się z nimi pracuje.

Pierwszy krok to zainstalowanie rozszerzenia.

```
1 $ pip install djangorestframework
```

Drugim krokiem jest dodanie aplikacji do INSTALLED_APPS.

```
1 INSTALLED_APPS = [
2     ...
3     'rest_framework',
4     'maths',
5     'posts',
6     'greetings',
7 ]
```

Obiekty wystawione poprzez API powinny być w formacie tekstowym. Najpopularniejsze do tego typu zadań to np. JSON, XML czy YAML.

Proces, w którym obiekty zamieniamy na tekst, nazywamy serializacją, a odwrotny proces deserializacją. Oba procesy realizowane są w DRF poprzez specjalne klasy – serializerzy.

W tym momencie możemy zacząć tworzyć nasze serializerzy. Odpowiednim dla nich miejscem jest plik serializers.py. Zrobimy to dla aplikacji posts.

```
1 from rest_framework import serializers
2
3 from posts.models import Post, Author, Tag
4
5 class PostSerializer(serializers.ModelSerializer):
6     class Meta:
7         model = Post
8         fields = ('id', 'title', 'content', 'created', 'modified', 'author', 'image', 'tags')
9
10    class AuthorSerializer(serializers.ModelSerializer):
11        class Meta:
12            model = Author
13            fields = ('id', 'nick', 'email')
14
15    class TagSerializer(serializers.ModelSerializer):
16        class Meta:
17            model = Tag
```

```
18     fields = ('id', 'word', 'created')
```

; Po tym kroku należy zdefiniować odpowiednie dla naszego API widoki. Chcemy tu obsłużyć operacje CRUD. Dla postów możemy to opisać w następującej tabeli

Operacja	Metoda	Typ końcówki	Koncowka
Create	POST	list	/api/v1/posts
Retrieve many	GET	list	/api/v1/posts
Retrieve one	GET	detail	/api/v1/posts/
Update	PUT/PATCH	detail	/api/v1/posts/
Delete	DELETE	detail	/api/v1/posts/

Możemy oczywiście pisać każdy widok z osobna, ale DRF dostarcza nam `ViewSet`, czyli przygotowany pod te zadania zestaw widoków. Obsługę tych widoków umieścimy w osobnym pliku. Nazwijmy go `api_views.py`. Plik powinien być oczywiście w naszej aplikacji.

```
1 from rest_framework.viewsets import ModelViewSet
2
3 from posts.models import Post
4 from posts.serializers import PostSerializer
5
6 class PostViewSet(ModelViewSet):
7     queryset = Post.objects.all()
8     serializer_class = PostSerializer
```

Kolejne ułatwienie przychodzi, gdy chcemy opisać routing, czyli obsługę adresów. Moglibyśmy oczywiście dodawać tu osobne wpisy w pliku `urls.py` w naszej aplikacji, ale zamiast tego lepiej posłużyć się dostarczonymi przez DRF routerami.

W katalogu `jazzy` (tam, gdzie znajduje się `settings.py`) utwórzmy plik o nazwie `api.py` z następującą zawartością:

```
1 from rest_framework import routers
2 from posts import api_views as posts_views
3
4 router = routers.DefaultRouter()
5 router.register('posts', posts_views.PostViewSet)
```

Następnie w głównym `urls` dodajmy `import`.

```
1 from jazzy.api import router
```

Wprowadźmy też ścieżkę:

```
1 urlpatterns = [  
2     ...  
3     path('api/v1/', include(router.urls)),  
4     ...  
5 ]
```

W tym momencie, jeśli wejdziemy na ten adres, powinniśmy zobaczyć taki widok:

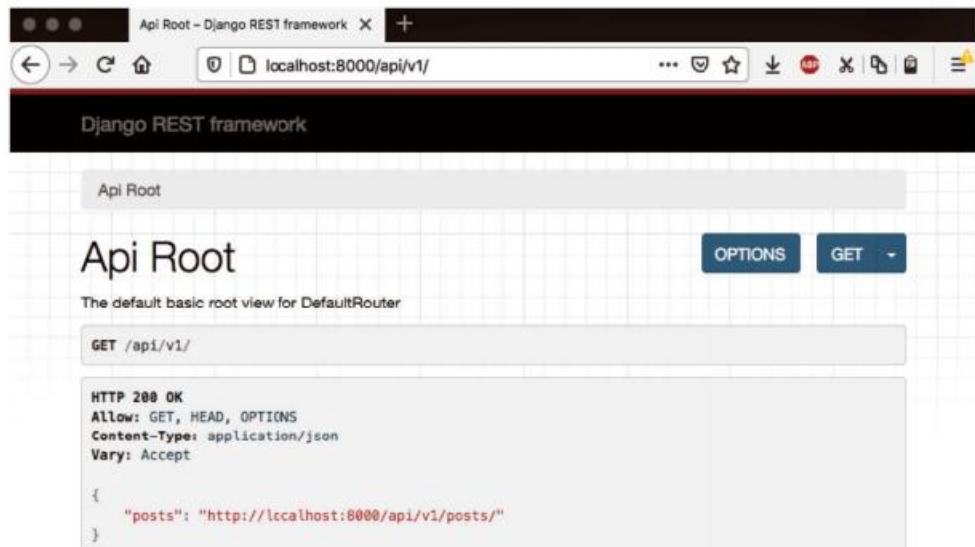


Figure 0.12: image-20210611091550599

Jest to punkt wejścia do naszego API. Od razu widzimy, co możemy zrobić dalej. De facto jest to strona internetowa opisująca nasze API. Możemy o tym pomyśleć jako jego dokumentacji. Jeśli do linka dodamy określenie formatu, to zobaczymy go w odpowiednim: `localhost:8000/api/v1/?format=json`.

W narzędziach do zapytań takich jak HTTP Client czy Postman, domyślny rezultat będzie JSON-em, a nie stroną HTML. Zawsze też można dodać odpowiedni nagłówek w zapytaniu.

Zapytanie:

```
1 GET localhost:8000/api/v1/  
2 Content-Type: application/json
```

Odpowiedź:

```
1 GET http://localhost:8000/api/v1/
2
3 HTTP/1.1 200 OK
4 Date: Wed, 05 Aug 2020 12:20:10 GMT
5 Server: WSGIServer/0.2 CPython/3.8.2
6 Content-Type: application/json
7 Vary: Accept, Cookie
8 Allow: GET, HEAD, OPTIONS
9 X-Frame-Options: DENY
10 Content-Length: 47
11 X-Content-Type-Options: nosniff
12
13 {
14     "posts": "http://localhost:8000/api/v1/posts/"
15 }
16
17 Response code: 200 (OK); Time: 28ms; Content length: 47 bytes
```

Jeśli w przeglądarce wejdziemy na adres listy postów, zobaczymy, że w formie strony otrzymamy od razu formularz do utworzenia nowego postu.

The screenshot shows a Django REST framework API interface. At the top, there's a navigation bar with 'Api Root' and 'Post Viewset List'. Below it, a title 'Post Viewset List' is displayed with 'OPTIONS' and 'GET' buttons. A URL 'GET /api/v1/posts/' is shown. The main area has a red header 'Django REST framework' and a black header with 'Content-Type: application/json' and 'Vary: Accept'. Below this, a JSON response is shown:

```
content-type: application/json
Vary: Accept

{
    "id": 16,
    "title": "Kolacja we dwoje",
    "content": "Romantyczna kolacja z najlepszym jedzeniem",
    "created": "2020-08-01T12:20:47.204197Z",
    "modified": "2020-08-01T12:20:47.204246Z",
    "author": 2,
    "image": "http://localhost:8000/media/photos/2020/08/01/kolacja-we-dvoje_rwpIyIr.jpg",
    "tags": [
        2,
        3
    ]
}
```

At the bottom, there's a form with fields: 'Title' (REST jest super!), 'Content' (Django Rest Framework wymagał), 'Author' (<Author: rkorzen>), 'Image' (Przeglądaj... Nie wybrano pliku.), and 'Tags' (dramat, komedia, wspaniałe). A 'POST' button is at the bottom right.

Figure 0.13: image-20210611091758902

Możemy też wypełnić odpowiednie pola w zakładce "RAW Data".

The screenshot shows a POST request configuration window. At the top right are two buttons: "Raw data" and "HTML form". The "Raw data" button is selected. Below it, the "Media type:" dropdown is set to "application/json". The "Content:" field contains the following JSON object:

```
{  
    "title": "",  
    "content": "",  
    "author": null,  
    "image": null,  
    "tags": []  
}
```

At the bottom right of the form is a blue "POST" button.

Figure 0.14: image-20210611085824169

Po wysłaniu formularza, w odpowiedzi otrzymamy utworzony post.

Post Viewset List

[OPTIONS](#)[GET ▾](#)**POST /api/v1/posts/**

HTTP 201 Created
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{  
    "id": 17,  
    "title": "REST jest super!",  
    "content": "Django Rest Framework wymiata!",  
    "created": "2020-08-05T12:31:33.208828Z",  
    "modified": "2020-08-05T12:31:33.208871Z",  
    "author": 1,  
    "image": null,  
    "tags": [  
        3  
    ]  
}
```

Figure 0.15: image-20210611085844621

W podobny sposób możemy zadziałać w kliencie HTTP.

Żądanie:

```
1 POST localhost:8000/api/v1/posts/  
2 Content-Type: application/json  
3  
4 {  
5     "title": "Życie bez DRF to dramat!",  
6     "content": "Jak ja sobie tradziem bez tego?",  
7     "author": 1,  
8     "image": null,  
9     "tags": [  
10         1  
11     ]  
12 }
```

Odpowiedź:

```
1 POST http://localhost:8000/api/v1/posts/  
2  
3 HTTP/1.1 201 Created  
4 Date: Wed, 05 Aug 2020 12:35:57 GMT  
5 Server: WSGIServer/0.2 CPython/3.8.2  
6 Content-Type: application/json  
7 Vary: Accept, Cookie
```

```
8 Allow: GET, POST, HEAD, OPTIONS
9 X-Frame-Options: DENY
10 Content-Length: 206
11 X-Content-Type-Options: nosniff
12
13 {
14     "id": 18,
15     "title": "Życie bez DRF to dramat!",
16     "content": "Jak ja sobie łradziem bez tego?",
17     "created": "2020-08-05T12:35:57.211628Z",
18     "modified": "2020-08-05T12:35:57.211692Z",
19     "author": 1,
20     "image": null,
21     "tags": [
22         1
23     ]
24 }
25
26 Response code: 201 (Created); Time: 33ms; Content length: 204 bytes
```

Proste i piękne, prawda? Oczywiście jest tu jeszcze cała masa zagadnień, które warto by poruszyć, ale jest to już materiał na osobny kurs.